ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

MÁSTER OFICIAL EN VISIÓN ARTIFICIAL

Trabajo Fin de Máster

Autolocalización visual aplicada a la Realidad Aumentada

Autor: Alejandro Hernández Cordero

Tutor: Prof. Dr. José María Cañas Plaza

Curso académico 2013/2014

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

http://jderobot.org/index.php/Ahcorde-tfm

"There are two problems in moder science:
-too many people use different terminology to solve the same problems
-even more people use the same terminology to address completely differents issues"

Anonymous

Agradecimientos

Lo primero de todo agradecer a Jose María, tutor de este Trabajo Fin de Máster, por todo su apoyo y libertad a la hora de realizar mi trabajo. No solo durante este proyecto si no durante los últimos años en el Grupo de Robótica de la Universidad Rey Juan Carlos.

Este trabajo no hubiera sido posible sin la ayuda y orientación del resto de profesores el Máster de Visión. A todo ellos, muchas gracias!

Gracias por los buenos momentos que he vivido con los compañeros del Máster y de despacho durante el último año, especialmente a Borja, Edu y Gonzalo (aunque siempre serás *Águila*).

Y por supuesto agradecer a mi familia su apoyo incondicional. Sin ellos hoy no estaría donde estoy hoy.

Gracias a todos!

Resumen

Uno de los campos más atractivos dentro de la visión artificial es la autolocalización visual. Este tipo de algoritmos abre la puerta a aplicaciones con teléfonos móviles ó tabletas. En este área se puede destacar el *Proyecto Tango de Google*, *Magic Plan* o las *Google Glasses*. Este Trabajo Fin de Master se encuadra dentro de la autolocalización visual, y el objetivo concreto que se ha abordado es el diseño y la programación de varios algoritmos que estiman en tiempo real la posición y orientación de una cámara móvil utilizando exclusivamente las imágenes obtenidas por la cámara. Se explican diferentes técnicas partiendo de una técnica sencilla como es DLT hasta técnicas más complejas como monoSLAM o PTAM.

El algoritmo se ha validado experimentalmente haciendo uso de una cámara de videoconferencia y en un dispositivo Android. El algoritmo calcula la trayectoria realizada y la muestra en un ventana OpenGL, junto con un modelo de la cámara, y además se integra con un videojuego de Realidad Aumentada.

Para el desarrollo del proyecto se ha utilizado JdeRobot 5.2 como plataforma software. Esta plataforma ha permitido reutilizar componentes desarrollados en otros proyectos de forma sencilla. Se ha utilizado el lenguaje de programación C++, ICE para los interfaces de comunicación, Qt como biblioteca gráfica, OpenGL y OGRE para representar modelos 3D, OpenCV para el procesamiento de las imágenes y Eigen para los cálculos de álgebra lineal.

Índice general

Índice de figuras

Capítulo 1

Introducción

La vista es el sentido más utilizado por los seres humanos para captar la información de nuestro entorno. Esta habilidad consiste en la capacidad para detectar la luz e interpretarla. Tanto los seres humanos como los animales poseen un sistema perceptivo que nos permite crear un esquema de nuestro entorno, averiguar detalles de los objetos que nos rodean. Debido a estos detalles es posible reconocer los objetos por su color, por su forma, detectar si existe movimiento en ellos o incluso estimar aproximadamente la distancia que nos separa. Además, la vista es un importante apoyo al sentido del equilibrio.

La evolución de la visión no se debe sólo a la mejora de los órganos sensoriales, los ojos, sino a una cerebro preparado para procesar la toda la información que recibe. Esta tarea incluye un proceso de selección, ya que no toda la información que llega al ojo es útil. La memoria desempeña un papel fundamental en el proceso de visión y ayuda a clasificar los objetos de forma natural.

La visión artificial es la reproducción de estas habilidades en máquinas mediante el uso de sistemas informáticos. Las cámaras son sensores de bajo coste cada vez más comunes en nuestra vida cotidiana. Este abaratamiento junto al aumento de la capacidad de cómputo de los ordenadores han reavivado el interés en la visión por ordenador, que ha crecido enormemente en los últimos años.

En este primer capítulo se decribe brevemente el contexto en el que se ha desarrollado este Trabajo Fin de Máster, la visión artificial de modo genérico, la realidad aumentada y la actualidad sobre este tipo de tecnologías.

1.1. La visión artificial

La visión artificial es un campo de la inteligencia artificial que pretende obtener información del mundo a partir de una o varias imágenes, que normalmente vienen dadas de forma de matriz numérica. La información relevante que se puede obtener a partir de las imágenes puede ser el reconocimiento de objetos, la recreación en 3D de la escena que se observa, el seguimiento de un objeto, etc.

El inicio de la visión artificial se produjo en 1961 por parte de Larry Roberts, quien creó un programa que podía ver una estructura de bloques, analizar su contenido y reproducirla desde otra perpectiva, utilizando para ello una cámara y procesando la imagen desde un ordenador. Sin embargo, para obtener el resultado las condiciones de la prueba estaban muy controladas. Otros muchos científicos también han tratado de solucionar el problema de conectar una cámara a un ordenador y hacer que éste describa lo que ve. Finalmente, los científicos de la época se dieron cuenta de que esta tarea no era sencilla de realizar, por lo que se abrió un amplio campo de investigación, que tomó el nombre de visión artificial.

En este campo se persigue, por ejemplo, que el ordenador sea capaz de reconocer en una imagen distintos objetos al igual que los humanos lo hacemos con nuestra visión. Se ha demostrado que este problema es muy complejo y que algo que para nosotros resulta automático puede que se tarde mucho tiempo en que lo resuelva una máquina. Por otra parte, a pesar del alto precio computacional que se paga por utilizar cámaras como sensor, si se consigue analizar correctamente la imagen, es posible extraer mucha información de ella que no podría obtenerse con otro tipo de sensores.

En los años 90 empezaron a aparecer los primeros ordenadores capaces de procesar las imágenes lo suficientemente rápido. Además se comenzó a dividir los posibles problemas de la visión artificial en otros más específicos que pudiesen resolverse. A partir de entonces la visión artificial comenzó a emplearse para múltiples tareas y su desarrollo fue concentrándose en problemas como la segmentación, el reconocimiento de formas o el filtrado de bordes. La utilización de estas técnicas hace posible el desarrollo de sistemas que persiguen objetivos concretos tales como el reconocimiento facial o de iris, reconocimiento de objetos o el seguimiento 2D de objetos. Por ejemplo, en robótica la visión puede utilizarse para la navegación de los robots así como para la autolocalización o reconstrucción de lo que el robot está viendo.

A continuación se presentan algunas aplicaciones y escenarios de aplicación, no sólo en el área académica sino también en la vida cotidiana, que hacen uso de los descubrimientos

y avances logrados por diversos investigadores en visión artificial.

- *Robótica*: Unos de los procesos más complejos que tiene que realizar un robot es interpretar el mundo a su alrededor a través de la información que adquiere mediante los sensores, como puede ser una cámara de vídeo. Esta información puede usarse para la localización o reconocimiento de objetos o incluso el seguimiento de los mismos.

- *Videojuegos*: Este sector ha contribuido notablemente al desarrollo de la visión artificial. Ejemplos claros de su uso son el dispositivo *Kinect* desarrollado por *Microsoft* (cámara RGBD), y *Eye Toy*, desarrollado por *Play Station* para el reconocimiento de los gestos realizados por los jugadores. En la nueva generación de consolas de videojuegos se está avanzando en mejorar la interacción jugador-consola.

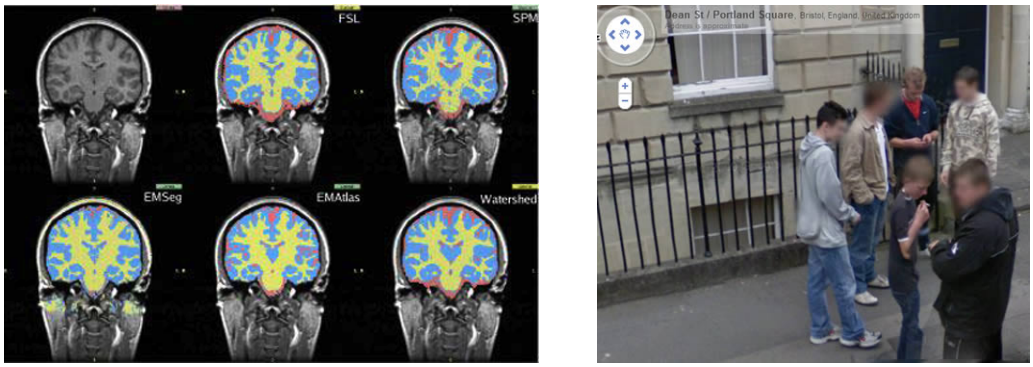
(a) (b)

Figura 1.1: Sensores basados en visión para videoconsolas

- *Compra visual*: Consiste en tomar una foto de un producto en una tienda para buscar en internet información sobre dicho producto o similares y así comparar sus características, calidad o precio. Una empresa innovadora en este sector es *OculusAI* con su aplicación móvil *Productify*.

- *Medicina*: Uno de los ojetivos de la visión artificial es el tratamiento y análisis de imágenes, detección de patrones y reconstrucción 3D para ayudar al correcto diagnóstico por parte del especialista clínico. Aplicaciones comunes aquí son la detección y caracterización automática de tumores, mejora de la imagen de microscopía o análisis hiperespectral para extraer la composición de los tejidos orgánicos contenidos en una imagen.

- *Reconocimiento de objetos*: Su uso está muy extendido en controles de aduanas y aeropuertos, para detectar objetos que pueden ser peligrosos o estén prohibidos.



(a) (b)

Figura 1.2: (a) Imágenes de neurorrehabilitación (b) Emborronamiento de caras de Google Street View

También sirve para recopilar información estadística, desde el número de usuarios del metro hasta información personal de los usuarios escaneando su DNI o pasaporte. Uno de los usos más extendidos son los lectores de códigos de barras de los productos en supermecados y códigos QR [1]

- *Biometría*: La biometrá estudia los métodos automáticos para el reconocimiento único de humanos basados en uno o más rasgos conductuales o rasgos físicos intrínsecos. Dentro de estos sistemas se encuentran los sistemas de reconocimiento facial. Estos son muy usados en seguridad, como un ejemplos de ello se encuentran los sistemas de control de acceso a los laboratorios de la *NASA*, el sistema de vigilancia de la ciudad de Londres que realiza un reconocimiento facial mediante cámaras distribuidas por la ciudad; o renocomiento de caras y posterior emborranado automático de cara de *Google Street View* (Figura 1.2(b)).

1.2. Autolocalización visual

Dentro de la visión artificial un campo interesante es la autolocalización que consiste en conocer la posición de la cámara y hacia dónde mira, a partir de las imágenes que se reciben por el sensor y sin ninguna información adicional. Se ha abordado desde 2 comunidades distintas a las vez. Por un lado, la comunidad de visión artificial denominó el problema como *Structure from Motion*, donde la información es procesada por lotes. Y también por la comunidad robótica que denomina el problema como *SLAM Visual* y tiene restricciones

[1] QR code es el nombre de una marca comercial de un tipo de código de barras bidimensional.

como por ejemplo el tiempo real o la robustez.

1.2.1. Structure from Motion

Dentro de la visión artificial el *Structure from Motion* (SfM) es la línea de investigación que toma como entrada únicamente un conjunto de imágenes, y pretende conocer de manera totalmente automática la estructura 3D de la escena vista y las ubicaciones de las cámaras donde las imágenes fueron capturadas. El SfM es una de las áreas más atractivas de investigación en la última década. Ha llegado a un estado de madurez donde alguno de los algoritmos tienen aplicación comercial [Microsoft, 2011] (Figura 1.3).

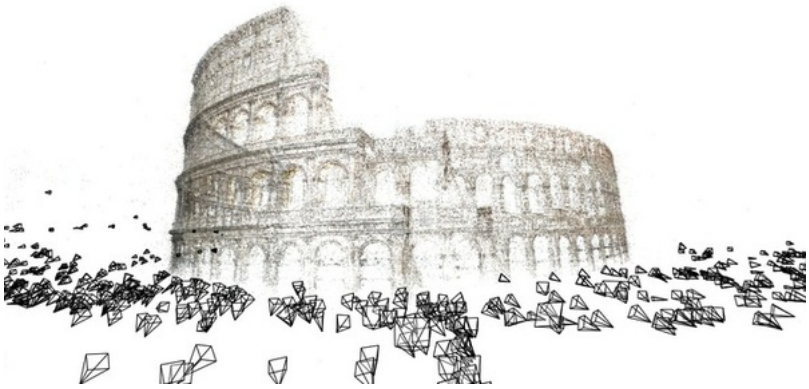


Figura 1.3: PhotoTourism. Microsoft

El SfM surge en la segunda mitad del siglo XIX, denominado fotogrametría, que tuvo como objetivos extraer información geométrica de las imágenes a partir de un conjunto de características manualmente identificadas por el usuario. La fotogrametría hace uso de técnicas de optimización no lineales como el ajuste de haces para reducir al mínimo error de retroproyección. El problema abordado por la comunidad de visión artificial ha sido en su mayoría lograr la completa automatización del proceso. Esto provocó avances en tres aspectos: en primer lugar, restricciones impuestas al movimiento bajo el supuesto de rigidez de la escena; en segundo lugar, la detección de características y descriptores [Canny., 1986], [Harris and Stephens, 1988]; y por último, la eliminación de espúreos.

Dadas multiples vistas, un punto tridimensional, puede ser reconstruido mediante triangulación. Un prerequisito importante es determinar la calibración de la cámara. Se puede representar por un matriz de proyección. La teoría geométrica de SfM permite calcular las matrices de proyección y los puntos 3D utilizando solo correspondencia entre los puntos de cada imagen. Para mejorar el rendimiento del SfM se puede aprovechar el

conocimiento sobre la escena, con el fín de reducir el número de grados de libertad. Por ejemplo, las restricciones que imponen el paralelismo y la coplanaridad, pueden utilizarse para reconstruir formas geométricas simples como líneas y polígonos planos, a partir de sus posiciones proyectas envistas simples.

1.2.2. Visual SLAM

En paralelo a SfM, la estimación del movimiento de un robot móvil y de su entorno ha sido abordado por la comunidad robótica desde un punto de vista diferente. El *SLAM* (*Simultaneous Localization and Mapping*) es uno de los problemas fundamentales en la investigación robótica móvil. Por un lado, estima el movimiento de un robot móvil, y por otro, construye un mapa de los alrededores del robot a partir del flujo de datos de entrada proporcionado por uno o varios sensores (odometria, láseres, sonar o visión, entre otros) [Castellanos and Tardós, 1999], [H.J.S. Feder and Smith, 1999] o [Montiel and Zisserman., 2003]. En la mayoría de las ocasiones las mediciones de la odometría también se incluyen en la estimación y la fusión sensorial se lleva a cabo con frecuencia. En los últimos años los algoritmos de visión por ordenador han madurado los suficiente como para tener una posición predominante en los esquemas de fusión de sensores e incluso han utilizado visión como la única entrada sensorial. El SLAM monocular se refiere a la utilización de una única cámara como sensor predominante en la realización de SLAM [A. J. Davison and Stasse., 2007].

El SfM y el SLAM monocular presentan diferencias. Por un lado, el SfM ha tratado de resolver el problema de un forma más general, es decir, de cualquier tipo de información visual. Por otro lado, el SLAM monocular se ha centrado en el enfoque secuencial procesando secuencias de video en tiempo real. Esto es consecuencia de la aplicación: Un robot móvil necesita una estimación de la posición de manera continua con el fin de introducirla en el bucle de control, y por tanto, el procesamiento por lotes no tiene sentido.

Esta restricción de procesamiento secuencial en tiempo real también se aplica a otras áreas como la Realidad Aumentada, con el fin de insertar de manera coherente y sin discontinuidades los objetos virtuales en cada fotograma. Por lo tanto, en Realidad Aumentada se puede hacer uso de algoritmos de SLAM Visual como el propuesto por George Klein [Klein and Murray., 2007].

1.3. Realidad aumentada

Una de las aplicaciones fundamentales de la autolocalización visual es servir de base para sistemas con Realidad Aumentada. Sabiendo en tiempo real dónde está la cámara en 3D se puede enriquecer sintéticamente la imagen con objetos ficticios de modo realista, como si esos objetos estuvieran en la escena real física.

El término Realidad Aumentada se usa para definir una visión directa o indirecta de un entorno del mundo real combinado con elementos virtuales, es decir, la creación de una realidad mixta en tiempo real. Consiste en un conjunto de dispositivos que añaden información virtual a la información física ya existente. Esta relativamente nueva tecnología ofrece oportunidades únicas para ampliar el alcanze del ser humano en de múltiples disciplinas, como la medicina, la arquitectura, la navegación o la educación.

En la actualidad se aceptan dos definiciones para explicar qué es la Realidad Aumentada. La primera fue realizada por Paul Milgram y Fumio Kishino en 1994 [Milgram and Kishino, 1994]. Definieron un concepto que llamaron *Continuo Virtualidad − Virtuality Continuun −* (Figura 1.4) que consiste en un escala continua entre lo completamente virtual y lo completamente real. El continuo realidad-virtualidad abarca todas las posibles variaciones y composiciones de objetos reales y virtuales. El área que está entre los extremos, donde tanto lo real como lo virtual se mezclan, recibe el nombre de Realidad Mixta. Ésta a su vez se dice que consta de Realidad Aumentada, donde lo virtual aumenta lo real, y la Virtualidad Aumentada, donde lo real aumenta la virtual.

Figura 1.4: Virtuality Continuun

La segunda de las definiciones la hizo el Doctor Ronald T. Azuma en 1997 [Azuma, 1997], donde afirma que para poder considerarse Realidad Aumentada, ésta ha de mezclar eventos de la realidad con elementos inexistentes en la misma, es decir, Realidades Virtuales. El individuo que la experimente ha de poder interactuar con ella en tiempo real y ha de estar en tres dimensiones.

Las tecnologías para producir Realidad Aumentada se pueden englobar en dos categorías: hardware y software, que se resumen a continuación.

1.3.1. Tecnologías Hardware

Los dispositivos dedicados a la Realidad Aumentada deben contar con un sistema de visualización para aumentar la realidad. A continuación se explicarán tres técnicas para la presentación de la Realidad Aumentada:

- *Display montado sobre la cabeza del usuario*: Este tipo de pantallas se conoce con el acrónimo de HMD (*head-mounted display*). Se trata de dispositivos con una lente muy próxima al ojo de usuario, que le permite ver el mundo real. Se proyectan los elementos virtuales, que con la combinación de la visión del mundo real, genera la Realidad Aumentada. La proyección de la Realidad Aumentada se puede realizar sobre una lente o directamente sobre la retina del usuario, en este caso el HMD recibe el nombre de *monitor virtual de retina*.

  Los HMD se pueden clasificar en base a dos parámetros. Según su configuración física sobre la cabeza del usuario: *monocular* cuando las imágenes virtuales sólo se pueden ser percibidas por un ojo, y *binoculares* cuando las imágenes pueden ser percibidas por ambos ojos, obteniendo de esta manera una imagen estereoscópica. También se pueden clasificar según su propósito: HMD destinados a Realidad Virtual o destinados a la representación de Realidad Aumentada, es decir, según el campo de visión del mundo real que se le permita ver. Cuando el dispositivo HMD reduce mucho el campo de visión del usuario, éste se ve totalmente inmerso en Realidad Virtual, ya que sólo será capaz de ver las imágenes creadas artificialmente.

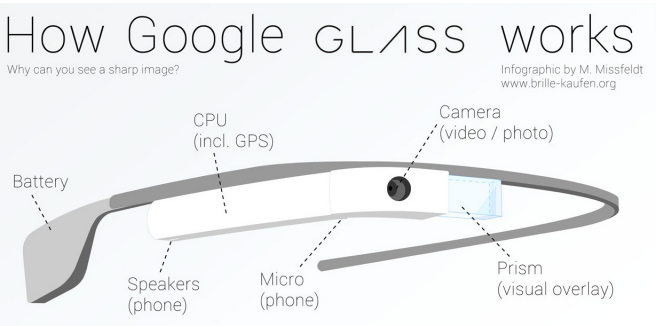


Figura 1.5: Google Glass

Un ejemplo actual de dispositivos dedicados a Realidad Aumentada es el *Project Glass* (Figura 1.5) que está realizando Google. Las gafas prototipo fueron presentadas en Abril de 2012 en Google+. Este concepto no es nuevo, lo que realmente destaca de este proyecto es el reducido tamaño de las gafas frente a otras soluciones ya existentes.

Además, Google ha comunicado su intención de llevar este dispositivo a uno nuevo que permita la integración con cualquier gafa convencional existente. El *Project Glass* ha tenido tanta repercusión debido a las posibilidades reales de implantación en el mercado así como a su gran potencial.

- *Display de mano* (Figura 1.6(a)) Estos sistemas constan de pantallas portátiles cuyo tamaño se ajusta a la mano de usuario, donde la Realidad Aumentada se monta sobre las imágenes de la cámara digital de la que dispone el dispositivo.

  Los primeros dispositivos con *display* de mano empleaban para realizar el seguimiento del dispotivo marcadores, más adelante comenzaron a utilizar geoposicionamiento GPS, brújulas digitales o acelerómetros/giróscopos de seis grados de libertad. Posteriomente se añadió información digital a las secuencias de vídeo en tiempo real, utilizando sistemas de visión.

(a)

(b)

Figura 1.6: (a) Display de mano (b) Display espacial

- *Display espacial*: (Figura 1.6(b)) En lugar de que el usuario tenga que llevar puesto o tenga que sostener el dispositivo, los sistemas de Realidad Aumentada Espacial (SAR, *Spatial Augmented Reality*) hacen uso de proyectores digitales para mostrar información gráfica sobre objetos físicos.

1.3.2. Tecnologías Software

Este tipo de software proporciona a dispositivos electrónicos tales como *Smartphones* o *Tablets*, la funcionalidad de Realidad Aumentada. El software es capaz de fusionar coherentemente entre las imágenes obtenidas con la cámara del dispositivo del mundo

real y las imágenes virtuales en 3D. Este proceso involucra diversos métodos de visión artificial, gran parte de ellos relacionados con el seguimiento en secuencias de vídeo.

En el mercado se encuentrán diversas soluciones software de Realidad Aumentada:

- *ArUco*[2]: Biblioteca reducida para el desarrollo de aplicaciones de Realidad Aumentada basada en OpenCV. Licencia BSD.
- *ARToolKit*[3]: Biblioteca que emplea funcionalidades de seguimiento de vídeo en tiempo real para calcular la ubicación de la cámara y la orientación relativa respecto a la posición de los marcadores físicos que se sitúan en el mundo real.
- *Metaio Mobile SDK*[4]: Proporciona funciones de seguimiento natural. Disponible para Android e IPhone.
- *Qualcomm AR SDK (Vuforia)*[5]: Detección y seguimiento de imágenes de referencia y marcadores usando características de detección natural. Disponible para Android e IPhone.

1.3.3. Aplicaciones

La Realidad Aumentada proporciona una nueva forma de interacción hombre-máquina que hace que las tecnologías relacionadas (autolocalización visual, generación de imágenes enriquecidas...) se apliquen en diferentes campos:

- *Entretenimiento*: Uno de los campos de aplicación más importante de la Realidad Aumentada son los videojuegos (Figura 1.7(a)), ya que proporciona una forma novedosa de interactuar con el entorno. En los últimos años han ido ganando popularidad plataformas de juegos con controles no tradicionales, que se escapan del mando clásico como es el caso de la *Wii* de *Nintendo* o los juegos tactiles de los *Smartphones*.
- *Publicidad*: Se utilizan anuncios interactivos susceptibles de proporcionar interactividad para llamar la atención de usuario (Figura 1.7(b)).

[2]http://www.uco.es/investiga/grupos/ava/node/26
[3]http://www.hitl.washington.edu/artoolkit/
[4]http://www.metaio.com/sdk/
[5]https://developer.qualcomm.com/mobile-development/add-advanced-features/augmented-reality-vuforia

(a) (b)

Figura 1.7: (a) Imágenes de videojuego Invizimals (b) Publicidad mediante Realidad Aumentada en un supermercado

- *Militares*: Aquí uno de sus posibles usos es proporcionar información del campo de batalla en tiempo real, ya sea en cascos HMD o vehículos militares. En aviones, los llamados HUD (*head-up displays*, Figura 1.8(a)) son considerados como uno de los pioneros de la Realidad Aumentada, permitiendo al piloto obtener información como la velocidad o altitud sin necesidad de dejar de mirar hacia otro lado.

(a) (b)

Figura 1.8: (a) Head-up displays de un avión de combate (b) Proyección de información en la luna de un coche

- *Dispositivos de navegación*: Facilitan el uso de la navegación dentro de un edificio mostrando dónde se ubican cada una de las estancias del mismo. Incluso en automóviles emplear la luna para superponer información de tráfico, destino, etc. En la Figura 1.8(b) se puede observar la información de velocidad proyectada sobre la luna del coche.

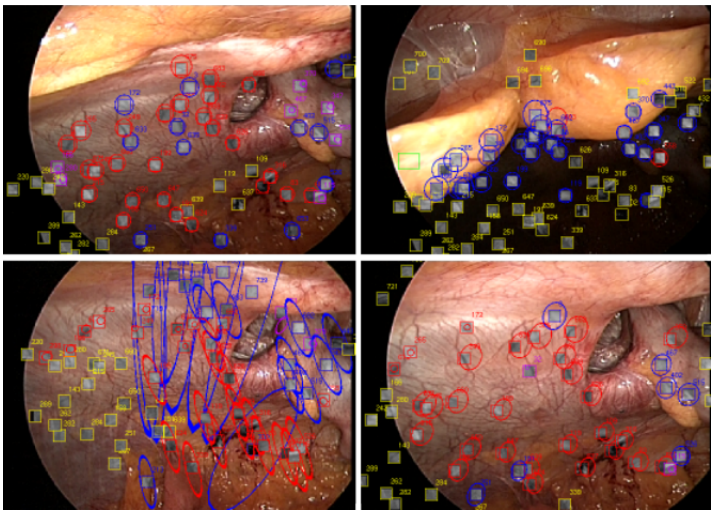
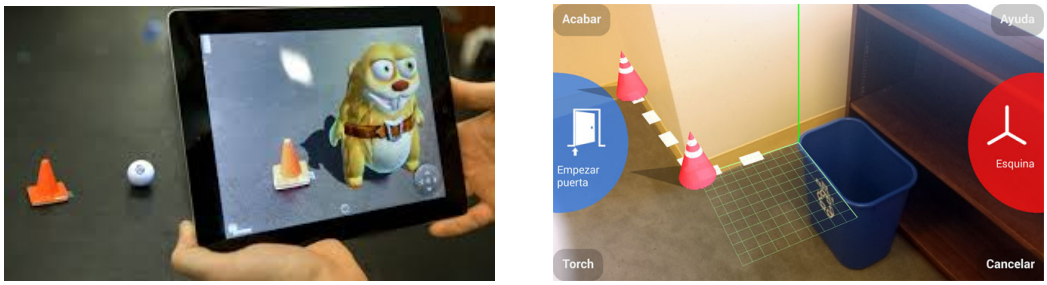


Figura 1.9: Imagen de una operación utilizando laparoscopia

- *Medicina*: No es de extrañar que gracias a su potencial, este ámbito haya atraido a investigadores de la medicina. Por ejemplo se permite superponer la superficie del cuerpo la reconstrucción 3D de las estructutras internas del paciente. También se utiliza para el tratamiento de fobias, agregando virtualmente el objeto causante en el campo de visión del paciente. Una aplicación de estos algoritmos, aunque no hace uso de Realidad Aumentada, se usa en un quirófano ['Oscar G. Grasa, 2011] (Figura 1.9) donde se utiliza la autolocalización de la cámara y su capacidad para realizar medidas con bastante precisión para medir el tamaño de una prótesis.

En la actualidad existe un gran número de empresas desarrollando tanto software como hardware para la creación de soluciones de Realidad Aumentada, entre ellas podemos encontrar:



(a) (b)

Figura 1.10: (a) Sphero (b) MagicPlan

Orbotix[6] es una empresa americana que ha conseguido unir la robótica con la Realidad Aumentada. Esta empresa comercializa un pequeño robot, Sphero, con forma de esfera que es manejable a través de un dispositivo móvil mediante una conexión Bluetooth. Combinando el robot con la cámara del telefono móvil permite jugar a juegos de Realidad Aumentada, como se puede ver en la Figura 1.10(a).

La aplicación MagicPlan[7] permite con muy poco esfuerzo medir y crear planos de casas con la ayuda de la cámara de nuestro dispositivo móvil usando Realidad Aumentada o de forma manual dibujando el plano con sus herramientas de dibujo. En la Figura 1.10(b) se puede observar una captura del sistema en funcionamiento.

(a) (b)

Figura 1.11: Structre Sensor Occipital

La *startup* americana Occipital[8] el pasado año lanzó una campaña de *crowdfunding* y *pre-order* en *Kickstarter* conseguiendo más de 1.500.000 dólares gracias a un escaner 3D de bajo coste (Structure Sensor 1.11(a)) especialmente diseñado para plataformas móviles. Con este dispositivo se podrán digitalizar objetos en 3D en tiempo real, digitalizar el entorno o crear aplicaciones de Realidad Aumentada. Este dispositivo puede ser muy útil para determinados profesionales, por ejemplo arquitectos o personas relacionadas con la impresión 3D.

Dentro de las fronteras españolas se puede encontrar una aplicación de Realidad Aumentada exitosa que está desarrollando la empresa Seabery[9]. En concreto, se trata de una herramienta denominada *Soldamatic* (Figura 1.12), un método de formación en soldadura por simulación con Realidad Aumentada que está cambiando de forma radical la educación de los futuros soldadores.

[6]http://www.gosphero.com/es/
[7]http://www.sensopia.com/english/index.html
[8]www.occipital.com
[9]http://seabery.es/

Figura 1.12: Soldamatic

1.3.4. Realidad Aumentada en el Grupo de Robótica

En el Grupo de Robótica de la Universidad Rey Juan Carlos se han desarrollado varios trabajos sobre calibración de cámaras y autolocalización de cámaras, que conforman los antecedentes del presente Trabajo Fin de Máster. Se han realizado trabajos relacionados con la calibración de cámaras, donde a partir de este conocimiento se puede estimar la posición 3D de un objeto o medir distancias desde una imagen o varias. Se pueden destacar los proyectos realizados por Redo [Kachach, 2008] y Agustín [Díaz, 2013]. Se ha trabajado también con sensores RBGD [Cantero, 2013] de bajo coste como *Kinect* (*PrimeSense*) o *Xtion* (*Asus)* diseñando e implementando un calibrador de dispositivos RGBD utilizando un patrón 3D conocido. El trabajo realizado por Luis Miguel [Ramos, 2010] sobre monoSLAM estima en tiempo real la posición y orientación de una cámara móvil autónoma en un entorno estático utilizando sus exclusivamente imágenes. Actualmente se está desarrollando una tesis doctoral relacionada con la autolocalización de un cámara en tiempo real en entornos dinámicos.

Con este Trabajo Fin de Máster se pretende dar un paso más allá sobre estos antecedentes y no sólo crear un algoritmo de localización visual robusto y fiable, dotarle además de la utilidad de Realidad Aumentada. En él se han implementado dos soluciones nuevas de autolocalización visual, una basada en DLT (*Direct Linear Transformation*) y otra refactorizando el código de PTAM. Además se ha utilizado, comprendido y modificado el código de MonoSLAM que actualmente mantiene el Grupo de Robótica de la Universidad Rey Juan Carlos.

La presente memoria detalla todos los aspectos relevantes que conlleva el desarrollo de las técnicas de localización visual, y los experimentos realizados. Se articula en 6 capítulos. El capítulo 2 fija los objetivos y requisitos planteados. El capítulo 3 ofrece una descripción

del entorno de trabajo sobre el que se ha realizado el sistema, tanto de software como hardware. Los fundamentos teóricos utilizados para el desarrollo de los algoritmos se exponen en el capítulo 4. Los experimentos y el desarrollo software realizados para la validación del sistema se exponen en el capítulo 5. Por último, en el capítulo 6 se reflejan las conclusiones extraídas haciendo alusión a los resultados obtenidos y las posibles líneas futuras de mejora por las que se puede continuar y extender este trabajo.

Capítulo 2

Objetivos

En este capítulo se exponen los objetivos principales y específicos que se pretenden alcanzar y la metodología puesta en práctica para conseguirlos.

2.1. Descripción del problema

El objetivo principal de este proyecto es crear un algoritmo de localización visual en tiempo real y hacer uso de esta información para crear una aplicación de Realidad Aumentada. El objetivo principal se ha articulado en tres subobjetivos específicos los cuales se describen a continuación:

1. Desarrollo, prueba y comparativa de distintas alternativas de algoritmos de autolocalización visual en tiempo real.

2. Desarrollo de una aplicación de Realidad Aumentada que sirva de validación experimental de lo desarrollado en el punto anterior.

3. Si el primer subobjetivo utiliza un ordenar personal como computador de referencia, en este tercer subobjetivo se integrará la aplicación en un dispositivo móvil y se probará en una tableta Android.

2.2. Requisitos

Teniendo en cuenta los objetivos marcados, el Trabajo Fin de Máster debe satisfacer también los requisitos descritos a continuación:

- **Software modular**: El sistema hará uso de la plataforma de desarrollo *JdeRobot*, que es el entorno de trabajo del Grupo de Robótica de la URJC. Esta arquitectura es multilenguaje pero la mayoría de sus componentes están desarrollados sobre el lenguaje $C/C++$. El trabajo Fin de Máster será desarrollado en Java y $C++$.

- **Plataforma**: El sistema se ejecutará tanto en un entorno GNU/Linux Debian ó Ubuntu [1] como en un dispositivo Android.

- **Visión artificial**: Para la autolocalización monocular se debe hacer uso exclusivo de una única cámara, no pudiendo utilizar otro tipo de sensores que ayuden en esta tarea. En concreto, no se puede hacer uso de técnicas estéreo o sensores RGBD.

- **Cámara**: El sistema debe funcionar con todo tipo de cámaras, a pesar de que tengan distintas distancias focales. Cada cámara real se calibrará con el software incluido en *JdeRobot* [Díaz, 2013].

- **Tiempo real**: Los algoritmos utilizados deberán ser lo suficientemente eficientes como para ser ejecutados de forma iterativa tanto por un PC como por un dispositivo Android conseguiendo una localización precisa.

- **Robustez**: El sistema debe disponer de la suficiente información como para mantener la localización de la cámara lo más ajustada a la realidad posible, funcionando incluso con cambios en la iluminación, movimientos rápidos de la cámara o pérdida de nitidez.

2.3. Metodología de desarrollo

El modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos. Este modelo permite desarrollar el proyecto de forma incremental, aumentando su complejidad progresivamente y haciendo posible la generación de prototipos funcionales.

Este tipo de modelo de ciclo de vida permite obtener productos parciales que pueden ser evaluados, total o parcialmente, facilitando la adaptación a los cambios requeridos, algo que sucede habitualmente en los proyecto de investigación.

El modelo en espiral se realiza por ciclos donde cada ciclo representa una fase del proyecto. Dentro de cada ciclo del modelo en espiral se pueden diferenciar 4 partes principales que pueden verse en la Figura 2.1, donde cada una de las partes tiene un objetivos distinto:

[1] http://www.ubuntu.com/

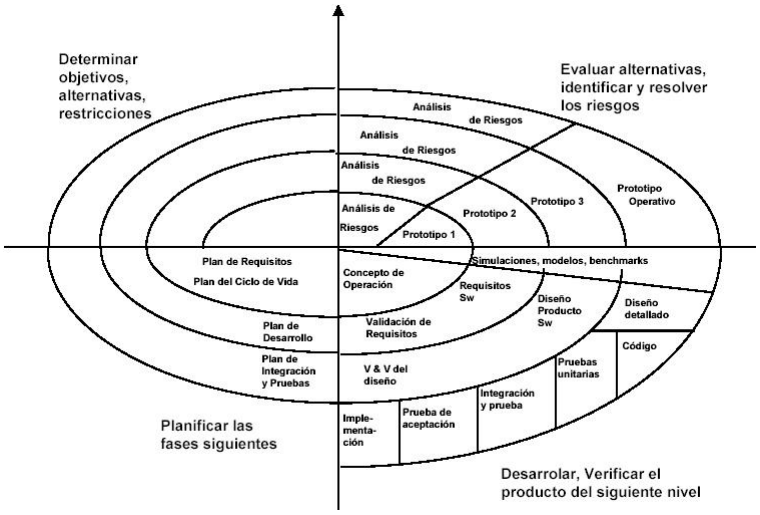


Figura 2.1: Modelo en espiral (Barry Boehm, 1986)

- Determinar objetivos: Se establecen las necesidades que debe cumplir el sistema en cada iteraciíon teniendo en cuenta los objetivos finales. Típicamente según avancen las iteraciones aumentarán el coste del ciclo y su complejidad.

- Evaluar alternativas: Determina las diferentes formas de alcanzar los objetivos establecidos en la fase anterior, utilizando distintos punto de vista, como el rendimiento que puede tener en espacio y tiempo, las formas de gestionar el sistema, etc. . Además, se consideran explícitamente los riesgos, intentando mitigarlos lo máximo posible.

- Desarrollar y verificar: Desarrollar el producto siguiendo la mejor alternativa para poder alcanzar los objetivos del ciclo. Una vez diseñado e implementado el producto se realizan las pruebas necesarias para comprobar su funcionamiento.

- Planificar: Teniendo en cuenta el funcionamiento conseguido por medio de las pruebas realizadas, se planifica la siguiente iteración revisando los posibles errores cometidos a lo largo del ciclo y comenzando un nuevo ciclo de la espiral.

Los ciclos que se han seguido en el presente Trabajo Fin de Máster está relacionados con cada una de las etapas que se describen en la siguiente sección. A lo largo de estas etapas, se han realizado reuniones semanales con el tutor del Trabajo Fin de Máster para revisar los objetivos que se pretenden alcanzar y para evaluar las alternativas de desarrollo.

2.3.1. Plan de trabajo

Durante el transcurso del Trabajo Fin de Máster, se han marcado progresivamente una serie de etapas a cumplir dividiendo el trabajo a realizar en distintas fases:

- **Familiarización con JdeRobot**: El objetivo de esta fase es aprender a utilizar el software JdeRobot y las aplicaciones y *drivers* que contiene, ya que algunos de estos formarán parte del proyecto final.
- **Desarrollo de un algoritmo de autolocalización visual basada en DLT**: El objetivo de esta fase es comprender correctamente la geometría de las cámaras y resolver el problema de una forma sencilla utilizando patrones.
- **Desarrollo de una interfaz gráfica**: El objetivo de esta fase es contruir una interfaz gráfica que muestre en tiempo real la posición y orientación de la cámara, así como el seguimiento de los puntos.
- **Desarrollo de un algoritmo de autolocalización visual basada en PTAM**: El objetivo de esta fase es hacer una implementación propia de PTAM, un algoritmo exitoso en la comunidad investigadora.
- **Piloto Seabery**: Llegados a este punto, con la tecnología de autolocalización madura se desarrollará un piloto de autolocalización visual para la empresa Seabery.
- **Videojuego de Realidad Aumentada**: Una vez desarrollado el algoritmo de autolocalización visual, se implementará un videojuego haciendo uso de él.
- **Familiarización con Android y *Java Native Interface (JNI)***: El objetivo de esta fase es estudiar el API de Android, para poder capturar imágenes y representar la Realidad Aumentada en la pantalla del teléfono o tableta. Por otro lado, aprender a compilar código C++ haciendo uso del *JNI*.
- **Integración en Android**: Una vez comprendido cómo compilar código C++ utilizando *JNI*, se pretende utilizar el código de los algoritmos desarrollados dentro de un dispositivo Android.

El seguimiento temporal detallado de este plan de trabajo está disponible en el mediawiki[2] a modo de bitácora junto con imágenes y vídeos que muestran los resultados alcanzados.

[2]http://jderobot.org/index.php/Ahcorde-tfm

Capítulo 3

Infraestructura

En este capítulo se describen los elementos empleados en el Trabajo Fin de Máster, tanto software como hardware, así como las herramientras utilizadas y que han sido de ayuda para alcanzar lo objetivos propuestos.

3.1. Elementos hardware

Para el funcionamiento del sistema se necesita una única cámara conectada con un procesador. Se han realizado pruebas con dos plataformas. Por un lado, un ordenador personal y una webcam convencional conectada por USB y, por otro lado, una tableta *Android* con una cámara integrada.

(a) (b)

Figura 3.1: Hardware utilizado

Las características del ordenador personal son un procesador Intel(R) Core(TM)2 Quad CPU Q9300 @ 2.50GHz. El sistema operativo utilizado para la implementación ha sido

Debian Wheezy, una de las distribuciones de *GNU/Linux* más importantes actualmente. Este sistema es de libre distribución y soporta oficialmente las arquitecturas hardware *Intel x86* y *AMD64* entre otras.

La cámara utilizada ha sido la *Logitech* WebCam Pro 9000 (Figura 3.1(a)). Se comunica con el PC utilizando un puerto USB. Su rendimiento típico tiene una tasa de refresco de 30fps con una resolución de 320x240px.

La tableta utilizada ha sido la *Samsung Galazy Tab 2* (Figura 3.1(b)). Cuenta con *Android* 4.0 (Ice Cream Sandwich) con un procesador Dual-Core a 1GHz y dos cámaras integradas.

3.2. JdeRobot

Este Trabajo Fin de Máster ha sido desarrollado empleando el entorno de programación multiplataforma de software libre JdeRobot[1] mantenido y mejorado a lo largo de los años por el Grupo de Robótica de la Universidad Rey Juan Carlos. La filosofía de esta plataforma es crear una herramienta que proporcione un método sencillo de desarrollar aplicaciones de robótica y visión artificial.

JdeRobot está programado fundamentalmente en C/C++, aunque existen componentes escritos en otros lenguajes de programación como Python ó Java. Cada aplicación se organiza como un conjunto de componentes comunicándose con el esquema RPC cliente-servidor (o publicación suscripción), donde cada componente se ejecuta como un proceso. Ofrece una interfaz sencilla para la programación de sistemas de tiempo real y resuelve problemas relacionados con la sincronización de los procesos y la adquisición de datos.

JdeRobot simplifica el acceso a los dispositivos hardware desde el programa, permitiendo obtener el valor de un sensor únicamente leyendo una variable local incluso aunque el sensor esté en otra máquina. Ofrece un amplio repertorio de componentes *drivers*. En el caso de las cámaras, ofrece la misma interfaz de imagen para distintas fuentes de vídeo, lo cual proporciona una gran flexibilidad y portabilidad a las aplicaciones. Todos los componentes se comunican a través de interfaces explícitos ICE.

JdeRobot hace uso de ICE, un *middleware* de comunicaciones desarrollado por la empresa ZeroC[2] orientado a objetos distribuidos bajo una doble licencia (GNU GPL y una licencia propietaria). Se utiliza para aplicaciones distribuidas de Internet sin la necesidad de

[1] http://jderobot.org/
[2] http://www.zeroc.com/

utilizar los protocolos HTTP y es capaz de atravesar cortafuegos, a diferencia de la mayoría de *middleware* de este tipo. Es compatible con diferentes lenguajes de programación como C++, Java, Python y la mayoría de sistemas operativos como Windows, MAC OS X, Linux y Solaris. Una variante de ICE es ICE-E que puede ejecutarse dentro de teléfonos móviles. Utilizando ICE se han diseñado un conjunto de interfaces para que unos componentes interactúen con otros, siendo éste el único mecanismo de comunicación entre los componentes.

Las aplicaciones creadas para el presente trabajo están programadas sobre la pataforma JdeRobot 5.2.

3.3. OpenCV

OpenCV es una librería de visión artificial desarrollada inicialmente por Intel, mantenida actualmente por Willow Garage[3] e Itseez[4]. Es una librería de código abierto y está publicada con licencia BSD, lo que permite su libre utilización en propósitos comerciales o de investigación.

Figura 3.2: Filtro de Canny implementado en OpenCV

Esta librería proporciona un extenso conjunto de funciones para trabajar con visión artificial cuyo desarrollo se ha realizado primando la eficiencia. Se han programado

[3]http://www.willowgarage.com/
[4]http://itseez.com/

utilizando C y C++ optimizados, pudiendo además hacer uso del sistema de primitivas de rendimiento integrado de los procesadores Intel, que son un conjunto de rutinas de bajo nivel específicas en estos procesadores y que ofrecen un gran rendimiento. Contiene más de 500 funciones que abarcan una gran gama de áreas en el procesamiento de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estéreo, descriptores visuales o visión robótica.

Tiene interfaces para C, C++, Python y Java y soporta los siguientes sistemas operativos: Windows, Linux, Mac OS, iOS y *Android*. La documentación detallada de esta librería se puede encontrar en Internet y continuamente actualizada en forma de wiki[5].

La versión utilizada en este trabajo ha sido OpenCV 2.4.8. Se puede destacar el uso de esta librería para la transformación del espacio de color de imágenes, la calibración de cámaras o la extracción de características (por ejemplo FAST).

3.4. OpenGL

OpenGL[6] es una especificación estándar que define una API multiplataforma para desarrollar aplicaciones con gráficos 2D y 3D. A partir de primitivas geométricas simples, como puntos o rectas, permite generar escenas tridimensionales complejas. Actualmente es utilizado en realidad virtual, desarrollo de videojuegos (Figura 3.3) y multitud de representaciones científicas.

Partiendo de la especificación de OpenGL, los fabricantes de hardware crean implementaciones, que son bibliotecas de funciones que se ajustan a los requisitos de la especificación, utilizando aceleración hardware cuando es posible. Dichas implementaciones deben superar unos tests de conformidad para que sus fabricantes puedan calificar su implementación como conforme a OpenGL y para poder usar el logotipo oficial. Hay varias implementaciones para múltiples plataformas Hardware y Software como Linux, Windows, MacOS.

OpenGL tiene dos propósitos esenciales:

- Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, presentando al programador una API única y uniforme.
- Ocultar las diferentes capacidades de las diversas plataformas hardware.

[5] `http://opencv.willowgarage.com/wiki/`
[6] http://www.opengl.org/

Figura 3.3: Captura de pantalla de videojuego desarrollado en OpenGL

En este proyecto se utiliza OpenGL para visualizar la escena compuesta por los puntos del mapa y la cámara, mostrando la posición relativa de uno respecto al otro. Esto permitirá depurar los valores obtenidos por los algoritmos de autolocalización. Al utilizar OpenGL se aprovecha la GPU para realizar las operaciones anteriores descargando a la CPU de este trabajo. Por lo tanto, la visualización no ralentiza significativamente la velocidad de ejecución de la aplicación.

3.5. Bibioteca OGRE 3D

OGRE 3D[7], acrónimo del inglés *Object-Oriented Graphics Rendering Engine*, es un motor de *renderizado* 3D orientado a escenas, escrito en el lenguaje de programación C++. Sus bibliotecas evitan la dificultad de la utilización de capas inferiores de librerías gráficas como OpenGL y Direct3D, y además, proveen una interfaz basada en objetos del mundo y otras clases de alto nivel. El motor es software libre, con licencia MIT y con una comunidad muy activa.

OGRE 3D ofrece soporte en los siguientes elementos de un motor gráfico: renderizado, gestión de la escena, texturizado, iluminación, sombras, *shaders*, animación, mallas, curvas,

[7]http://www.ogre3d.org/

Figura 3.4: Captura del test *Fresnel Reflections and Refractions*

superficies, efectos especiales, *scripting* y físicas. Ogre contiene un pequeño módulo de físicas que puede ser extendido con bibliotecas específicas de físicas como ODE[8] o Bullet[9].

Esta herramienta ha ido útil en el desarrollo del experimento de validación del algoritmo de autolocalización, permitiendo el desarrollo de un pequeño videojuego.

3.6. Biblioteca Eigen

Eigen[10] es una librería de algebra lineal escrita en el lenguaje de programación C++. Es una librería de código abierto licenciada bajo MPL2 desde la versión 3.1.1, las versiones anteriores están licenciadas bajo LGPL3+. Esta librería permite trabajar con matrices, vectores y ofrece métodos de resolución y reducción de sistemas lineales entre otros.

Eigen ha sido utilizada con frecuencia en este proyecto para realizar cálculos matriciales que son frecuentes en la geometría proyectiva. Se han empleado las clases ofrecidas para la resolución de sistemas de ecuaciones compatibles determinados y sistemas de ecuaciones lineales sobredimensionados. Se ha utilizado la versión Eigen 3.1.2

[8] http://www.ode.org/
[9] http://bulletphysics.org/wordpress/
[10] http://eigen.tuxfamily.org/index.php?title=Main_Page

3.7. Biblioteca Qt

Qt[11] es una biblioteca multiplataforma usada para desarrollar interfaces gráficas de usuario, así como para el desarrollo de programas sin interfaz gráfica, como herramientas para la línea de comandos y consolas para servidores.

Qt es software libre a través de Qt project, donde participa tanto la comunidad libre, como desarrolladores de Nokia, Digia y otras empresas. Qt es distribuida bajo los términos de la GNU Lesser General Public License. Qt es utilizada en KDE, entorno de escritorio para sistemas como GNU/Linux o FreeBSD, entre otros. Qt utiliza el lenguaje de programación C++ de forma nativa, adicionalmente puede ser utilizado en otros lenguajes de programación a través de *bindings*.

Funciona en todas las principales plataformas, y tiene un amplio apoyo. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos, además de estructuras de datos tradicionales. Qt es utilizada principalmente en Google Earth, Skype, VLC media player, VirtualBox entre otros.

Qt se utiliza en este proyecto para crear la interfaz gráfica. Cabe destacar que Qt ofrece una clase especial que contiene un *canvas* para OpenGL. La versión utilizada ha sido Qt 4.8.0

3.8. Android

Android es un sistema operativo orientado a dispositivos móviles basado en un versión del núcleo de Linux. Inicialmente fue desarrollada por *Android Inc.*, compañia que fue comprada por Google, y en la actualidad lo desarrollan los miembros de la O*pen Hansdet Alliance* (liderada por Google).

En noviembre de 2013 Google afirmó que se activaban 1.500.000 dispositivos diariamente, alcanzando la cifra del 92 % de nuevos *smartphones* para el trimestre comprendido entre diciembre 2012 y febrero 2013 en España. Este dato convierte a *Android* en una buena plataforma para desarrollar (gracias a que es software libre) y para distribuir. Además *Android* ofrece un conjunto de herramientas a disposición del desarrollador que hacen más sencilla la programación en dispositivos móviles.

Lo componentes principales del sistema operativo de *Android* son:

[11] http://qt-project.org/

- **Aplicaciones**: Las aplicaciones base incluyen un cliente de correo electrónico, programa de SMS, calendario, mapas, navegador, contactos y otros. Todas las aplicaciones están escritas en el lenguaje de programación *Java*.
- ***Framework* de aplicaciones**: Los desarrolladores tienen acceso completo a las mismas APIs del *framework* usadas por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes: cualquier aplicación puede publicar sus capacidades y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del *framework*). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.
- **Bibliotecas**: *Android* incluye un conjunto de bibliotecas $C/C++$ usadas por varios componentes del sistema. Estas características se exponen a los desarrolladores a través del *framework* de aplicaciones. Algunas son: System C library (implementación de la libreria extandar de C), bibliotecas de medios, bibliotecas de gráficos, SQLite, entre otras. *Android* incluye un conjunto de bibliotecas base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java.
- ***Runtime* de *Android***: Cada aplicación *Android* corre su propio proceso, con su propia instancia de la máquina virtual de Java Dalvik. Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecuta archivos en el formato Dalvik Executable (.dex), el cual está optimizado para consumo de memoria mínimo. La máquina virtual está basada en registros, y corre clases compiladas por el compilador de Java que han sido transformadas al formato .dex por la herramienta incluida "dx".
- **Núcleo Linux**: *Android* depende de Linux para los servicios base del sistema como seguridad, gestión de memoria, gestión de procesos, *stack* de red, y modelo de controladores. El núcleo también actúa como una capa de abstracción entre el hardware y el resto del *stack* de software.

Se ha utilizado *Android* para validar los algoritmos de autolocalización visual. El dispositivo mostrará las imágenes de la cámara y se hará uso de OpenGL para representar objetos en 3D en la pantalla de la tableta con *Android*.

3.8.1. Fundamentos de una aplicación *Android*

La plataforma *Android* proporciona diferentes componentes a la hora de programar en función del objetivo de tu aplicación. *Android* provee cuatro tipos diferentes de

componentes:

- ***Activity***: Una actividad es el componente más usado en las aplicaciones *Android*. Típicamente una actividad representa una pantalla individual en el terminal y presenta una interfaz gráfica al usuario. Por ejemplo, en una aplicación de listado de teléfonos utilizaríamos dos actividades. Una para mostrar el listado de nombres y teléfonos y la segunda para mostrar la información detallada del contacto seleccionado. La navegación entre las pantallas se realiza iniciando nuevas actividades. Cuando una actividad es abierta, la actividad previa es puesta en pausa y agregada el *history stack* y no volverá al estado de ejecución hasta que vuelva a ser invocada.

- ***Services***: Un servicio no tiene interfaz gráfica, pero puede ejecutarse en segundo plano por un tiempo indefinido (se asemeja mucho al demonio de los sistemas Linux). Por ejemplo, podemos utilizar un servicio para que vaya capturando cada cierto tiempo la posición GPS y nos avise cuando estemos cerca de algún amigo. Mientras tanto el usuario puede seguir realizando otras tareas.

- ***Broadcast receivers***: Este tipo de componentes se utilizan para recibir y reaccionar ante ciertas notificaciones *broadcast*. Este tipo de componentes no tienen interfaz gráfica y pueden reaccionar ante eventos como cambio de zona horarias, llamadas o nivel de batería. Todos los *receivers* heredan de la clase base *BroadcastReceiver*.

- ***Intent***: Este tipo de componentes es una clase especial que usa *Android* para moverse de una pantalla a otra. Un *Intent* describe lo que una aplicación desea hacer. Cualquier *activity* puede reutilizar funcionalidades de otros componentes con solo hacer una solicitud en la forma de *Intent*.

3.8.2. Java Native Interface

Java Native Interface (JNI) es un entorno de programación que permite que un programa escrito en Java ejecutado en la máquina virtual Java (JVM) pueda interactuar con programas escritos en otros lenguajes como C, C++ y ensamblador.

El JNI se usa para escribir métodos nativos que permitan solventar situaciones en las que una aplicación no puede ser enteramente escrita en Java. Muchas de las clases de la API estándar de Java dependen del JNI para proporcionar funcionalidad al desarrollador y al usuario, por ejemplo las funcionalidades de sonido o lectura/escritura de ficheros.

Dado que puede ser usado para interactuar con código escrito en otros lenguajes como C++, también se usa para operaciones y cálculos de alta complejidad temporal. El código nativo es por lo general más rápido que el que se ejecuta en una máquina virtual.

En este proyecto JNI ha sido útil para compilar el código C++ y ejecutarlo en un dispositivo *Android*. La versión utilizada ha sido NKD 9.0b.

3.8.3. Librería OpenGL ES

OpenGL ES[12] (*OpenGL for Embedded Systems*) es una variante simplificada de la API gráfica OpenGL diseñada para dispositivos integrados tales como teléfonos móviles, PDAs y consolas de videojuegos. La define y promueve el *Grupo Khronos*, un consorcio de empresas dedicadas a hardware y software gráfico interesadas en APIs gráficas y multimedia.

Esta librería es utilizada por la mayoría de los dispositivos más populares de hoy en día:

- OpenGL ES ha sido seleccionada como la API para gráficos 3D oficial en el sistema operativo Symbian OS y la plataforma para dispositivos móviles *Android*.
- OpenGL ES 2.0 es la librería gráfica 3D para el dispositivo Nokia N900 con sistema operativo *Maemo* basado en Linux.
- OpenGL ES es la librería gráfica 3D en el SDK del iPhone.
- OpenGL ES 1.0 más algunas extensiones y con soporte de *Cg* está disponible para la *PlayStation* 3 como API gráfica oficial.

3.9. Eclipse IDE

El entorno de desarrollo integrado (IDE) *Eclipse*[13] es un entorno de desarrollo de código abierto, multiplataforma, para desarrollar "Aplicaciones de Cliente Enriquecido" La definición que da el proyecto *Eclipse* acerca de su software es: "una herramienta universal - un IDE abierto y extensible para todo y nada en particular".

Eclipse emplea módulos (en inglés *plug-ins*) para proporcionar toda su funcionalidad al frente de la plataforma de cliente rico, a diferencia de otros entornos monolíticos donde

[12] http://www.khronos.org/opengles/
[13] http://www.eclipse.org/

las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulos lo hace una plataforma ligera para componentes de software. Además de permitir a *Eclipse* extenderse usando otros lenguajes de programación como C/C++ y Python, le permite trabajar con lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet y Sistema de gestión de base de datos. La arquitectura de *plugins* permite escribir cualquier extensión deseada en el ambiente, como sería gestión de la configuración. Se provee soporte para Java y CVS en el SDK de *Eclipse*.

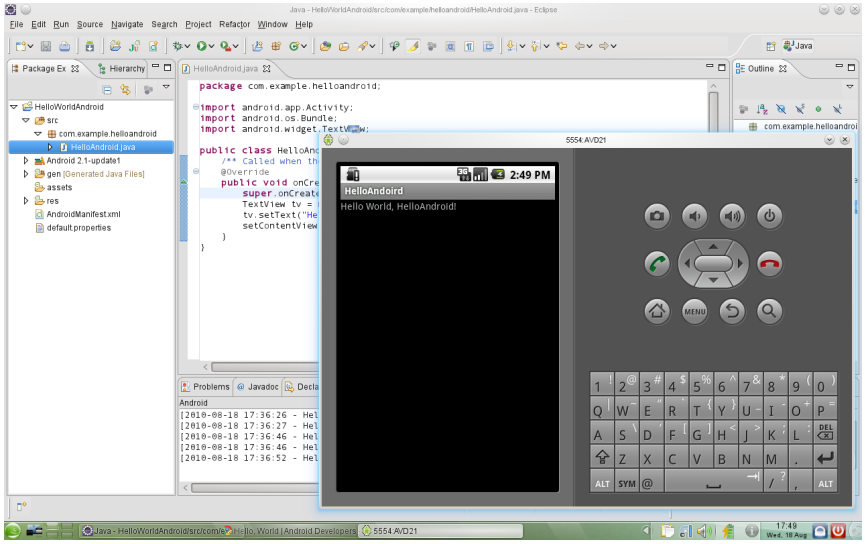


Figura 3.5: *Android* SDK en *Eclipse*

Android ofrece una extensión muy completa para el entorno de desarrollo de *Eclipse* que facilita el acceso al *API*, al emulador y al depurador. En la Figura 3.5 se puede observar el entorno de desarrollo de *Eclipse* con la extensión para *Android* y el emulador arrancado.

El emulador de *Android* es una herramienta muy potente, ya que es muy fiable y tiene un alto grado de robustez. Aunque es algo lenta, como la mayoría de los emuladores, tiene dos grandes puntos a su favor. El primero es que puede emular casi todas las características reales del móvil: fallos en la red, cámara de fotos, SMS, llamadas o posicionamiento *GPS*. El segundo es su fiabilidad, aplicación que se prueba y funciona correctamente en el emulador, lo hace igual de bien en el dispositivo móvil real.

Capítulo 4

Fundamentos Teóricos

En este capítulo se describen algunos de los procedimientos y técnicas matemáticas más importantes que se han utilizado en el Trabajo Fin de Máster. En concreto algunos conceptos de geometría proyectiva, el filtro de Kalman y el filtro de Kalman extendido. También se presentan los distintos algoritmos de autolocalización utilizados como monoSLAM, PTAM y DLT. El objetivo de estas técnicas es la extracción de información espacial y geométrica desde la información visual de la imagen.

4.1. Modelo de cámara PinHole

El problema a resolver implica proyecciones de un espacio tridimensional (el mundo externo a la cámara) en un espacio bidimensional (las imágenes son planas). Es fundamental conocer el modelo geométrico de cámara y la manera en que un punto 3D se convierte en un punto 2D y esto conlleva modelarlo matemáticamente.

El modelo de cámara *PinHole* recoge la idea de una proyección cónica, es decir, asume que todos los rayos de luz pasan por el mismo punto, el foco de la cámara. Se basa en el modelo de cámara oscura, en el que los rayos de luz entran en una caja por un agujero minúsculo e impactan en el lado contrario, formando una imagen del objeto que caja tiene enfrente.

Las cámaras actuales permiten el uso del modelo *Pinhole* ya que el modelo cuadra pese al uso de lentes. En la Figura 4.1 se muestra de manera simplificada el plano imagen frente al foco de la cámara y no detrás, como lo estaría en un cámara real. Este modelo es útil por su sencillez y tiene buena precisión a la hora de modelar las cámaras utilizadas en este proyecto.

4.1.1. Parámetros intrísecos

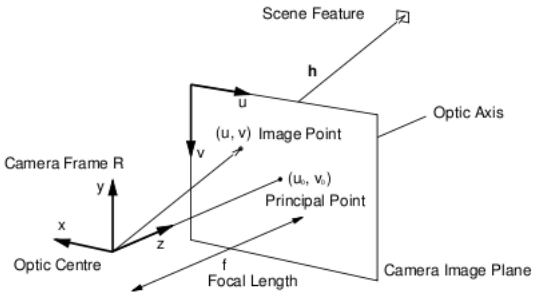


Figura 4.1: Modelo pinhole de cámara

Una cámara ideal con el foco situado en el origen de coordenadas, apuntado en la dirección positiva de eje Z, proyecta los puntos 3D en el plano imagen según la siguiente ecuación, donde f es la distancia focal, es decir, la distancia del foco al plano imagen.

$$\begin{pmatrix} u \\ v \end{pmatrix} = f \cdot \begin{pmatrix} -\frac{x}{z} \\ -\frac{y}{z} \end{pmatrix} \tag{4.1}$$

El origen de coordenadas de las imágenes almacenadas en un sistema informático suele encontrarse en la esquina superior izquierda de la imagen, por lo que, si la imagen tiene dimensiones m x n la ecuación queda:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{m}{2} \\ \frac{n}{2} \end{pmatrix} + f \cdot \begin{pmatrix} -\frac{x}{z} \\ -\frac{y}{z} \end{pmatrix} \tag{4.2}$$

Aun suponiendo que la lente fuera ideal, si ésta no está perfectamente alineada con el plano de proyección esto da lugar a que el centro óptico no se encuentre en $\begin{pmatrix} \frac{m}{2} \\ \frac{n}{2} \end{pmatrix}$ sino en un punto genérico $\begin{pmatrix} u_0 \\ v_0 \end{pmatrix}$ llamado punto principal. Además, la imagen podría estar ligeramente achatada, lo que se modelo con una distacia focal distinta en el eje X que en el eje Y. El modelo básico queda:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} u_0 \\ v_0 \end{pmatrix} + \begin{pmatrix} f_x & 0 \\ 0 & f_y \end{pmatrix} \begin{pmatrix} -\frac{x}{z} \\ -\frac{y}{z} \end{pmatrix} \tag{4.3}$$

Utilizando coordenadas homogéneas se puede expresar 4.3 como 4.4:

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w^{cam} = K \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w^{cam} = KP_w^{cam} \tag{4.4}$$

El término λ aparece porque se están utilizando clases de equivalencia y coordenadas homogéneas, es un factor de escala.

La matriz de proyección K con la focal y las coordenadas del centro óptico contiene los parámetros intrínsecos de la cámara. En este proyecto necesitaremos cámaras calibradas, es decir, de parámetros intrínsecos conocidos. No ha sido necesario introducir la distorsión radial ni el *skew*, ya que las cámaras utilizadas no introducen grandes deformaciones en estos parámetros.

La correspondencia de un punto 3D, en coordenadas de la cámara, a otro punto 2D en la imagen no es única. Dado un punto $\begin{pmatrix} u \\ v \end{pmatrix}$ todos los puntos que pertenecen a la recta que une el centro de proyección con el punto 3D y el punto de la imagen impactan en el mismo punto en la imagen.

4.1.2. Parámetros extrínsecos

En el caso de usar cámaras móviles, no se puede asumir que el foco está en el origen de coordenadas ni que la cámara mira en la dirección positiva del eje Z. El punto P_w^{cam} está expresado en el sistema de coordenadas de la cámara. En nuestro sistema las coordenadas vienen expresadas respecto a otro sistema de referencia absoluto que no tiene porqué ser el de la cámara. Dado que partimos de un punto P_w^{abs} expresado en un sistema de referencia en el universo, para hallar el píxel P_{im} correspondiente a este punto lo primero es expresar el punto en el mismo sistema de referencia de la cámara. Sólo entonces se pueden aplicar las ecuaciones anteriores. De modo genérico, para pasar del sistema de referencia absoluto al sistema de referencia de la cámara, se tiene que aplicar una rotación y un traslación genéricas (pudiendo ser alguna de ellas nula). Las matrices correspondientes a este cambio de base se denominan matriz de rotación y translación extrínseca, RT_{ext}.

Este cambio de coordenadas se puede expresar de forma matricial en coordenadas homogéneas con la siguiente formulación:

$$P_w^{cam} = R \cdot T \cdot P_w^{abs} \tag{4.5}$$

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{cam} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w \tag{4.6}$$

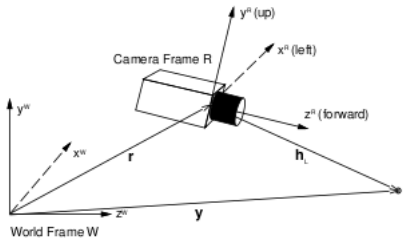


Figura 4.2: Modelo extrínsecos y relación entre sistemas de referencia

Combinando los parámetros extrínsecos e intrínsecos obtenemos la ecuación general para proyectar cualquier punto 3D del universo sobre el plano imagen. En una cámara real esto no siempre es posible, el sensor de la cámara tiene unas dimensiones limitadas y ciertos puntos del plano imagen se salen de su campo de visión por lo que proyectan fuera del tamaño del fotograma.

$$P_{im} = K \cdot R \cdot T \cdot P_w^{abs} \tag{4.7}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_{im} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_w \tag{4.8}$$

El objetivo de las técnicas de SLAM Visual consiste en estimar en tiempo real los parámetros extrínsecos de un cámara móvil, es decir, los coeficientes de la matriz RT.

4.2. DLT

DLT (*Direct Linear Transformation*), es un algoritmo de autocalibración de cámaras asumiendo como la realidad conocida un patrón 3D con múltiples marcadores. El problema de la calibración consiste en calcular los parámetros intrínsecos (distancia focal y punto central) y extrínsecos (posición y orientación) de una cámara. En general los parámetros extrínsecos se representan con dos matrices genéricas de rotación y translación como los explicados en la sección 4.1. Este algoritmo se ha aplicado como un algoritmo de autolocalización 3D. Por lo tanto, el objetivo es calcular los parámetros extrínsecos de la cámara, dentro de los cuales está la posición y orientación de la cámara.

4.2.1. Matriz genérica de proyección

Esta técnica estudia el paso de 3D a 2D de una cámara partiendo de un patrón de calibración, del cual se conoce con antelación la posición de ciertos puntos (por ejemplo respecto a un marco de referencia ligado al mismo objeto) y en qué píxeles de la imagen proyectan.

La ecuación general para proyectar cualquier punto 3D del universo sobre el plano imagen se explicó en la sección 4.1 (ecuación 4.8). Si se reescribe la ecuación para dejar sólo una matriz incógnita M. La matriz genérica de proyección, tiene el siguiente aspecto:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_{im} = KRT = M \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{w} = \begin{bmatrix} h_{11} & h_{12} & h_{13} & h_{41} \\ h_{21} & h_{22} & h_{23} & h_{42} \\ h_{31} & h_{32} & h_{33} & h_{43} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}_{w} \tag{4.9}$$

El primer paso es calcular las once incógnitas de la matriz M, teniendo en cuenta que el parámetro h_{34} se puede fijar a un valor constante. Cada punto 3D proporciona dos ecuaciones 4.10 y 4.11, por lo tanto serán necesarios al menos seis emparejamientos entre puntos 3D y puntos 2D para poder calcular la matriz M.

$$u = \frac{h_{11}x + h_{12}y + h_{13}z + h_{14}}{h_{31}x + h_{32}y + h_{33}z + 1} \tag{4.10}$$

$$v = \frac{h_{21}x + h_{22}y + h_{23}z + h_{24}}{h_{31}x + h_{32}y + h_{33}z + 1} \tag{4.11}$$

Con seis puntos se puede formular un sistema de doce ecuaciones para resolver las doce incógnitas. Se trata de un sistema lineal sobredimensionado ya que se dispone de un

mayor número de ecuaciones que de incógnitas por resolver. La solución en este caso no es exacta ya que cada subcojunto de once ecuaciones posibles dará una solución distinta pero parecida a la que daría otro subconjunto. Por lo tanto, el objetivo es hallar la solución que menos error comete a la hora de pasar de 3D a 2D, esto se conoce como optimización (en lugar de resolución) de un sistema sobredimensionado. En la Figura 4.3 se pueden observar las entradas y salidas del algoritmo DLT. Se tiene como entrada al menos 6 puntos y se obtiene la posición y orientación de la cámara junto con sus parámetros de calibración.

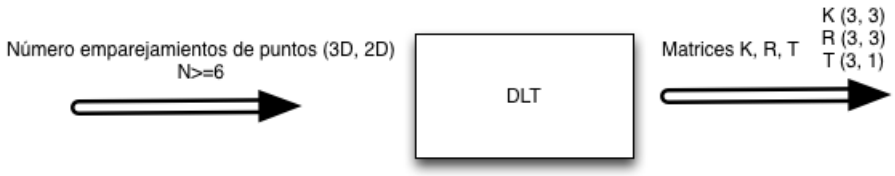


Figura 4.3: *Pipeline* del algoritmo DLT

Una vez construido el sistema de ecuaciones sobredimensionado, se ha resuelto utilizando la clase ofrecida por *Eigen* que resuelve el problema utilizando la *SVD* (*Singular Value Descomposition*) para optimizar. El resultado son las once incógnitas que forman la matriz M (4.9). Esta solución permite utilizar tantas ecuaciones como puntos conocidos en la imagen se detecten, siendo siempre mayor que seis puntos, con lo que la solución obtenida será más precisa y robusta a medida que introducimos más emparejamientos.

4.2.2. Descomposición de la matriz de proyección

Una vez conocida la matriz de proyección es necesario descomponer esta matriz para calcular explicitamente los parámetros extrínsecos e intrínsecos de la cámara. En los valores de la matriz M están escondidos los valores K, R, T que se quieren extraer por separado para conocer la posición y orientación exactos de la cámara. Existen varias formas de descomponer la matriz de proyección M, como por ejemplo, la descomposición RQ, una variante de QR. La descomposición QR no es única. Para forzar la unicidad es necesario restringir el signo positivo de la distancia focal. A continuación se detalla el procedimiento seguido para descomponer la matriz de proyección M. Con el método explicado en la sección anterior se ha obtenido la matriz M de proyección salvo un factor de escala. Se denomina $\widehat{M}$ a la matriz estimada.

$$\widehat{M} = |\gamma| M \tag{4.12}$$

Se definen los vectores:

$$q_1 = \left[\widehat{m_{11}}\widehat{m_{12}}\widehat{m_{13}}\right] \tag{4.13}$$

$$q_2 = \left[\widehat{m_{21}}\widehat{m_{22}}\widehat{m_{23}}\right] \tag{4.14}$$

$$q_3 = \left[\widehat{m_{31}}\widehat{m_{32}}\widehat{m_{33}}\right] \tag{4.15}$$

$$q_4 = \left[\widehat{m_{14}}\widehat{m_{24}}\widehat{m_{34}}\right]^T \tag{4.16}$$

Por lo tanto $\widehat{M}$:

$$\widehat{M} = \begin{bmatrix} q_1 & q_{41} \\ q_2 & q_{42} \\ q_3 & q_{43} \end{bmatrix} \tag{4.17}$$

Después se calcula el valor absoluto del factor de escala $|\gamma|$ usando el vector unitario R_3^T, q_3 es la última fila de la matriz de rotación R. Por la ortogonalidad de R:

$$\sqrt{\widehat{m_{31}^2} + \widehat{m_{32}^2} + \widehat{m_{33}^2}} = |\gamma|\sqrt{r_{31}^2 + r_{32}^2 + r_{33}^2} \tag{4.18}$$

La raíz da 1 ya que las filas de R son vectores unitarios. Si dividimos las componentes de $\widehat{M}$ por $|\gamma|$, se obtiene la matriz de proyección normalizada que se seguirá llamando $\widehat{M}$, que difiere de M en un cambio de signo σ. El signo de γ se calcula a partir de m_{34} (i.e. q_{43})

$$T_z = \sigma\widehat{m_{34}} \tag{4.19}$$

El signo se deduce por el origen del mundo, si está delante de la cámara ($T_z > 0$) o detrás ($T_z < 0$).

La obtención de R_3:

$$r_{3i} = \sigma\widehat{m}_{3i}, i = 1, 2, 3 \tag{4.20}$$

Los centros ópticos (u_o y v_0) se calculan con los productos escalares de las filas $q_1^T q_3$ y $q_2^T q_3$ (conocidos ya que se conoce M) que producen los siguiente resultados usando las restricciones de ortogonalidad de R:

$$q_1^T q_3 = u_0 \tag{4.21}$$

$$q_2^T q_3 = v_0 \tag{4.22}$$

Para determinar f_x y f_y se hace a partir de q_1 y q_2 respectivamente:

$$f_x = sqrt(q_1^T q_1 - u_0^2) \tag{4.23}$$

$$f_x = sqrt(q_2^T q_2 - v_0^2) \tag{4.24}$$

Por último, para calcular el resto de parámetros extrínsecos se siguen estas ecuaciones:

$$r_{1i} = \sigma(u_0 \widehat{m}_{3i} - \widehat{m}_{1i})/f_x, i = 1, 2, 3 \tag{4.25}$$

$$r_{2i} = \sigma(v_0 \widehat{m}_{3i} - \widehat{m}_{2i})/f_y, i = 1, 2, 3 \tag{4.26}$$

$$T_x = \sigma(u_0 T_z - \widehat{m}_{14})/f_x \tag{4.27}$$

$$T_y = \sigma(v_0 T_z - \widehat{m}_{24})/f_y \tag{4.28}$$

Como resultado final se obtiene la posición de la cámara T_x (4.27), T_y (4.28) y T_z (4.19) y la orientación expresada como una matriz R (ecuaciones 4.25, 4.26 y 4.20).

4.3. MonoSLAM

Los algoritmos conocidos como SLAM (*Simultaneous Localization And Mapping*) han sido utilizados históricamente para generar mapas en entornos desconocidos utilizando robots o vehículos autónomos y localizando a éstos a su vez dentro de el mapa generado. En ocasiones también se utiliza información a priori, por ejemplo un mapa inicial del entorno en el que se encuentra el robot. Típicamente se han utilizado estos algoritmos con sensores láser [Hahnel *et al.*, 2003], sonar o cámaras estéreo.

El término *MonoSLAM* (Monocular SLAM) se refiere a la utilización de una sola cámara para la localización y creación de mapas en entornos desconocidos. Fue propuesto

y desarrollado por primera vez por Andrew Davison a partir del año 2002 [Davison, 2002] [Davison, 2003].

El algoritmo propuesto por Andrew Davison utilizaba un filtro extendido de Kalman (en el Anexo 6.2 se puede encontrar los fundamentos matemáticos de los filtros de Kalman y los filtros de Kalman extendidos) para estimar la posición y orientación de la cámara, así como la posición de una serie de puntos en el espacio 3D. Originalmente para determinar la posición inicial de la cámara, es necesario dotar al filtro de Kalman de información a priori con la posición en 3D de al menos 3 puntos. A partir de ese momento el algoritmo es capaz de situar la cámara en 3D y de generar nuevos puntos para crear el mapa y servir como apoyo a la propia localización de la cámara.

4.3.1. EKF MonoSLAM

A continuación se muestran las principales características del EKF utilizado por el algoritmo de autolocalización visual monoSLAM. El vector de estado está compuesto por el estado propio de la cámara (posición, orientación y velocidades) y por el estado de cada uno de los puntos 3D existentes en un momento determinado en el mundo (su posición). Esto significa que tanto el vector de estado como su covarianza asociada cambiarán su tamaño de forma dinámica dependiendo del número de puntos.

$$\hat{x} = \begin{bmatrix} x_v \\ f_1 \\ f_2 \\ \vdots \\ f_N \end{bmatrix} \tag{4.29}$$

El estado de la cámara se compone de la posición 3D de la cámara, su velocidad lineal, la rotación de la cámara (representada con un cuaternión) y la velocidad angular. Por su parte, cada punto se representa con su posición 3D.

$$x_v = \begin{bmatrix} r^W \\ v^W \\ q^{WR} \\ \omega^R \\ p_{f_1}^W \\ ... \\ p_{f_N}^W \end{bmatrix} \tag{4.30}$$

$$x_{v|k} = f(x_{v|k-1}) = \begin{bmatrix} r_{|k-1}^W + v_{|k-1}^W \Delta k \\ v_{|k-1}^W \\ q_g(\omega_{|k-1}^R \Delta k) \times q_{|k-1}^{WR} \\ \omega_{|k-1}^R \\ p_{f_1|k-1}^W \\ ... \\ p_{f_N|k-1}^W \end{bmatrix} \tag{4.31}$$

El modelo de observación se compone de las proyecciones de cada uno de los puntos 3D en el plano imagen.

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = h \begin{bmatrix} x_v \\ f_i^W \end{bmatrix} = K\Big(R\Big(T \begin{bmatrix} x_v \\ f_i^W \end{bmatrix}\Big)\Big) \tag{4.32}$$

Como se explica en el anexo 6.4 el sistema tiene que ser lineal o linealizable en el entorno de trabajo para obtener una mejor estimación con el filtro. En el caso de monoSLAM se realizan muchas iteraciones por segundo, es decir, el sistema realiza iteraciones cada pocos milisegundos. En esta escala de tiempos todo es lineal o linealizable. Esta es una de las razones que da el propio autor para usar el EKF.

4.3.2. Seguimiento de múltiples puntos en 2D

En monoSLAM para el seguimiento de múltiples puntos de interes se usan distintos filtros extendidos de Kalman independientes. En cada iteración se aplica el detector de puntos de interés a la imagen de entrada, a continuación se realiza el proceso de asociación punto detectado-filtro, después se realiza el paso de predicción de los filtros extendidos de Kalman, y por último, se ejecutará el paso de actualización del filtro. De cada punto del mapa monoSLAM guarda su aparición visual (parche de píxeles vecinos monocromático) y su posición.

En nuestra implementación la dectección de puntos de interés en la imagen se realiza por medio de la función de OpenCV *cvGoodFeaturesToTrack*. El resultado de esta función es un conjunto desordenado de puntos (x, y) de la imagen que se considerarán candidatos a ser el vector de observación de los puntos que son objetivo del seguimiento.

Estos puntos de interés contienen una alta derivada espacial en la luminancia de la imagen. El algoritmo calcula para cada punto de la imagen la matriz de gradiente de un parche cuadrado centrado en dicho punto y calcula los autovalores de esta matriz. El parámetro que determina la calidad de una esquina es el mínimo de los autovalores.

```
1 void goodFeaturesToTrack(InputArray image, OutputArray corners,
2           int maxCorners, double qualityLevel, double minDistance,
3           InputArray mask=noArray(), int blockSize=3,
4           bool useHarrisDetector=false, double k=0.04 )
```

La función devuelve, como máximo, tantos puntos como indique el valor *maxCorner*, garantizando un calidad (*qualityLevel*) y una distancia mínima entre ellos (*minDistance*).

En entornos fuertemente estructurados el detector de puntos de interés tiene éxito en la mayoría de los casos. Aún así es posible detectar falsas esquinas (por ejemplo, en zonas con brillos) que se deberán descartar. El algoritmo puede devolver falsas esquinas que no son persistentes a lo largo del tiempo. Puede también darse el caso de un entorno hostil en el que no existan puntos de apoyo fiables en escenarios sin texturas.

Los puntos obtenidos en la fase de extracción de características no se pueden utilizar directamente como obsevaciones del filtro de Kalman, ya que no se sabe qué candidato corresponde a qué filtro. Es necesario un paso intermedio entre la extracción de características y la ejecución de los filtros de Kalman, que es el emparejamiento. El procedimiento de búsqueda del mejor candidato es el siguiente:

- Se establece un área de búsqueda (de forma elíptica) alrededor del punto del paso de predicción del filtro de Kalman.
- Para cada punto que esté dentro del área, se calcula la distancia de Mahalanobis[1] al centro de área de búsqueda según la matriz de covarianza P del filtro de Kalman.
- Se extrae un parche cuadrado de imagen en torno a dicho punto y se calcula la función de divergencia con el parche que se guardó en la última asociación al punto objetivo.

[1]Comparar entre distancias Mahalanobis equivale a comparar verosimilitudes sin tener que evaluar la función densidad de probabilidad de la densidad de predicción

- Se ignoran los candidatos cuya divergencia excede de un determinado umbral.
- Si no queda ningún candidato, se considera rechazado. No se hará el paso de correción del filtro EKF.
- Se calcula el coste asociado a cada candidado como:

$$coste = (1 - P) \cdot distancia^2 + P \cdot divergencia^2$$

- Se establece el vector de observación para el paso de corrección del EKF de monoSLAM el del candidato cuyo coste sea mínimo.
- Se extrae el nuevo parche de imagen en torno a la posición corregida y se guarda en memoria asociado a ese punto.

La función de divergencia utilizada por Andrew Davison en su tesis doctoral es la siguiente (ec 4.33), constituye una medida de las diferencias entre dos parches monocromáticos de igual tamaño.

$$Divergencia = \sum_{parche} \frac{\left[\frac{g_1 - \hat{g_1}}{\sigma_1} - \frac{g_0 - \hat{g_0}}{\sigma_0}\right]^2}{n_{pixel}} \tag{4.33}$$

donde g_0 y g_1 son valores de luminancia en posiciones homólogas de los dos parches; $\hat{g_0}$ y $\hat{g_1}$ son los valores medios; σ_0 y σ_1 las deviaciones típicas de la luminancia a lo largo de los parches.

Parches iguales producen una salida de valor 0, este valor aumenta cuanto mayor sea la diferencia entre los parches. Parches totalmente distintos producen salidas por encima de 1. Una ventaja de esta medida es que está normalizada respecto a cambios generales de intensidad, dado que estima mejor la correspondencia a lo largo de períodos largos de operación del sistema, en los que la luminosidad del escenario pueden cambiar.

Después, sólo queda por realizar los dos últimos pasos que se corresponden a la fase de predicción del filtro EKF y la fase de corrección.

Para la asociación entre los puntos y parches, una subregión de la imagen que se está procesando. Éste se trata como una imagen independiente. Supone una disminución drástica de la carga computacional puesto que el procesamiento no se realiza sobre toda la imagen, sino sólo sobre este pequeño rectángulo. Típicamente sólo se computa sobre unos pocos cuadrados de 15x15 dispersos a lo largo de la imagen.

4.3.3. MonoSLAM eliminación de espúreos

Uno de los puntos débiles de los algoritmos de localización basados en filtros de Kalman, como monoSLAM, es su baja tolerancia a los falsos positivos, es decir, a los puntos que no estén correctamente emparejados. Por la propia naturaleza del algoritmo, el filtro trata de minimizar el error entre los puntos detectados y los predichos, de forma que en caso de que un emparejamiento sea erróneo afectará directamente a la localización estimada. En el peor de los casos un solo emparejamiento erróneo puede llevar a una localización errónea del algoritmo, por lo que el porcentaje máximo de falsos positivos que puede soportar es de 0 %.

Para resolver esta fragilidad algoritmica se utiliza el algoritmo 1-Point RANSAC [Civera *et al.*, 2010] que permite detectar emparejamientos erróneos para no tenerlos en cuenta en la fase de actualización del filtro de Kalman. Este método utiliza a su vez el algoritmo RANSAC (*Random Sample Consensus*) [Fischler and Bolles, 1981], un método iterativo para estimar parámetros de un modelo con observaciones que contengan díscolos (*outliers*).

El algoritmo de 1-Point Ransac divide la fase de actualización del filtro de Kalman en dos partes.

- Utiliza RANSAC de forma iterativa seleccionando en cada iteración una única característica (punto de interés observado) y actualizando el estado del filtro (no la covarianza) con esta única observación. Con este nuevo estado comprueba cuántas características validan esta hipótesis, es decir, cuántas características están lo suficientemente cerca de la posición predicha para este nuevo estado. Se etiquetan como puntos de baja innovación a todos aquellos que validaron la mejor iteración y se actualiza el EKF completo (estado y covarianza) con estos puntos y sólo con ellos. El número de iteraciones cambia dinámicamente dependiendo del número de características que validaron la hipótesis.

- En el paso anterior se etiquetaron como *outliers* los puntos con una innovación muy alta, que pueden corresponder a un *outlier* real o a un *inlier* que se encuentre muy cerca de la cámara. Así, se predice de nuevo la posición de los *outliers* y se realiza una búsqueda en una región de probabilidad de 99 %. Si resulta un *inlier* se utiliza en el filtro EKF. En caso de clasificarse como *outlier* se descarta, es decir, se ignora para actualizar la estimación el filtro EKF.

Este algoritmo permite eliminar *outliers* mejorando la fiabilidad de algoritmo, y tiene

como contrapartida un mayor coste computacional, aunque no es demasiado alto y puede seguir funcionando en tiempo real.

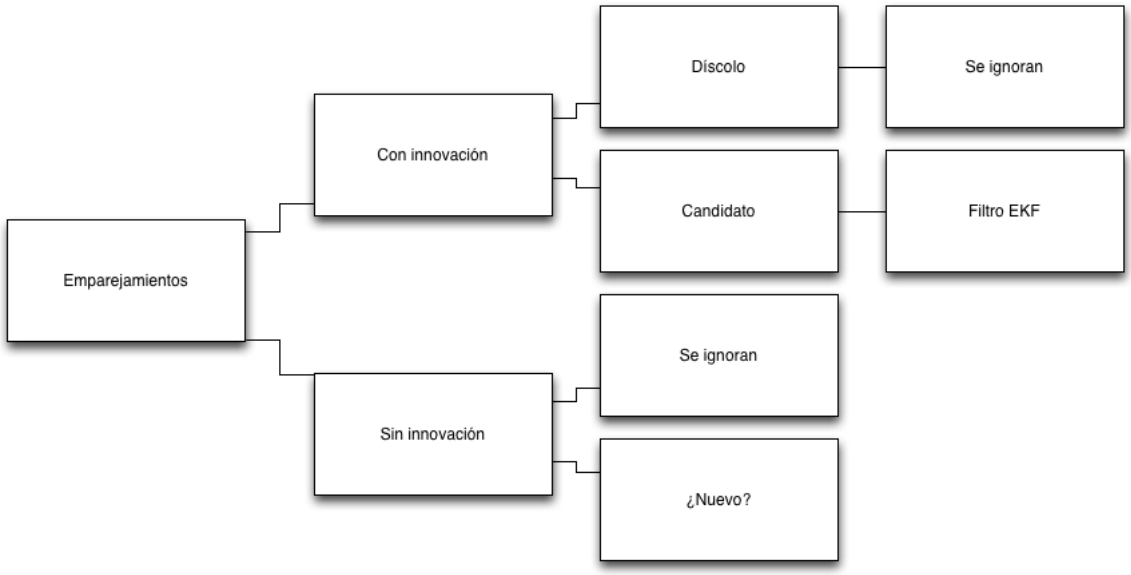


Figura 4.4: Diagrama de bloques eliminación de espúreos

En la Figura 4.4 se puede observar el diagrama de flujo que siguen los emparejamientos. Los emparejamientos con baja innovación se ignoran o puede ser un nuevo candidatos de baja calidad. Los puntos de alta innovación se clasifican como un díscolo, es decir, la eliminación de un expúreo o un posible candidato para el filtro EKF.

4.3.4. Parametrización inversa de la distancia

Otra de las limitaciones del algoritmo inicial de Andrew Davison es la necesidad del indicar al menos 3 puntos para inicializar el algoritmo. Este problema se resuelve con la parametrización inversa de la distancia propuesta por Javier Civera [Civera *et al.*, 2008]. Esta última técnica se basa en parametrizar la característica (el punto 3D observado en la imagen) de forma que pueda ser inicializada en un solo fotograma. Se describe una característica con 6 parámetros.

$$y_i = \Big\{ x_i, y_i, z_i, \theta, \Phi, \rho \Big\} \tag{4.34}$$

Los 3 primeros parámetros representan un punto tomado sobre la recta sobre la que se vió por primera vez el punto. Los dos siguientes parámetros representan las coordenadas

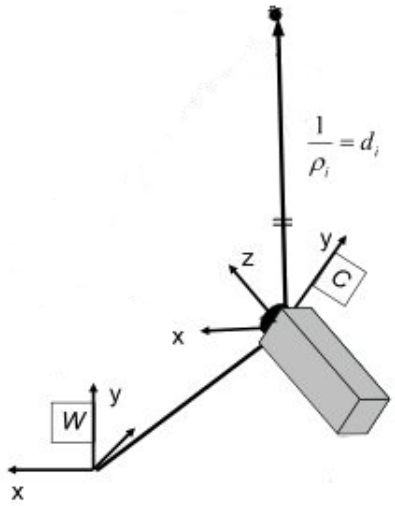


Figura 4.5: Parametrización inversa.

polares de la dirección de la recta. El sexto parámetro es la inversa de la distancia desde el punto representado por los tres primeros parámetros hasta el punto de interés.

La parametrización inversa de la distancia resuelve de una manera elegante el problema de la inicialización en el SLAM monocular. Los modelos propuestos anteriormente no eran capaces de introducirse en la estimación cuando eran visualizados por primera vez, debido a la imposibilidad de representar la profundidad a lo largo del rayo de proyección. El enfoque anterior consistía en retrasar esta inicialización en 3D hasta que la profundidad característica convergía a un valor. Estas características inicializadas de manera retrasada no contribuían a la estimación hasta que tenían suficiente paralaje (Figura 4.6). Es decir, aquellas características con paralaje cero, jamás se introducían en la estimación, y por lo tanto, su información se pierde.

La parametrización inversa no sólo tiene su interés en la inicialiación no retrasada. Estos puntos con cero paralaje son conocidos en el SfM como puntos en el infinito. Tiene un particular interés en exteriores. Una estrella, por ejemplo, se observa en el mismo lugar de una cámara que ha sido trasladada a través de muchos kilómetros apuntando al cielo sin rotar. Esta información no resulta útil para la estimación de la traslación de la cámara, pero es un referencia perfecta para la estimación de la rotación.

Las ventajas de esta parametrización inversa son enormes ya que aumentan la estabilidad del algoritmo y permiten una mayor facilidad de programación. La principal desventaja es el aumento del coste computacional.

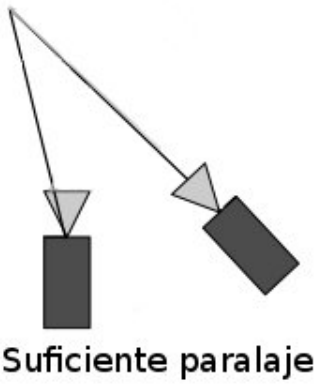


Figura 4.6: Paralaje, diferencia en distancia y orientación entre dos posiciones de la cámara, que permite triangular de modo robusto.

4.4. PTAM

PTAM son las siglas de *Parallel Tracking and Mapping*, un algoritmo desarrollado en 2007 por George Klein [Klein and Murray., 2007] que tenía como objetivo resolver el mismo problema que MonoSLAM, es decir, localización y generación de mapas simultáneas, pero desde un punto de vista totalmente diferente. El mayor problema de los algoritmos basados en MonoSLAM es que su tiempo de ejecución aumenta exponencialmente con el número de puntos. Esto es debido a que en cada iteración del algoritmo se actualiza tanto la posición de cada elemento del mapa (*mapping*) como la posición actual de la cámara (*tracking*) como se observa en la Figura 4.7. Esto hace que aunque el *Tracking* pueda mantenerse en tiempos de ejecución estables, el *Mapping* a partir de un cierto número de puntos impide que pueda ejecutarse el algoritmo en tiempo real.

PTAM no hace uso de técnicas de filtrado como monoSLAM, realiza un optimización mediante ajuste de haces, técnica que tiene su origen el SfM. Sí comparte con él la necesidad de detectar puntos de interés y emparejamiento con los anteriores.

PTAM parte de la idea de que sólo es necesario funcionar en tiempo real en la parte de *Tracking*, mientras que el *Mapping* no tiene porqué realizarse en cada iteración ni necesita ser tan eficiente. Así, el *Tracking* y el *Mapping* funcionan en los hilos separados de forma asíncrona, como se observa en la Figura 4.8.

El algoritmo de PTAM hace uso de *keyframes*, es decir, de fotogramas claves que se utilizan tanto para localizarse como para crear el mapa de puntos.

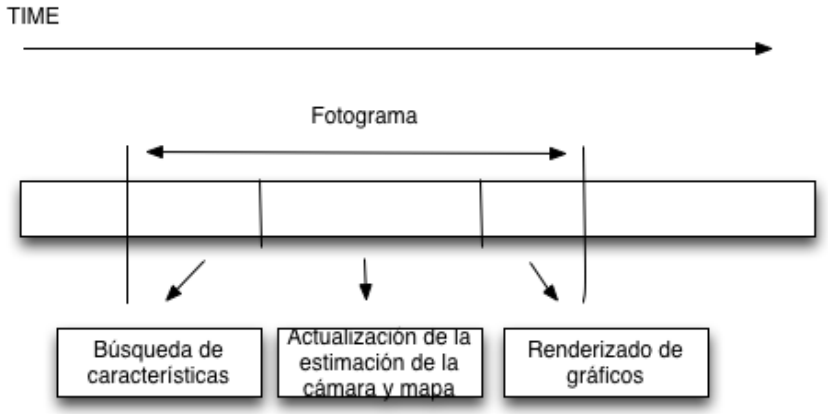


Figura 4.7: Tareas realizadas en una iteración de MonoSLAM.

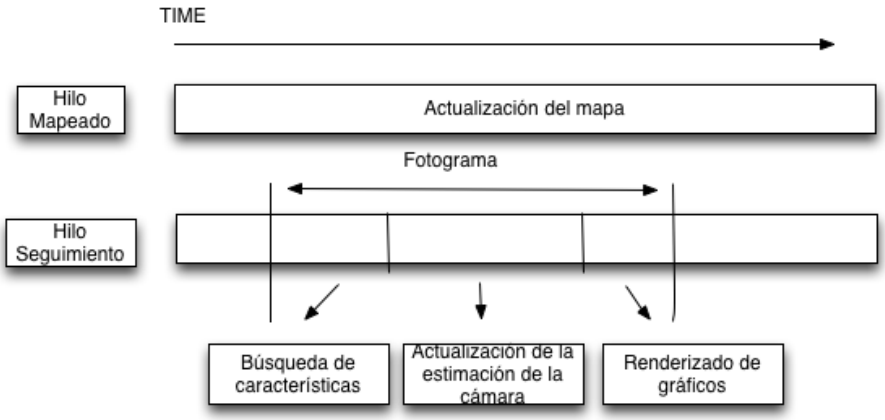


Figura 4.8: Tareas realizadas en una iteración de PTAM.

4.4.1. Seguimiento

En esta sección se detalla cómo se ha realizado el seguimiento de los puntos 2D en la imagen y la técnica de emparejamiento utilizada entre los parches de las imágenes.

En nuestra implementación como extrator de características se utiliza el detector FAST [Rosten *et al.*, 2010](*Features from Acelerated Segment Test*). Se inspeccionan los valores de intensidad de los píxeles en un círculo de radio r alrededor del píxel candidato p, como muestra la Figura 4.9. Un píxel del círculo es considerado *bright* si su intensidad es al menos t (valor umbral) unidades superiores de intensidad del píxel candidato p, y *dark* si su intensidad es la menos t unidades inferior. El píxel candidato p es considerado punto de interés si al menos un arco de longitud n píxeles *bright* o *dark* es encontrado en el círculo.

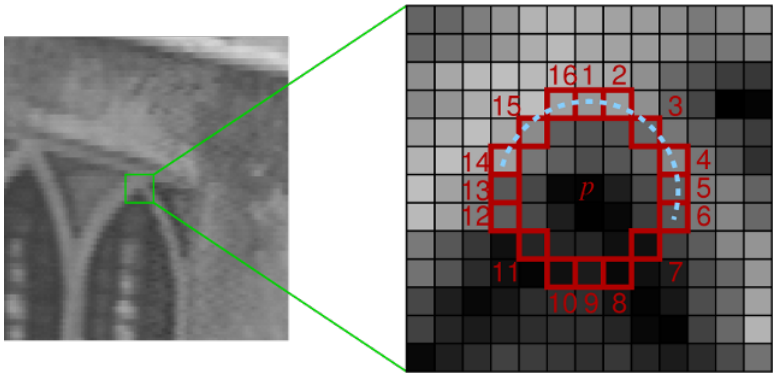


Figura 4.9: Ejemplo de detección FAST

Lo primero que se realiza es una subdivisión de la imagen a distintas resoluciones, lo que se conoce como pirámide de la imagen, para realizar un análisis multirresolución. A cada uno de estos niveles se le pasa un filtro de esquinas *FAST* para detectar los puntos más característicos de la imagen. Se utiliza un umbral diferente en cada nivel. El objetivo es equilibrar la densidad de características relativas a cada nivel.

Para buscar un punto del mapa en la imagen (obtenido o refrescado en el *keyframe* K) se realiza una búsqueda local alrededor de la proyección del punto 3D en la imagen. Para realizar esta búsqueda el parche tiene que ser deformado para tener en cuenta los cambios de vista entre la primera observación del parche y la posición actual de la cámara. Se realiza un transformación afín caracterizada por la matriz A para que la predicción sea más realista:

$$A = \begin{bmatrix} \frac{\partial u_c}{\partial u_s} & \frac{\partial u_c}{\partial v_s} \\ \frac{\partial v_c}{\partial u_s} & \frac{\partial v_c}{\partial v_s} \end{bmatrix} \tag{4.35}$$

donde u_s y v_s corresponden a los desplazamientos en píxeles horizontales y verticales en el primer nivel de la pirámide. Por otro lado, u_c y v_c corresponden al desplazamiento de píxeles en el fotograma actual en la imagen de la pirámide a tamaño completo. Esta matriz se encuentra retroproyectando el desplazamiento de cada píxel en el parche de la pirámide de *keyFrame* K y proyectando éste en el fotograma actual.

La relación entre las proyecciones asegura que la matriz compensa no sólo cambios de perspectiva y escala, sino también las variaciones en la distorsión de la lente a través de la imagen. El determinante de la matriz A se utiliza para decidir en qué nivel de la pirámide del fotograma actual el parche debe ser buscado. El determinante de A corresponde al área, el píxeles cuadrados. El $\frac{det(a)}{4}$ es el área correspondiente al siguiente nivel de la pirámide y así sucesivamente en cada nivel. El nivel de la piramide l es elegido como $\frac{det(a)}{4^l}$. Es decir, se intenta buscar el parche en el nivel de la pirámide que más se acerque a su escala.

Un parche de $8x8$ píxeles es generado por el nivel usando $A/2^l$ y una interpolación bilineal. La intensidad media de los píxeles en el parche se resta a cada valor de píxeles individuales para propocionar una cierta resistencia a cambios de iluminación. A continuación, la mejor combinación para este parche dentro de un radio determinado en torno a la posición predicha se localiza en la pirámide objetivo. Esto se hace evaluando las puntuaciones de *Zero mean SSD* de todos los puntos FAST obtenidos dentro de una región de búsqueda circular y seleccionando la localización con mejor diferencia. Si está por debajo de un umbral preestablecido, el parche se considera encontrado. En algunas cosas, especialmente en altos niveles de la pirámide, una ubicación, un solo píxel, no tiene suficiente precisión para obtener buenas resultados en el seguimiento. La localización del parche puede ser mejorada realizando una minimización iterativa del error.

El mapa que se usa para el seguimiento contiene inicialmente sólo dos *keyframes* y describe un reducido volumen del espacio. A medida que la cámara se mueve por el espacio 3D y se aleja de su postura inicial, se añaden muevos *keframes* y puntos característicos al mapa, que permiten al mapa crecer.

Se añade un nuevo *keyframe* cada vez que se cumplan las siguientes condiciones: (1) La calidad del seguimiento de la cámara tiene que ser buena, (2) el último *keyframe* fue añadido hace más de 20 fotogramas, y (3) cuando la cámara está a más de una mínima distancia de un punto importante del mapa.

Este último requisito evita un problema común del SLAM monocular de una cámara estacionaria, ya que puede corromper el mapa, y establece un distancia mínima para la triangulación de una nueva característica. Esta distancia depende de la distancia media a las características observadas, de manera que los *keyframes* están espaciados más cerca

cuando la cámara está muy cerca de una superficie, y más separados cuando se observan objetos distantes.

Cada *keyframe* inicialmente asume el seguimiento de la estimación de la posición y orientación de la cámara. Debido a las restricciones de tiempo real, el sistema de seguimiento sólo puede haber medido un subconjunto de las características potencialmente visibles en el fotograma. El hilo de mapeado reproyecta y mide el resto de características del mapa y añade observaciones existosas a su lista de mediciones.

El sistema de seguimiento calcula un conjunto de puntos característicos FAST para cada nivel de la pirámide. La supresión de puntos y la umbralización basada en la puntuación de *Shi-Tomasi* es usada para reducir el conjunto de puntos característicos de cada mivel de la pirámide. A continuación se descartan los puntos característicos cerca de las observaciones exitosas. Cada punto característico puede ser un candidato a nuevo punto del mapa.

Un nuevo punto del mapa requiere información de profundidad. No es posible obtener esta información con un sólo *keyframe*, y requiere triangulación con algún otro punto de vista. Se selecciona el *keyframe* más cercano (en términos de posición de la cámara) como segundo punto de observación. Las correspondencias entre los dos puntos de vista se establecen con una búsqueda epipolar. Los puntos alrededor de los puntos caraterísticos que se encuentran a los largo de la línea epipolar en el segundo punto de vista se compararán con los puntos del mapa candidato utilizando *zero-mean SSD*. No se utiliza ninguna transformación y se busca en el mismo nivel de la pirámide. Además, no se busca una línea epipolar infinita, se utiliza la hipotesis sobre la profundidad probable de nuevos puntos candidatos. Si se ha encontrado un coincidencia, el nuevo punto se triangula y es agregado al mapa.

Antes de proyectar los puntos, se actualiza la posición de la cámara en función de la velocidad y dirección original que tenía, de forma similar a lo realizado en la fase de predicción del filtro EKF, así la velocidad v y la nueva posición x de la cámara se calculan como sigue:

$$v_t = \frac{x_t - x_{t-1}}{\Delta t}$$
$$x_{t+1} = x_t + \Delta t\prime(v_t)$$

donde Δt es el tiempo transcurrido entre x_t y x_{t-1}, mientras que $\Delta t\prime$ es el tiempo entre x_t y x_{t+1}. Una vez tenemos la nueva posición, se proyectan una serie de puntos que serán buscados en la imagen, así, para la actualización general se proyecta un subconjunto de los 50 puntos más grandes que se deberían encontrar en la imagen, mientras que para la

actualización precisa se eligen 1000 puntos al azar. Estos puntos se comparan con la imagen actual teniendo en cuenta la posición y orientación de la cámara utilizando subimágenes de 8x8 píxeles.

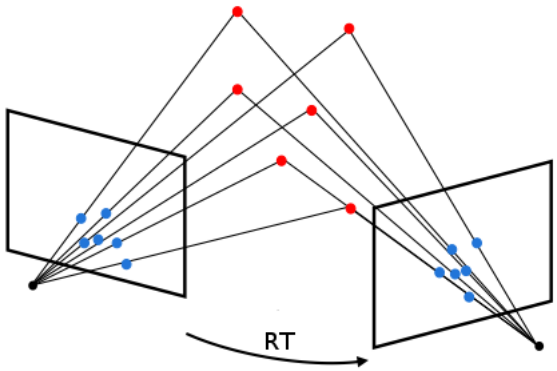


Figura 4.10: Rotación y translación entre dos *Keyframes*

La posición de la cámara se obtiene utilizando algoritmos clásicos de *Structure from Motion*, en concreto se utiliza el *algoritmo de 3 puntos* [Hartley and Zisserman, 2000] teniendo en cuenta los errores entre la proyección de cada uno de los puntos y su posición en la imagen asociada.

Estos sencillos pasos logran que el *Tracking* pueda ejecutarse en tiempo real, proporcionando además una gran precisión y robustez.

En resumen, el algoritmo de seguimiento necesita un mapa para localizarse en cada una de las imágenes de la secuencia de video. Para localizarse se realiza un emparejamiento entre la observación actual y los *keyframes*, una vez obtenidos los emparejamientos se realiza el algoritmo de los 3 puntos donde se obtiene una matriz de rotación y translación incremental respecto al último *keyframe*.

4.4.2. Mapeado

El *Mapping* comienza a partir de una inicialización en estéreo. Esta inicialización es manual, para ello es necesario mover la cámara ligeramente para que sea capaz de hacer un *Tracking* en la imagen de una serie de puntos de apoyo seleccionados al azar, a partir de los cuales calcula la posición inicial mediante un *algoritmo de los cinco puntos* [Nister, 2003]. Los puntos iniciales calculados, así como la posición de la cámara pueden no tener una posición en 3D totalmente reales, pero representarán a la realidad en una escala diferente.

Una vez realizada esta inicialización, el *Mapping* contará con los siguientes pasos en cada iteración:

- Esperar a un nuevo *Keyframe*
- Añadir nuevos puntos al mapa
- Optimizar el mapa
- Mantener el mapa

No todas las observaciones son añadidas como observaciones clave (*Keyframes*), sino que se tienen que cumplir las tres condiciones para utilizar una observación como *Keyframe*: que la distancia a otro *Keyframe* sea suficiente, el último *Keyframe* fue añadido hace más de 20 fotogramas y que la localización actual sea buena. Una vez que se almacena un *Keyframe*, se almacena la imagen con su pirámide de 4 niveles y sus esquinas detectadas.

Al añadirse un nuevo *Keyframe*, los puntos que ya se encuentran en el mapa se buscan en ese nuevo *Keyframe* y se añaden nuevos puntos al mapa en caso de existir. Para añadir estos nuevos puntos se busca en los *Keyframes* anteriores estos nuevos puntos para inicializar su posición mediante estéreo.

Mientras no se añadan nuevos *Keyframes*, el algoritmo intenta refinar el mapa con los *Keyframes* de los que dispone para mejorar tanto el mapa como la localización del *Tracking*. Para ello utiliza un algoritmo de optimización llamado ajuste de haces cuyo objetivo es minimizar el error de retroproyección con respecto a M posiciones en 3D de la cámara y N puntos en 3D:

$$min_{a_j,b_i} \sum_{i=1}^{n} \sum_{j=1}^{m} v_{ij} d(Q(a_j, b_i), x_{ij})^2$$

donde cada cámara está parametrizada con un vector a_j y cada posición en 3D con un vector b_i. $Q(a_j, b_i)$ es la proyección predicha del punto 3D i en la posición j de la cámara, x_{ij} es la proyección detectada, $d(x, y)$ es la distancia euclídea entre dos puntos de la imagen y v_{ij} una variable binaria que indica si el punto i es visible desde la posición j.

El algoritmo necesita como entrada tanto las proyecciones detectadas en cada imagen como una posición tentativa de las posiciones de las cámaras y de los puntos en 3D. El tiempo de cómputo de este algoritmo es muy alto, aunque no es preocupante al no necesitar el *Mapping* funcionamiento en tiempo real. No obstante, se han buscado fórmulas para

reducir este tiempo, como por ejemplo refinar los puntos únicamente con los *Keyframes* cercanos y ejecutando el algoritmo con todo el mapa en pocas ocasiones.

Por último, la fase de mantenimiento del mapa intenta comprobar la coherencia de los datos entre los distintos *Keyframes*, eliminando asociaciones de puntos mal realizadas o reintentando añadir nuevos puntos.

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

En resumen, el algoritmo cuenta con un mapa y los *Keyframes*. Esta información se combina con la observación de la cámara y su posición actual. En concreto se realiza un ajuste de haces que consiste en realizar un optimización no lineal que minimiza el error de retroproyección entre los *keyframes* y la observación actual. De tal forma que una vez calculado el mapa se pueden obtener nuevos puntos mediante triangulación estereo.

Actualmente se pueden encontrar algoritmos que hacen un mayor énfasis en el mapeado, obteniendo mapas no sólo basados en primitivas, como los puntos, sino en mallas densas. Este es el caso del algoritmo propuesto por Richard A. Newcombe [Newcombe *et al.*, 2011] denominado DTAM. Este es un sistema que estima la posición y orientación de la cámara y reconstruye no sólo las características puntuales extraídas de la imagen, sino todos los píxeles de la imagen con un cámara RGB. Se estiman mapas de profundidad detallados con texturas en los distintos *keyframes* produciendo una malla en 3D de millones de vértices. Se utilizan las distintas imágenes del flujo de vídeo para mejorar la estimación la calidad, minimizando una energía espacialmente regularizada. Este algoritmo es altamente paralelizable y DTAM se ejecuta en tiempo real gracias a la aceleración gráfica. Este método denso de reconstrucción de la escena permite la interacción física con el entorno permitiendo la creación de aplicaciones de Realidad Aumentada más complejas.

4.4.3. Comparativa con MonoSLAM

Se han realizado numerosas comparativas entre los dos algoritmos anteriores, PTAM y monoSLAM, como la realizada por Hauke Strasdat [Strasdat *et al.*, 2012]. En ellas se concluye que para mapas que tengan muy pocos puntos es más eficiente utilizar algoritmos de tipo MonoSLAM, mientras que para el resto de casos es mejor utilizar algoritmos de tipo PTAM.

Esto se produce ya que PTAM permite utilizar miles de puntos simultáneamente y sigue funcionando en tiempo real. Además, el algoritmo es más robusto, puesto que puede

recuperarse ante observaciones no favorables e incluso secuestros, algo que no es posible con MonoSLAM.

Por su parte, en caso de contar con mapas con muy pocos puntos, monoSLAM puede ser más eficiente, puesto que PTAM consume más tiempo de cómputo al generar el mapa, aunque sin influir en el *Tracking*.

También destacar que gracias a la parametrización inversa de la que hace uso monoSLAM éste no necesita inicializarse y puede arrancar solo. Por el contrario PTAM necesita un movimiento inicial la cámara.

Capítulo 5

Desarrollo informático

Una vez presentados los requisitos de la aplicación, las herramientas utilizadas y los fundamentos teóricos utilizados en el presente Trabajo Fin de Máster, en esta sección se describen las distintas soluciones software desarrolladas. A continuación se presenta cada uno de los bloques sotfware de la aplicación, el diseño detallado del conjunto de bloques y la implementación que se ha llevado a cabo en cada una de sus partes.

En este capítulo también se valida experimentalmente el sistema diseñado y programado. Los experimentos que lo validan consisten en una prueba de concepto, el desarrollo de un videojuego que hace uso de la Realidad Aumentada, la integración del sistema en un dispositivo Android y el piloto para la empresa Seabery.

5.1. Componente VisualSLAM

El objetivo principal de este componente es proporcionar una estimación de la posición y orientación de la cámara en tiempo real utilizando distintos algoritmos (DLT, monoSLAM y PTAM). En la Figura 5.1 se observa el diagrama de caja negra de componente VisualSLAM. La entrada al sistema es un flujo de imágenes que se reciben por una interfaz ICE estandar de JdeRobot (*image*) y la salida del sistema proporciona una estimación de la posición y orientación de la cámara, esta información se entrega mediante un interfaz ICE estandar de JdeRobot (pose3D). Se utilizan estas interfaces para facilitar la interoperabilidad con otros componentes de JdeRobot.

En la Figura 5.2 se observa el diagrama de bloques del componente Visual SLAM como una caja blanca. Se puede apreciar que el sistema cuenta con una interfaz de usuario (GUI) donde se visualizan las imágenes de la cámara y la posición y orientación en una

Figura 5.1: Diagrama de caja negra VisualSLAM

ventana 3D. Es posible utilizar la aplicación VisualSLAM sin hacer uso de la interfaz de usuario, para ello en el fichero de configuración se especifica la opción *noGUI* y la hebra de visualización no se ejecuta.

Por otro lado, el bloque de procesamiento 2D de la imagen se encarga de realizar el emparejamiento entre los puntos característicos de la secuencia de imágenes recibidas. En el caso de monoSLAM mediante el emparejamiento de esquinas explicado en la sección 4.3.2 y en el caso de PTAM el basado en el detector FAST explicado en la sección 4.4.1.

Y por último, el núcleo de la aplicación permite resolver el problema de la estimación de la posición y orientación de la cámara de tres maneras diferentes, mediante monoSLAM (código disponible en JdeRobot) y las dos implementaciones realizadas en este Trabajo Fin de Máster: DLT y PTAM.

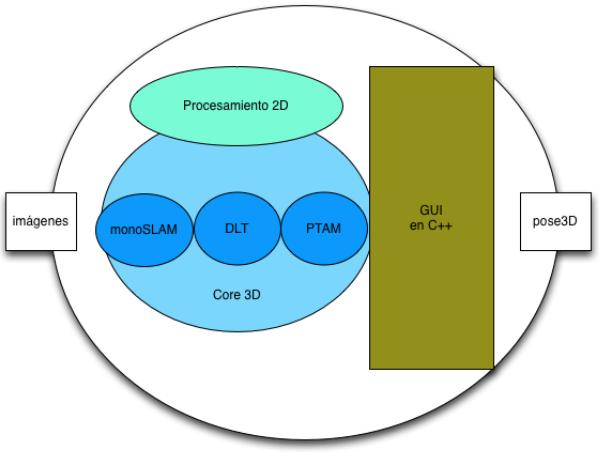


Figura 5.2: Diagrama de bloques Visual SLAM (caja blanca)

A continuación se detalla el bloque de procesamiento 2D y dentro del bloque CORE 3D los algoritmos: DLT y PTAM.

5.1.1. Procesamiento 2D

En las implementaciones de los algoritmos de autolocalización visual desarrollados (Capítulo 4) se explicaba cúal es el procesamiento de imagen correspondiente para cada algoritmo. En el caso de PTAM los puntos característicos de la imagen de extraen utilizando FAST (4.4.1) y para realizar el emparejamiento entre los distintos puntos de los fotogramas se utilizan parches de pequeñas dimensiones que se asocian utilizando las puntuaciones de *Zero mean SSD*. Por otro lado, monoSLAM utiliza el extractor de caraterísticas de OpenCV *goodFeaturesToTrack*, explicado en la sección 4.3.2. La asociación entre los distintos puntos caraterísticos se realiza utilizando la ecuación de divergencia 4.33 explicada en la sección 4.3.2. Para el prototipo desarrollado a la empresa Seabery se utiliza la versión de monoSLAM explicada en la sección 4.3 pero modificando el procesamiento de imagen. En este caso el objetivo era reconocer unos patrones conocidos a priori. A continuación se explica el proceso de extracción de los marcadores en la imagen.

La Realidad Aumentada basada en marcadores utiliza símbolos que el software interpreta. Hay diversos tipos de marcadores, lo más comunes son los llamados marcadores de códigos matriciales. Este tipo de marcador no fue diseñado para el uso en la Realidad Aumentada, sino que fueron inventados como el sucesor de los códigos de barras. Pero los marcadores pueden tener cualquier apariencia, como por ejemplo un cromo. En la Figura 5.3 se muestra un marcador utilizado en Realidad Aumentada.

Figura 5.3: Marcador utilizado en Realidad Aumentada

En esta sección se describe el proceso realizado para detectar los marcadores de la imagen. Los patrones tienen un fondo blanco y están codificados de forma matricial. El sistema de detección de los patrones contiene varios pasos: un filtro de color, búsqueda de polígonos de 4 lados, eliminación de la perspectiva, umbralización e identificación del patrón entre los conocidos.

Para hacer la detección más sencilla se ha utilizado un filtro de color HSV ya que es más robusto a cambios de iluminación que otros filtros como RGB. En la Figura 5.4(a) se puede ver cómo queda la imagen después del filtro de color.

A continuación se hace una búsqueda de contornos, utilizando la función *findContours* de *OpenCV*. Este proceso no devuelve únicamente los marcadores sino otros muchos bordes que puedan aparecer en la imagen. La función *findContours* recibe como entrada una imagen en escala de grises. Con *CV_RETR_TREE* se configura el modo, en éste caso se indica que se devuelven todos los contornos y topología de la imagen. Con *CV_CHAIN_APPROX_NONE* almacena todos los puntos del contorno. Las variables *contours* y *hierarchy* contienen los contornos de la imagen y la topología de la imagen respectivamente. Es necesario filtrar, se eliminan aquellos contornos que tengan un números reducido de puntos. En los que pasen esta etapa se realiza una aproximación poligonal, utilizando la función *approxPolyDP* de *OpenCV*. Si el polígono tiene 4 lados, no es excesivamente pequeño, no está dentro de otro polígono y es convexo será un posible candidato a marcador.

```
1    /*Find contours*/
2    cv::findContours(this->image, contours, hierarchy, CV_RETR_TREE
3                     CV_CHAIN_APPROX_NONE );
```

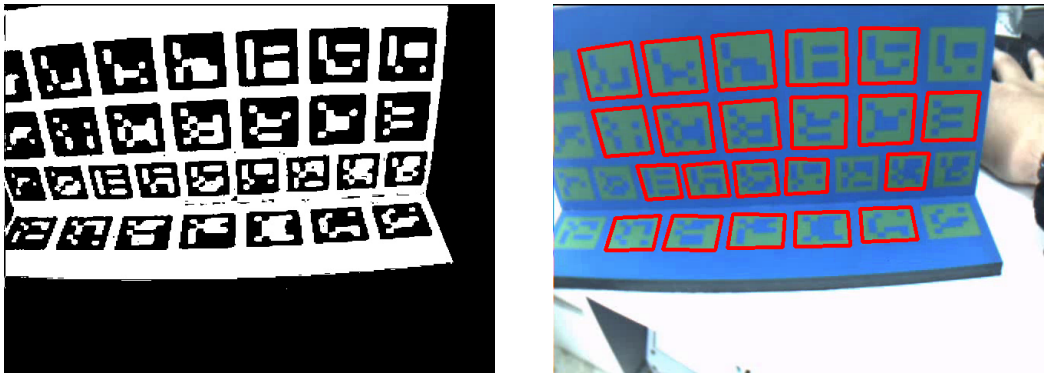
La función *approxPolyDP* aproxima los contornos a polígonos. El primer parámetro de la función es un contorno calculado por la función *findContours*. El segundo parámetro es el resultado de la aproximación. El tercer parámetro, *epsilon*, especifica la precisión de la aproximación. Y por último, el booleano indica si el polígono está cerrado. Si está a verdadero, el polígono aproximado tiene que estar cerrado. En caso contrario, no tiene que estar cerrado.

```
1      cv::approxPolyDP(contours[i], approximation,
2                       double(contours[i].size())*0.1, true);
```

Por último, se ordenan en el sentido de las agujas del reloj. En la Figura 5.4(b) se pueden observar de color rojo los distintos polígonos de 4 lados encontrados en la imagen.

Una vez que se dispone de los candidatos es necesario conocer si es un patrón y en caso de serlo a qué coordenada respecto al eje de referencia pertenece cada vértice del marcador. Para esta tarea lo primero es eliminar la proyección perspectiva para obtener una vista frontal del rectángulo utilizando una homografía. Se binariza la imagen utilizando *Otsu*, este algoritmo asume una ditribución bimodal y encuentra el umbral óptimo que maximiza la varianza entre clases permitiendo separar el blanco del negro. En la Figura 5.5(a) se puede observar la homografía del marcador filtrada.

El marcador está dividido en una rejilla de 5x5 (Figura 5.5(b)) que contiene la información. Para dedicir si un cuadrado de la rejilla es blanco o negro se cuenta el número



(a) (b)

Figura 5.4: (a) Imagen filtrada (b) Polígonos de cuatro lados

de píxeles que es distinto de cero y si supera un umbral se considera negro, en caso contrario se considera blanco. Todos los puntos tienen almacenado en memoria su posición 3D, respecto al eje de referencia principal. Realizando la resta entre el marcador encontrado y los marcadores conocidos aquella distancia que sea igual a cero dará como válido al marcador, puede ser necesario rotar los marcadores para encontrar una correspondencia.

(a) (b)

Figura 5.5: (a) Homografía del patrón (b) Homografía del patrón con rejilla

5.1.2. Core 3D: DLT

Una vez obtenidos los puntos de cada uno de los marcadores reconocidos en la imagen se procede a resolver la localización visual utilizando DLT, éste algoritmo resuelve el sistema optimizando mediante SVD (*Singular Value Descomposition*).

Para combinar la información obtenida de cada uno de los puntos del marcador encontrado en la imagen. De cada marcador se obtienen 4 puntos, uno por cada esquina. Por lo tanto, se añaden dos ecuaciones nuevas a la optimización por cada punto que tenga su correspondencia en 3D. Estas ecuaciones (4.11 y 4.10) se encuentran descritas en la sección 4.2. Cuantos más puntos se añadan al sistema de ecuaciones mayor precisión se obtiene en la solución.

Esta información tiene que ser introducida en Eigen para que resuelva el sistema de ecuaciones, para ello es necesario reservar el espacio en memoria y rellenar la memoria convenientemente. En la implementación desarrollada se cuenta con dos listas de puntos, por un lado los puntos 2D reconocidos en la imagen (*listaPuntos2D*) y por otro lado la correspondencia 3D de cada uno de los puntos*listaPuntos3D*.

```
1     Eigen::MatrixXd B( listaPuntos2D.size()*2 , 12 );
2     // Ax = b;
3     for(unsigned i = 0; i < listaPuntos2D.size(); i++){
4
5         cv::Point3f p3D = listaPuntos3D[i];
6         cv::Point2f p2D = listaPuntos2D[i];
7
8         //A
9         B(i*2, 0) = p3D.x;
10        B(i*2, 1) = p3D.y;
11        B(i*2, 2) = p3D.z;
12        B(i*2, 3) = 1;
13        B(i*2, 4) = 0;
14        B(i*2, 5) = 0;
15        B(i*2, 6) = 0;
16        B(i*2, 7) = 0;
17        B(i*2, 8) =  -p3D.x*p2D.x;
18        B(i*2, 9) =  -p3D.y*p2D.x;
19        B(i*2, 10) = -p3D.z*p2D.x;
20        B(i*2, 11) = -p2D.x;
21
22        B(i*2+1, 0) = 0.;
23        B(i*2+1, 1) = 0.;
24        B(i*2+1, 2) = 0.;
25        B(i*2+1, 3) = 0.;
26        B(i*2+1, 4) = p3D.x;
27        B(i*2+1, 5) = p3D.y;
28        B(i*2+1, 6) = p3D.z;
29        B(i*2+1, 7) = 1.;
30        B(i*2+1, 8) =  -p3D.x*p2D.y;
31        B(i*2+1, 9) =  -p3D.y*p2D.y;
32        B(i*2+1, 10) = -p3D.z*p2D.y;
33        B(i*2+1, 11) = -p2D.y;
34    }
```

Una vez rellena la estructura de datos se resuelve el sistema utilizando la SVD. La

función utilizada para resolver el sistema es *JacobiSVD* donde se pasa como argumento el sistema de ecuaciones y se indica qué matriz de la descomposición SVD se quiere calcular, ya que en caso de no necesitar las 3 matrices se puede calcular únicamente la que se necesite ahorrándose cálculos. En concreto, la matriz de proyección se encuentra en la matriz V del SVD. Por lo tanto, se obtiene la matriz de proyección que posteriormente será descompuesta para extraer los valores de localización.

```
1
2   // se resuleve el sistema
3     Eigen::JacobiSVD<Eigen::MatrixXd> svd(B, Eigen::ComputeFullV);
4
5     svd.computeV();
6     Eigen::MatrixXd V = svd.matrixV();
7
8   // se obtiene la matriz P
9     Projection = Eigen::MatrixXd(3, 4);
10
11    Projection(0, 0) = V(0, 11);
12    Projection(0, 1) = V(1, 11);
13    Projection(0, 2) = V(2, 11);
14    Projection(0, 3) = V(3, 11);
15
16    Projection(1, 0) = V(4, 11);
17    Projection(1, 1) = V(5, 11);
18    Projection(1, 2) = V(6, 11);
19    Projection(1, 3) = V(7, 11);
20
21    Projection(2, 0) = V(8, 11);
22    Projection(2, 1) = V(9, 11);
23    Projection(2, 2) = V(10, 11);
24    Projection(2, 3) = V(11, 11);
```

Finalmente se obtiene la orientación y posición de la cámara descomponiendo la matriz de proyección según lo explicado en la sección 4.2.2.

5.1.3. Core 3D: Refactorización de PTAM

En el presente Trabajo Fin de Máster se ha realizado una implementación propia del algoritmo PTAM. El primer motivo era comprender bien el algoritmo por completo. El segundo motivo era encontrar sus puntos débiles para mejorarlo. La implementación que

está disponible en la pagina web del autor[1] no utiliza librerías estándar, utiliza librerías de su grupo de investigación. En particular utiliza *TooN*, librería para el algebra lineal, *libCVD*, librería para el manejo de imágenes, captura de video y visión artificial y *Gvars* una librería para la configuración en tiempo de ejecución. Un tercer motivo de este enfoque era actualizar el código para usar librerías estándar como OpenCV y Eigen.

En esta sección se presentan los detalles más importantes de las clases implementadas para completar el algoritmo PTAM. Se destacarán los métodos y atributos más relevantes de las clases principales.

Construcción del mapa

La refactorización realizada parte de la separación en dos hilos como se explica en la sección 4.4. Un hilo para el *tracking* y otro para el *mapping*. Los hilos utilizados forman parte de la librería Qt. Para hacer uso de estos hilos la clase tiene que heredar de la clase *QThread* e implementar el método protegido *run*, este método ejecuta en otro hilo cada vez que se llame al método *start*. Por ejemplo, el hilo de visualización se ejecuta bajo un motor temporal que cumple con las restricciones de tiempo real, ejecutandose a 25fps. A continuación se muestra la definición de la clase que forma parte de la visualización:

```
1 class threadGUI: public QThread {
2 public:
3     threadGUI();
4     ...
5 protected:
6     void run();
7     ...
8 };
```

El modelo de cámara está definido en la clase *ATANCamera*. Esta clase contiene los siguientes atributos normalizados: distancia focal en X f_x, distancia focal en Y f_y, centro óptico en X u_0 y centro óptico en Y v_0. También contiene métodos para proyectar y retroproyectar. Esta clase está basada en los fundamentos teóricos explicados en la sección 4.1.

El algoritmo de PTAM se basa en el uso de los fotogramas claves. La definición de estos se realiza con la clase *Keyframe*. Esta clase contiene: la imagen a color, la pirámide

[1] http://www.robots.ox.ac.uk/ gk/PTAM/

de imágenes, la posición y orientación del *Keyframe* y todos los puntos característicos. El método *MakeKeyFrame_Lite* prepara el *keyframe* desde una imagen de la secuencia de vídeo, generando la pirámide de imágenes y utilizando FAST para la detección de los puntos característicos.

En general a los métodos se les pasa por referencia los argumentos para evitar hacer una copia cada vez que se llama a la función. El algoritmo gestiona continuamente imágenes que se pasan como argumento a los métodos, por lo tanto, si continuamente se tiene que realizar una copia se perdería mucha eficiencia. Este tipo de *optimización* se realiza a lo largo de todo el código. Por otro lado, haciendo uso de la palabra reservada en C++ *inline*, se le indica al compilador que cada llamada a la función *inline* debe ser reemplazada por el cuerpo de la función o método. En general la directiva *inline* es utilizada sólo cuando las funciones son pequeñas para evitar generar un ejecutable de tamaño considerable. Esta palabra reservada tiene la ventaja de acelerar un programa si éste invoca frecuentemente a la función *inline*, ya que esta evita usar la pila para pasar parámetros y evita las instrucciones de salto y retorno. Permite resumir considerablemente el código, en particular para acceder a una clase o variable de una clase.

Como se ha explicado anteriormente, el seguimiento de la cámara está separado de la construcción del mapa. A continuación se describen las clases más importates para la creación del mapa.

PTAM necesita un mapa para poder localizarse pero éste no debe ser introducido por el usuario, el algoritmo debe inicializarlo de manera autónoma. En la sección 4.4 se explican los detalles de la inicialización. La inicialización del algoritmo se realiza con el método *InitFromStereo*. Éste recibe como parametros dos *Keyframes*, el primero es donde se inició la inicialización y el segundo es el obtenido cuando se ha dado por concluida la fase de inicialización. La variable *vMatches* contiene los emparejamientos de los puntos 2D entre los dos *Keyframes*. Por último, este método devuelve la estimación de la posición y orientación de la cámara utilizando el algoritmo de los 5 puntos y crea el mapa inicial.

```
1  bool InitFromStereo(KeyFrame &kFirst, KeyFrame &kSecond,
2       vector<pair<Eigen::Vector2d, Eigen::Vector2d> > &vMatches,
3       mySE3 &se3TrackerPose);
```

La clase *Bundle* es el núcleo del ajuste de haces. Esta clase realiza el ajuste de la posición de los *keyframes* y los puntos del mapa. En concreto implementa el método de ajuste de haces de Levenberg-Marquardt. La implementado se ha realizado siguiendo el libro *Multiple View Geometry in Computer Vision* [Hartley and Zisserman, 2000]. Esta clase se instancia

pasándole como parámetro el modelo de cámara ya que el algoritmo tiene que realizar retroproyecciones, es decir, pasar un punto 3D a 2D. El método *AddCamera* añade un nuevo punto de vista asociado a un *keyframe*, si *bfixed* es falso indica que está aún no ha sido ajustado con el mapa. El método *AddPoint* añade un nuevo punto al mapa. Y por último, el método más importante de esta clase, *Compute*, es el que realiza el ajuste de haces. Esta función devuelve el número de iteraciones realizadas o un número negativo si ha ocurrido algún error.

```
1  //constructor
2  Bundle(const ATANCamera &TCam);
3  //nueva pose
4  int AddCamera(mySE3 se3CamFromWorld, bool bFixed);
5  //nuevo punto
6  int AddPoint(Eigen::Vector3d v3Pos);
7  //ajuste de haces
8  int Compute();
```

La clase *Relocaliser* se encarga de relocalizar la posición y orientación de la cámara. Para ello cada *Keyframe* guarda una versión desenfocada de sí mismo. Esta versión desenfocada se compara con la imagen de entrada, también desenfocada y se busca la coincidencia más cercana basado en la puntuación calculada en el método *ScoreKFs* (zero-mean SSD) y a continuación se estima la rotación de la cámara.

```
1  bool AttemptRecovery(KeyFrame &k);
2  void ScoreKFs(KeyFrame &kCurrentF);
```

La clase *PatchFinder* se encarga de encontrar un punto de mapa en un nuevo fotograma. Este proceso se realiza en varias etapas. La primera consiste en calcular una matriz de deformación para la vista actual. El método *CalcSearchLevelAndWarpMatrix* se encarga de calcular la matriz de transformación en un nivel de la pirámide concreto. A continuación se genera una plantilla del punto del mapa. El método *MakeTemplateCoarseCont* genera la plantilla deformada. La tercera etapa, el método *FindPatchCoarse* busca esta plantilla alrededor de un radio concreto (*nRange*) en los puntos característicos detectados por FAST. Y por último, se realiza una búsqueda más fina para un mayor precisión. El método *IterateSubPixToConvergence* e encarga de realizar una búsqueda iterativa que nunca superará una cierta distancia y un número máximo de iteraciones. A continuación se muestran las cabeceras de los métodos explicados:

```
1  //paso 1
2  int CalcSearchLevelAndWarpMatrix(MapPoint &p, mySE3 se3CFromW, Eigen::
       Matrix2d &m2CamDerivs);
3  //paso2
4  void MakeTemplateCoarse(MapPoint &p, mySE3 se3CFromW, Eigen::Matrix2d &
       m2CamDerivs);
5  //paso 3
6  bool FindPatchCoarse(Eigen::Vector2d ir, KeyFrame &kf,
7                       unsigned int nRange);
8  //paso 4
9  bool IterateSubPixToConvergence(KeyFrame &kf, int nMaxIts);
```

La clase más importate de la construcción el mapa es la clase *MapMaker*. Esta se encarga de mantener y construir el mapa. El mapa también está representado como una clase (*Map*), que contiene un vector con los puntos del mapa y otro vector con los *keyframes*. La construcción del mapa comienza con la inicialización estereo, añade nuevos *Keyframes*, realiza el ajuste de haces y ajusta la asociación de los datos. Esta clase implementa el hilo para la construcción del mapa. Los métodos más detacados son:*AddKeyFrameFromTopOfQueue*, que añade un nuevo *keyframe*; *AddSomeMapPoints* que añade un nuevos puntos al mapa; *AddPointEpipolar* que intenta crear un nuevo punto en el mapa de un solo candidato buscando este punto en otro *keyframe* y triangula si encuentra la coincidencia; *ClosestKeyFrame* que devuelve el *keyframe* más cercano; y *BundleAdjust* que realiza el ajuste de haces haciendo uso de las clases descritas anteriormente.

```
1  //nuevo keyframe
2  void AddKeyFrameFromTopOfQueue();
3  //nuevo puntos al mapa
4  void AddSomeMapPoints(int nLevel);
5  //epipolar
6  bool AddPointEpipolar(KeyFrame &kSrc, KeyFrame &kTarget, int nLevel, int
       nCandidate);
7  //devuelve el keyframe ms cercano
8  KeyFrame* ClosestKeyFrame(KeyFrame &k);
9  //realiza el ajuste de haces
10 void BundleAdjust(std::set<KeyFrame*>, std::set<KeyFrame*>, std::set<
       MapPoint*>, bool);
```

En la Figura 5.6 se puede observar el diagrama de bloques de la refactorización de PTAM. Por un lado, se dispone del hilo de localización y construcción del mapa, gestionado por la clases *MapMaker*. Y por otro lado, se dispone del otro hilo para el seguimiento y procesamiento de la imagen. Este hilo lo controla la clase *Tracker*. Estos dos hilos se comunican entre si haciendo uso de la exclusión mutua. Y por último, el hilo de la interfaz gráfica que recibe información de los dos hilos descritos anteriormente.

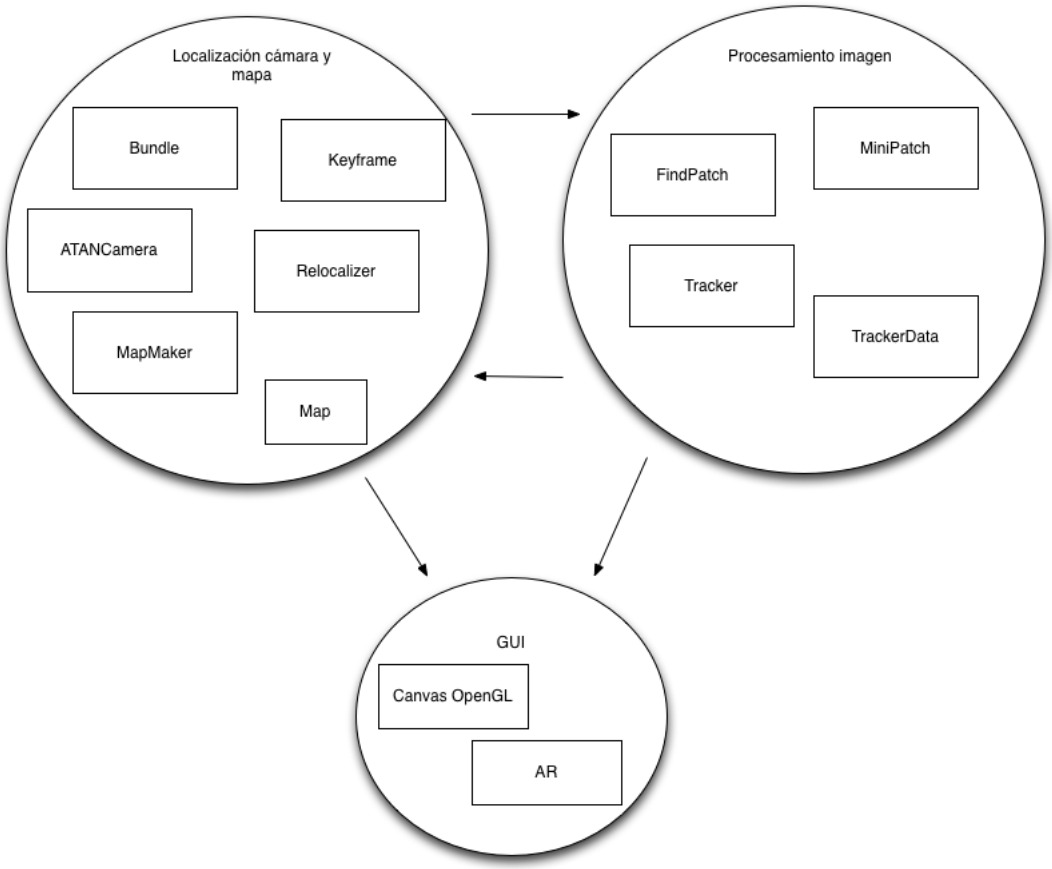


Figura 5.6: Diagrama de bloques con las clases de PTAM

Seguimiento de la cámara

Por otro lado, se dispone del hilo de seguimiento de la cámara que utiliza una serie de clases que abstraen el problema. A continuación se exponen las clases más importates.

Para expresar la posición y orientación de la cámara se han implementado dos clases. Por un lado, *SO3* que representa una matriz de rotación en el espacio tridimensional. Esta clase contiene métodos como: *coerce* que comprueba que la matriz de rotación es

valida, *inverse* que realiza una inversa optimizada o la sobrecarga de operadores para mostrar por pantalla el valor de la matriz de rotación. Y por otro lado, *SE3* que representa una transformación, una rotación (S03) y una translación. La translación es expresada como un vector tridimensional que representa una translación en los ejes X, Y y Z. Esta clase también incluye método para realizar la inversa o la sobrecarga de operadores para multiplicar (por vectores o matrices) o mostrar la matriz por pantalla.

La clase *Tracker* es una de las clases más importantes del sistema. También es la más relacionada con la visión artificial. Es la responsable de determinar la posición y orientación de la cámara de la secuencia de vídeo. Utiliza la clase *Map*, que contiene el mapa, para localizarse y se comunica con la clase *MapMaker* que se encarga de contruir el mapa (esta ejecuta en un hilo diferente).

El método *TrailTracking_Start* es llamado únicamente cuando el usuario indica que se va a arrancar la inicialización del algoritmo (con la barra espaciadora). En este método prepara toda la información necesaria antes de comenzar con el seguimiento 2D, como por ejemplo, guardar el *keyframe* y almacenar sus puntos característicos. Para realizar el seguimiento 2D de los puntos característicos de la imagen se ha creado la clase *MiniPatch*. Esta clase contiene dos métodos: *SampleFromImage* que define un parche en una imagen y *FindPatch* que realiza la búsqueda sobre los puntos FAST de la imagen de entrada. El método *TrailTracking_Advance* se encarga de realizar el seguimiento 2D en la fase de inicialización haciendo uso de las clases *MiniPatch* y *FindPatch*. Una vez terminada la fase de inicialización (pulsado de nuevo la barra espaciadora) se llama al método *InitFromStereo* explicado anteriormente.

El método *TrackMap* se encarga de realizar el seguimiento de la cámara cuando el algoritmo está inicializado, es decir, tiene un mapa. Para ello es necesario hacer el seguimiento 2D sobre los puntos característicos de la imagen. La explicación del algoritmo del seguimiento 2D de los puntos se puede encontrar en la sección 4.4.1. Este método proyecta todos los puntos del mapa en la imagen y se queda con aquellos que caen dentro de la imagen. A continuación intenta encontrar nuevos puntos en la imagen. Posteriormente actualiza la posición y orientación de la cámara de acuerdo a los puntos del mapa encontrados. La clase *PatchFinder* busca la proyección de un punto del mapa entre el fotograma actual y los *keyframes* y la clase *TrackerData* maneja las proyecciones de los puntos mapa y guarda los resultados intermedios.

```
1  void TrailTracking_Start();
2  int  TrailTracking_Advance(cv::Mat &imageColor);
3  void TrackMap(cv::Mat &imageColor);
```

La interfaz gráfica ha sido desarrollada íntegramente en Qt. En la Figura 5.7 se puede las dos ventanas creadas para visualizar y depurar el algoritmo. En la imagen de la izquierda se muestra la imagen junto a la Realidad Aumentada y en la imagen de la derecha se muestra un mundo 3D, utilizando OpenGL, que pinta los puntos del mapa generado y la posición y orientación de la cámara estimados.

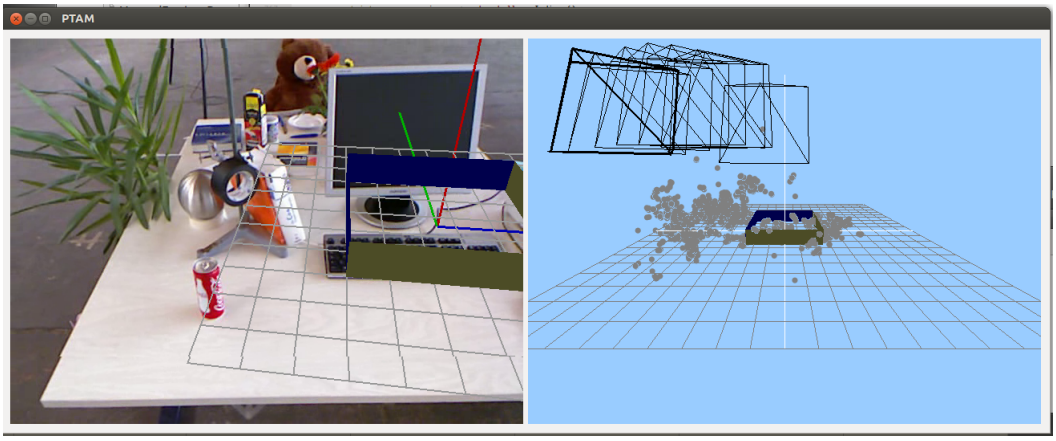


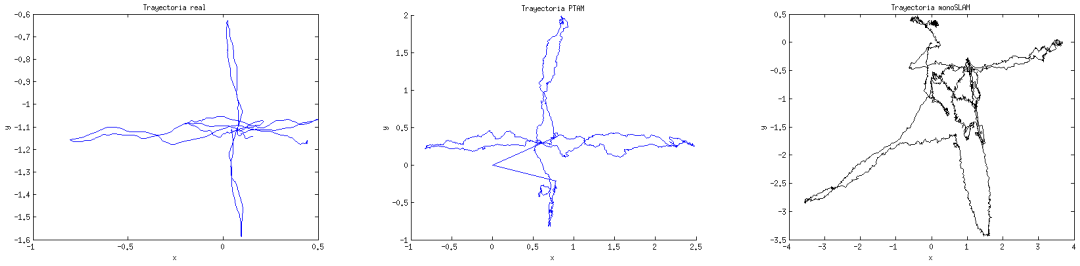
Figura 5.7: Interfaz gráfica de PTAM

5.2. Experimentos VisualSLAM

En esta sección se describen los experimentos realizados con el componente VisualSLAM que permiten validarlo. En primer lugar se describe la comparativa cualitativa realizada entre PTAM y monoSLAM. En esta sección se pretende comprobar el funcionamiento de los algoritmos de autolocalización visual, PTAM y monoSLAM, realizar una comparativa y discutir cuáles son sus puntos fuertes y débiles. A continuación se analizan una serie de experimentos realizados, donde se incluye el piloto desarrollado para la empresa Seabery con DLT y monoSLAM. También se describe la implementación realizada para el desarrollo de un videojuego. Y por último, se indican cuáles son los pasos necesarios para integrar PTAM es un dispositivo Android realizando el mínimo número de cambios al código implementado.

5.2.1. Validación de la autolocalización 3D

En primer lugar se comprobó que la trayectoria que el componente Visual SLAM estima de la cámara es la correcta. Para ello se ha utilizado un video de una base de datos [2], de la que se dispone de un vídeo de entrada y la posición real de la cámara en cada fotograma. Por lo tanto, el movimiento de la cámara es conocido, la cámara realiza un movimiento de izquierda a derecha y posteriormente se acerca y aleja la cámara. En la Figura 5.8(a) se puede observar cuál es la trayectoria real. En la Figura 5.8(b) y 5.8(c) se puede observar el movimiento estimado por el algoritmo, PTAM y monoSLAM respectivamente, sobre los ejes X e Y. En ambos casos la trayectoria descrita es similar a la real excepto la escala. Por esta razón se representa en figuras separadas. Estos métodos necesitan que se le indique la escala mediante el reconocimiento objetos o puntos conocidos de la secuencia de vídeo.



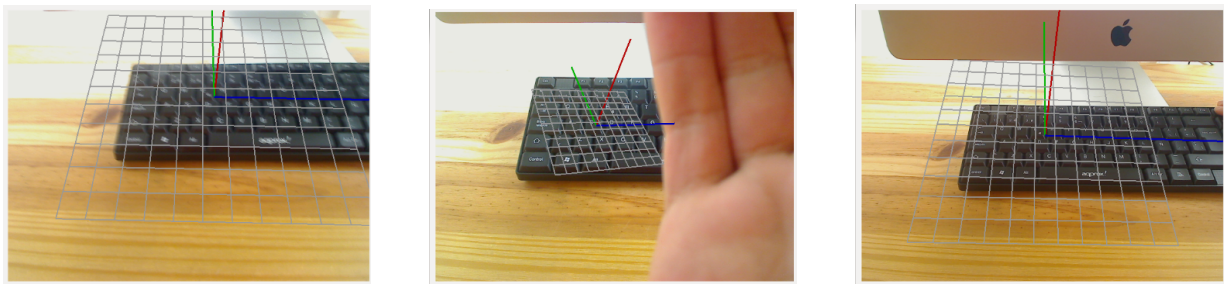
(a) (b) (c)

Figura 5.8: Trayectoria descrita por la cámara sobre los ejes X e Y

5.2.2. Robustez frente a oclusiones

Una de las características que presenta PTAM y que supone una mejora respecto a monoSLAM es la capacidad de recuperarse tras una oclusión temporal en la imagen, parcial o total. Se han estudiado dos casos diferentes de oclusiones. El primero cuando la oclusión es parcial. Como se muestra en la secuencia de imágenes de la Figura 5.9 y 5.10. En el primer fotograma de la secuencia ambos algoritmos están localizados, PTAM representa una rejilla y unos ejes mediante Realidad Aumentada. MonoSLAM tiene un pequeño dibujo debajo de las fotografías que indica cuál es la posición de la cámara. La oclusión parcial se produjo poniendo la mano parcialmente sobre el objetivo de la cámara. El sistema es capaz de recuperarse en ambos casos, ya que la imagen aún contiene puntos en los que apoyarse.

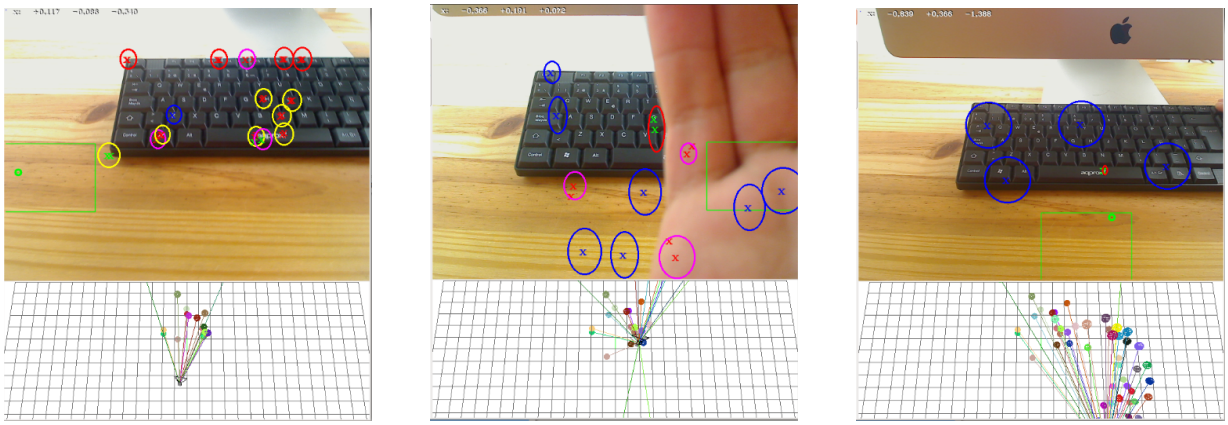
[2] http://vision.in.tum.de/data/datasets/rgbd-dataset/download



(a) (b) (c)

Figura 5.9: Secuencia de una oclusión temporal parcial en PTAM

Puede darse algún caso en monoSLAM en el que la cantidad de puntos en los que apoya la estimación no sea suficiente y el algoritmo termine perdiendo la estimación de la posición y orientación de la cámara aunque la oclusión sólo sea parcial.

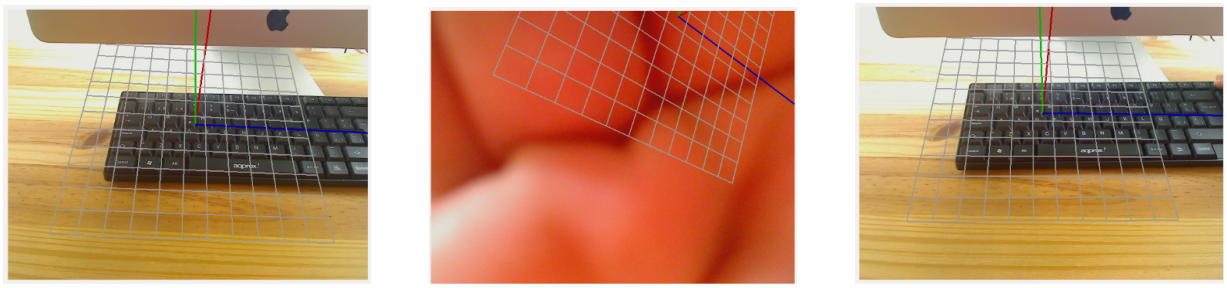


(a) (b) (c)

Figura 5.10: Secuencia de una oclusión temporal parcial en monoSLAM

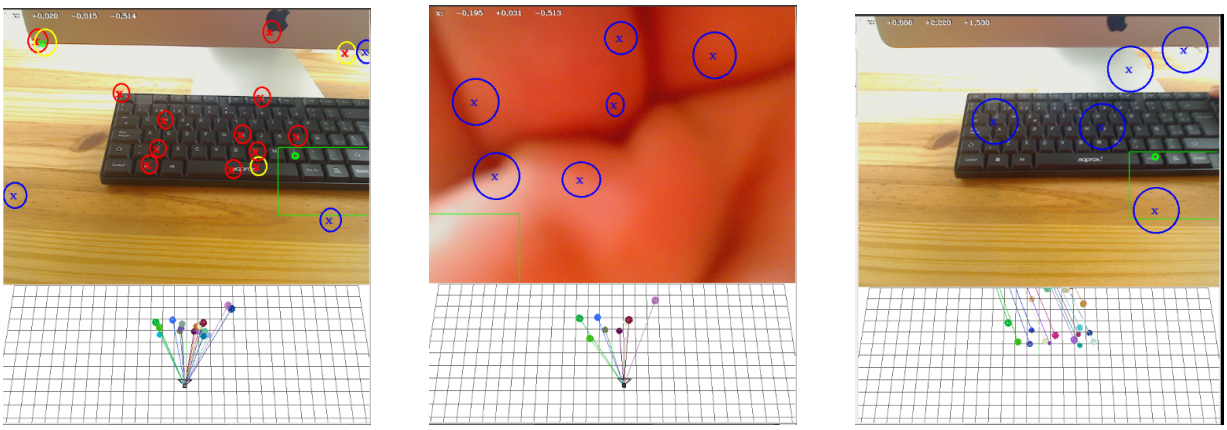
Sin embargo, cuando se produce una oclusión total, por ejemplo tapando el objetivo de la cámara, el algoritmo monoSLAM se pierde completamente, ya que no contiene puntos en la imagen suficientes para mantener su estimación de la posición y orientación de la cámara. En la Figura 5.12(a) se aprecia cómo la cámara esta localizada, pero cuando se tapa el objetivo (Figura 5.12(b)) y se vuelve a destapar (Figura5.12(c)) la cámara ha perdido la correcta estimación de su posición.

Mientras que PTAM, gracias a su algoritmo de relocalización, es capaz de recuperar correctamente la estimación. En la Figura 5.11(a) se puede observar cómo el algoritmo PTAM está bien localizado mostrando la rejilla justo encima del teclado. En la Figura 5.11(b) se tapa el objetivo de la cámara, instantáneamente el algoritmo se pierde ya que



(a) (b) (c)

Figura 5.11: Secuencia de una oclusión temporal total en PTAM



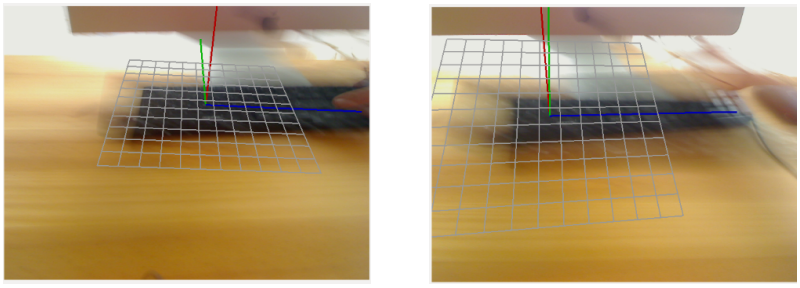
(a) (b) (c)

Figura 5.12: Secuencia de una oclusión temporal total en monoSLAM

no dispone de puntos en los que apoyar su estimación, pero cuando el objetivo se destapa (Figura 5.11(c)) el algoritmo es capaz de relocalizarse.

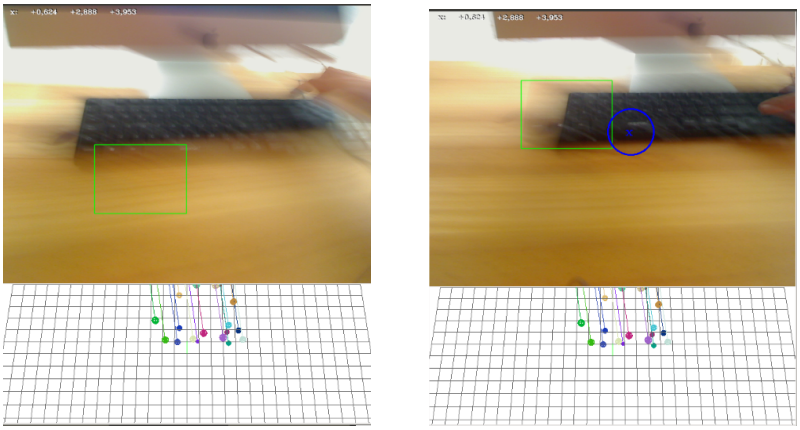
5.2.3. Robustez frente a movimientos bruscos y desenfoque

Los algoritmos basados en marcas naturales necesitan apoyarse en las texturas de los objetos para obtener una estimación de la posición. En caso de no tener suficiente textura el algoritmo no funcionará correctamente. Esto ocurre cuando dirigimos la cámara hacia una pared blanca, en este caso tanto PTAM como monoSLAM no son capaces de localizarse. Pero, como se ha explicado anteriormente, PTAM será capaz de recuperar la estimación en cuanto vuelva a una zona conocida con textura suficiente. MonoSLAM en cambio perderá completamente la estimación de la posición.



(a) (b)

Figura 5.13: Secuencia de movimiento rápido de la cámara en PTAM



(a) (b)

Figura 5.14: Secuencia de movimiento rápido de la cámara en monoSLAM

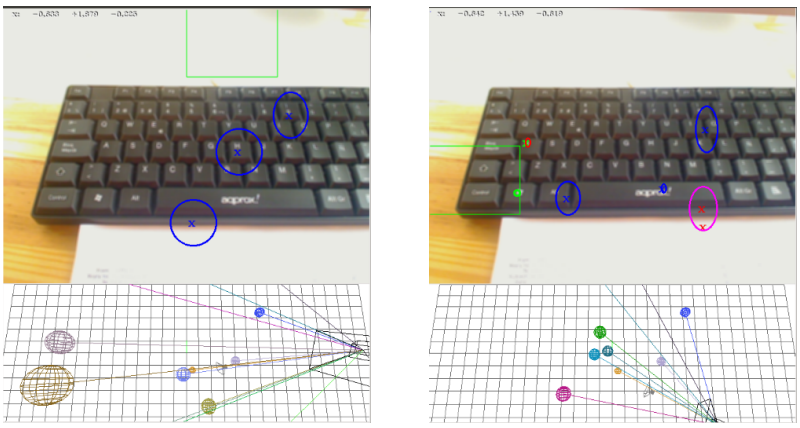
Un punto fuerte de PTAM frente a monoSLAM es que es capaz de soportar movimientos

rápidos de la cámara como se muestra en la Figura 5.13. Aunque el movimiento produzca desenfoque en la imagen (Figura 5.13(a)) el algoritmo PTAM es capaz de mantener la estimación de la posición y orientación de la cámara. Cuando este movimiento es excesivo, como el que se puede observar en la Figura 5.13(b) la estimación pierde algo de calidad. Como se aprecia en la Figura 5.14 en el caso de monoSLAM al aparecer el desenfoque en la imagen el algoritmo se pierde de inmediato.

(a) (b)

Figura 5.15: Secuencia con desenfoque cámara en PTAM

En caso de tener desenfoque en la imagen sin movimientos rápidos de la cámara, como en el punto anterior, también se porta mejor el algoritmo PTAM. Como se aprecia en la Figura 5.15 aunque la imagen este desenfocada, PTAM es capaz de inicializarse y localizarse correctamente. Sin embargo, monoSLAM pierde completamente la estimación ya que los puntos sobre los que apoya su estimación no son lo suficientemente robustos.



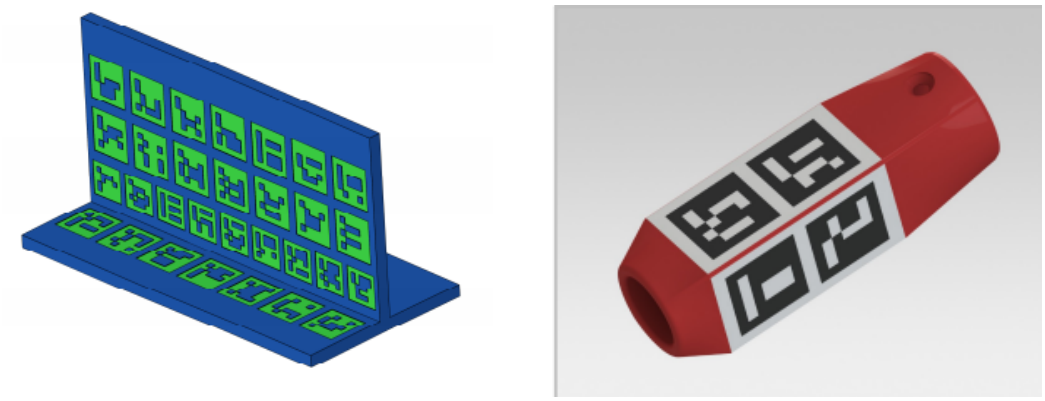
(a) (b)

Figura 5.16: Secuencia con desenfoque en monoSLAM

5.2.4. Piloto Seabery

En esta sección se detallan los experimentos realizados para el prototipo de autolocalización de cámaras desarrollado para la empresa Seabery. Como se indica en la introducción, esta empresa comercializa una herramienta denominada *Soldamatic*, un método de formación en soldadura utilizando la Realidad Aumentada.

El problema a resolver consta de dos partes. Por un lado, la detección de la pieza principal sobre la que se va a soldar, denominada Unión en T. Y por otro lado, la detección de la punta del soldador. En la Figura 5.17(a) se puede observar la pieza sobre la que se encuentran los marcadores, esta pieza contiene 56 marcadores distintos. En la Figura 5.17(b) se muestra la punta del soldador junto con los marcadores. La punta contiene 8 marcadores distintos.



(a) (b)

Figura 5.17: (a) Unión en T (b) Punta MIG

Se ha creado una estructura de datos que contiene la información de cada marcador: la posición en coordenadas relativas a la pieza y su codificación. Los marcadores están codificados como una matriz 5x5, el blanco se codifica como uno y el negro como cero. El método utilizado para la detección de los marcadores es el explicado en la sección 5.1.1

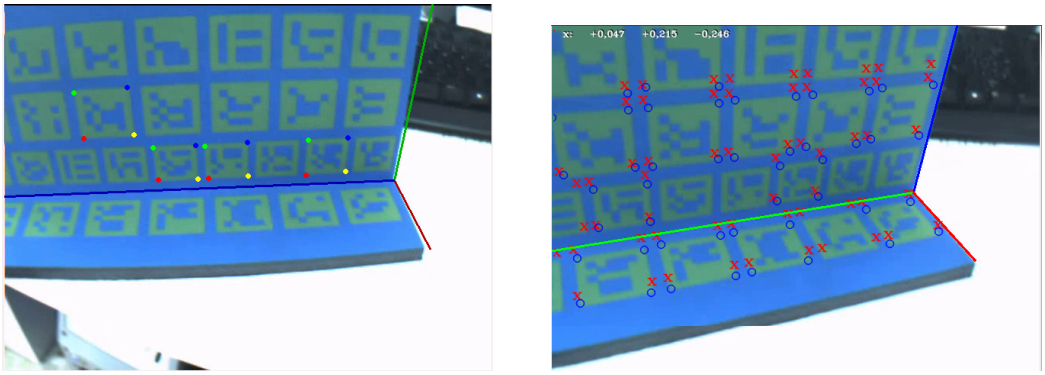
Se han realizado dos prototipos de autolocalización, uno basado en DLT y otro en monoSLAM. Con estos experimentos se pretende mostrar la mejora que supone utilizar un método como monoSLAM para la estimación de la posición y orientación de la cámara sobre el método con DLT.

Para la estimación de la posición y orientación se usan las implementaciones de DLT y monoSLAM que están dentro del componente VisualSLAM. El algoritmo de monoSLAM se

ha modificado para que los puntos que utiliza para la estimación de la posición y orientación sean sólo las esquinas de los marcadores.

La validación experimental, si funciona o no correctamente, se mide al pintar sobre las imágenes de entrada los ejes de coordenadas que se veían desde la posición estimada de la cámara y ver si coinciden con los reales.

En la Figura 5.18(a) se muestra la estimación de los ejes XYZ de la pieza, correspondiente a la estimación del algoritmo basado en DLT, junto a las esquinas de los marcadores detectados. Cada esquina aparece de un color diferente (rojo, verde, azul y amarillo). En la Figura 5.18(b) se muestra la estimación de los ejes XYZ utilizando monoSLAM. La X roja simboliza la estimación del filtro de Kalman y los círculos azules representan la detección de las esquinas del marcador para dicho fotograma.



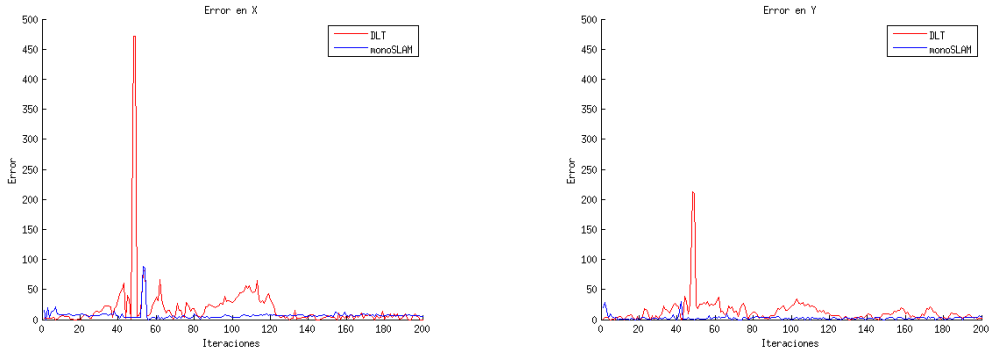
(a) (b)

Figura 5.18: Estimación e la posición y orientación (a) DLT (b) monoSLAM

Si se compara de manera cualitativa ambas imágenes se observa que la estimación basada en DLT tiene más error y se ajusta peor a la pieza que la estimación basada en monoSLAM, donde los ejes XYZ encajan mejor con la pieza.

También se ha realizado un análisis cuantitativo de la precisión de cada método. Para ello ha sido necesario etiquetar a mano la posición (0,0,0), es decir, el origen de coordenadas de la pieza. Se han etiquetado los primeros 200 fotogramas el vídeo. Este proceso es necesario ya que no se conoce la verdad absoluta de la posición de la cámara.

En la Figuras 5.19(a) y 5.19(b) se representa el error en píxeles de la imagen, en vertical y horizontal respectivamente, de ambos métodos respecto al etiquetado del vídeo realizado a mano. Como se puede comprobar, el método basado en DLT (línea roja) tiene más error en píxeles que el método basado en monoSLAM (línea azul). Incluso contiene errores mayores de 50 píxeles. Estos errores tan grandes se deben a que el método basado en DLT



(a) (b)

Figura 5.19: Error en píxeles del origen de coordenadas estimado (a) en X (b) en Y

proporciona una mejor estimación cuando tiene marcadores en los dos planos de la pieza que cuando los tiene sólo en uno de ellos. Sin embargo, la solución basada en monoSLAM tiene mucho menor error y es bastante más suave en comparación con DLT.

Lo anteriormente descrito mediante las gráficas se puede comprobar de manera numérica en la Tabla 5.1. El método basado en DLT tiene más del doble de error en horizontal que monoSLAM y en el vertical el error es 6 veces mayor. El error medio para el método basado en monoSLAM es de 8 píxeles mientras que el error para DLT es de 24 píxeles.

Método	Error en X	Error en Y	Error medio
monoSLAM	7.115 px	2.6 px	8.0573 px
DLT	19.195 px	13.07 px	24.8125 px

Cuadro 5.1: Error en píxeles de los distintos métodos utilizados

Con estos experimentos se ha comprabado que una solución basada en monoSLAM mejoraría la estimación de la posición de cámara y ajustaría mejor la Realidad Aumentada frente a la solución de DLT.

5.3. Videojuego con Realidad Aumentada

En esta sección se explica el videojuego desarrollado que desmuestra la utilidad de las técnicas de autolocalización visual para realizar una aplicación de Realidad Aumentada. La escena ficticia contiene un personaje controlado por el usuario humano y figuras 3D que

representan a los enemigos. Se puede mover la cámara libremente por el espacio y desde allí se observa la escena de modo natural.

5.3.1. Diseño

La arquitectura de JdeRobot permite un diseño en componentes distribuidos. En la Figura 5.20 se puede ver el diagrama de componentes de videojuego los tres componentes se ejecutan concurrentemente y funcionan en tiempo real.

Para el desarrollo de esta aplicación se ha hecho uso de un componente existente en *JdeRobot* llamado *cameraserver*. Este componente ofrece una interfaz ICE estandar que permite obtener el flujo de imágenes de una cámara. El segundo componente, *VisualSLAM*, se conecta al componente *cameraserver* y resuelve el problema de la localización visual en tiempo real utilizando para ello el algoritmo que se seleccione (MonoSLAM, PTAM o DLT, explicado en la sección 5.1), ofrece su resultado continuamente a través de una interfaz ICE estandar, llamada Pose3D, que contiene la posición y orientación actuales de la cámara.

Por último, el componente *AugmentedRealityGame* se conecta a *cameraserver* para obtener las imágenes de la cámara, estas imágenes se utilizan para representar los objetos de la Realidad Aumentada sobre la imagen, y también se conecta al interfaz *Pose3D* que ofrece *VisualSLAM*, para conocer la posición y orientación actuales de la cámara.

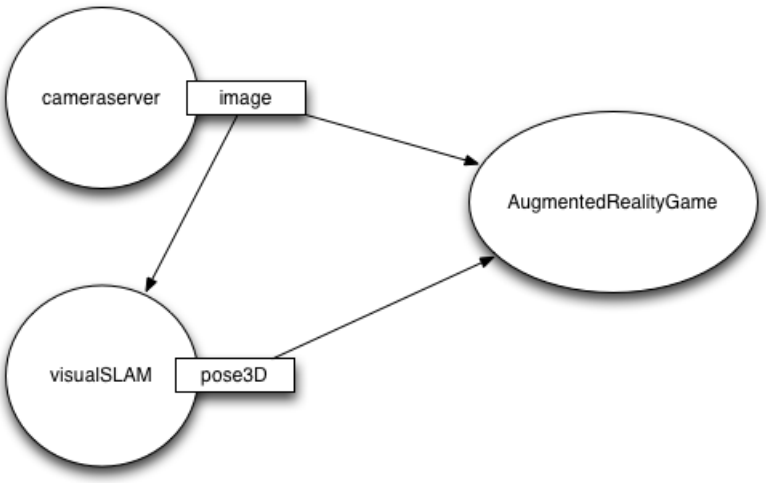


Figura 5.20: Diagrama de componentes del videojuego

En la Figura 5.21 se puede observar el componente *AugmentedRealityGame* como una caja blanca con sus bloques principales. Cuenta con dos hilos que comparten memoria

utilizando exclusión mutua. Uno de los hilos se ocupa de materializar la lógica del juego y, por otro lado, el segundo hilo se encargar de enriquecer las imágenes y *renderizar* las imágenes en pantalla.

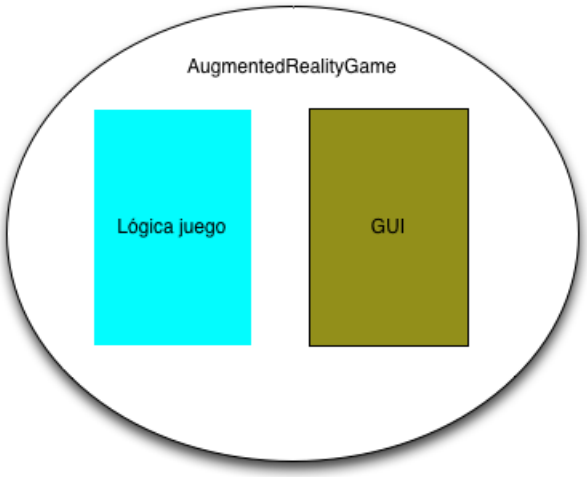


Figura 5.21: Componente *AugmentedRealityGame* como caja blanca

El videojuego desarrollado para este experimento es un juego 3D de acción, donde se controla a un mago con la capacidad de lanzar hechizos que tendrá que defenderse de los enemigos invasores (troles y demonios). El videojuego está desarrollado en una serie de clases. A continuación se detallan las clases más importantes:

- La clase *EstadoJuego* se encarga de gestionar el estado del juego que se encarga de las apariciones de enemigos, el jugador, los enemigos que actúan durante un momento determinado y los hechizos lanzados. Se encarga de gestionar y *renderizar* la escena. Por último, gestiona el teclado como dispositivo de entrada para que el usuario humano mueva a su personaje.

- La clase *AparicionEnemigos* modela la aparición de un enemigo a lo largo del desarrollo de una partida. De cada aparición interesa saber el tipo de enemigo al que se refiere, su posición y orientación iniciales.

- La clase *Actor* engloba de forma genérica a los elementos dinámicos y activos del juego como son los enemigos y el jugador. Contiene los atributos de nivel de vida, poder, velocidad, aceleracion y velocidad angular y lineal máxima.

- La clase *Jugador* hereda de *Actor* y representa al personaje protagonista. Aporta su energía mágica actual, el hechizo seleccionado, su posición anterior y el tiempo de

recuperación de la energía mágica o maná. En la Figura 5.22 se puede observar el personaje principal del videojuego.

Figura 5.22: Personaje principal

Tanto el personaje principal como los enemigos cuando se agota su vida desaparecen de la escena. Su vida se ve reducida cada vez que reciben un hechizo o ataque.

- La clase *Enemigo* cuenta con su tipo concreto. Hay tres tipos: *goblin*, diablillo o gólem de hielo. En la Figura 5.3.1 se pueden observar los enemigos del videojuego.
- La clase *Hechizo* representa los hechizos o proyectiles mágicos del juego que lanza el protagonista. De cada hechizo interesa saber su tipo (Bola de fuego, Gea o Ventisca), la dirección hacia la que se dirige, su nombre, descripción, poder, coste en maná, velocidad, tiempo que dura su explosión.

(a) (b) (c)

Figura 5.23: Enemigos del videojuego

Los enemigos no pueden ver por sí mismos y no son capaces de buscar al personaje principal para atacarle. Es necesario un sistema de búsqueda de caminos que guíe a los enemigos por el escenario. Como no se dispone de obstáculos en el escenario, el camino a realizar será una recta que une el personaje con el enemigo.

Una vez conocido el camino a realizar los enemigos conocen dónde está el personaje y ya pueden desplazarse por el escenario. Pero aún es necesario realizar el movimiento de seguir la línea. Para ello se hace uso de dos clases *Kinematic* y *Steering*. La clase *Kinematic* almacena propiedades del objeto que se tiene que mover: posición, orientación, velocidad lineal, velocidad angular y velocidad máxima. El movimiento del personaje lo modela la clase *Steering*, y funciona utilizando una fuerza vectorial que se transformará en una velocidad lineal para afectar a la velocidad del personaje y una velocidad angular para la rotación.

Cada personaje puede tener cincos estados diferentes: cuando no realiza ninguna acción y está a la espera de algún evento (IDLE), si está recibiendo algún daño (DAMAGE), cuando está atacando (ATTACK), cuando se está moviendo (RUN) o cuando su vida ha sido reducida a cero (ERASE). Estos estados están controlados con una máquina de estados, donde se irá cambiando de estado según los eventos provocados por el usuario y los enemigos.

5.3.2. Realidad Aumentada con OpenGL

Para la creación de gráficos en 3D, OpenGL implementa un *pipeline* gráfico integrado que sufre distintas transformaciones. Cada vértice pasa por este *pipeline*:

- **Tranformación de Modelado**: En la que los modelos se posicionan en la escena y se obtienen las coordenadas absolutas.
- **Transformación de Visualización**: Donde se especifica la posición de la cámara y se mueven los objetos desde las coordenadas del mundo a las coordenadas visualización o coordenadas de la cámara.
- **Transformación de Proyección**: Obteniendo coordenadas normalizadas en el cubo unitario.
- **Transformación de Recorte y de Pantalla**: Donde se obtienen tras el recorte (o *clipping* de la geometría), las coordenadas 2D en la ventana en pantalla.

En la Figura 5.24 se puede observar la relación que existe entre la cámara, el sistema de referencia y los objetos en la escena. La escena contiene una serie de objetos y éstos poseen una serie de coordenadas respecto al sistema de referencia. La cámara es desde donde se observa la escena, por lo tanto, su posición y orientación respecto al sistema de referencia tiene que ser especificado.

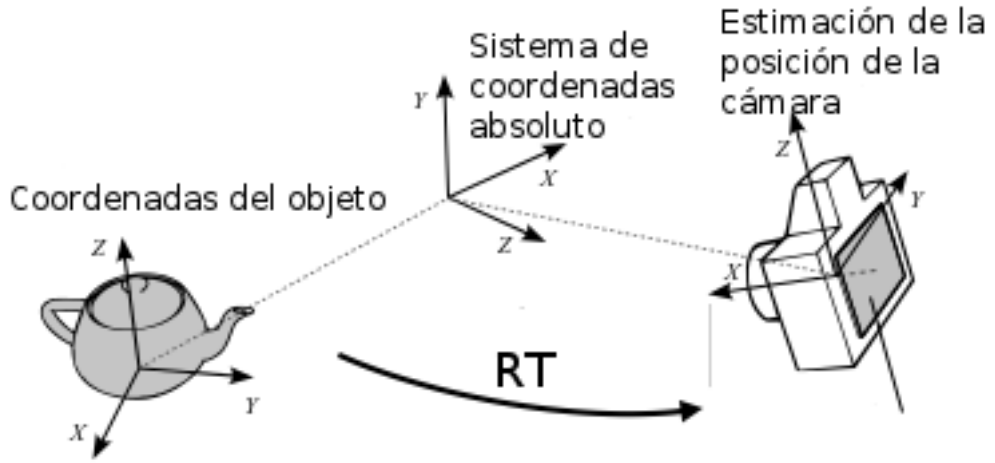


Figura 5.24: Rotación y translación entre sistema de referencia y cámara

OpenGL combina la Transformación de Modelado y la de Visualización en una tranformación llamada *Modelview*. De este modo OpenGL transforma directamente las coordenadas absolutas a coordenadas de visualización empleando la matriz *Modelview*.

La Transformación de Visualización debe especificarse antes que ninguna otra transformación de modelado. Esto es debido a que OpenGL aplica las transformaciones en orden inverso. De este modo, aplicando en el código las transformaciones de visualización antes que las de modelado nos aseguramos que ocurrirán después.

Para comenzar con la definición de la Transformación de Visualización, es necesario limpiar la matriz de trabajo actual. OpenGL cuenta con una función que carga la matriz identidad *glLoadIdentity()*. Una vez hecho esto, se puede posicionar la cámara virtual de varias formas:

- **glulookat**: Es la opción más utilizada, esta función recibe como parámetros un punto (*eye*) y dos vectores (*center y up*). El punto *eye* es el punto donde se encuentra la cámara en el espacio 3D y mediante los vectores libres *center* y *up* se orienta hacia donde mira la cámara.

- **Translación y rotación**: Otra opción es especificar manualmente una secuencia de translaciones y rotaciones para posicionar la cámara (mediante las funciones *glTraslate* y *glRotate*).

- **Carga de matriz**: Cargar la matriz de visualización calculada externamente, mediante la llamada a la función *glLoadMatrix*. En concreto es la matriz RT calculada por los algoritmo de autolocalización.

El último método es el más utilizado en aplicaciones de Realidad Aumentada, cargando directamente la matriz calculada con los métodos de autolocalización visual.

Las transformaciones de modelado nos permiten modificar los objetos de la escena. Existen tres operaciones básicas de modelado que implementa OpenGL con llamadas a funciones. No obstante se puede especificar cualquier operación aplicando una matriz definida por el usuario.

- **Translación**: El objeto se mueve a lo largo de un vector. Esta operación se realiza mediante la llamada *glTranslate(x,y,z)*.
- **Rotación** El objeto rota alrededor del eje definido por un vector. Esta operación se realiza mediante la llamada *glRotate(β, x, y,z)*, siendo β el ángulo de rotación en grados en sentido contrario a las agujas del reloj.
- **Escalado**: El objeto se escala un determinado valor en cada eje. Se realiza mediante *glScale(x, y, z)*.

Las transformaciones de proyección definen el volumen de visualización y los planos de recorte. OpenGL soporta modelos básicos de proyección: la proyección ortográfica y la proyección en perpectiva.

El núcleo de OpenGL define una función para definir la pirámide de visualización (o *frustrum*) mediante la llamada *glFrustrum()*, que requiere 6 parámetros. Otro modo de especificar la transformación es mediante la función *gluPerpective(fov, aspect, near, far*). En esta función *far* y *near* son las distancias de los planos de recorte (igual que los parámetros f y n del *frustrum*). La variable *fov* especifica en grados el ángulo en el eje Y de la escena que es visible para el usuario y la variable *aspect* indica la relación de aspecto de la pantalla (ancho/alto).

En la Figura 5.25 se puede observar el modelo de cámara que proporciona OpenGL. *fov* y *aspect* representan los parámetros intrínsecos de la cámara. El parámetro *Znear* y *Zfar* representa lo que está dentro o fuera de la imagen (si esta demasiado cerca o demasiado lejos no se proyecta en la imagen sintética de la cámara de visualización).

OpenGL internamente maneja pilas de matrices de forma que únicamente la matriz de la cima de cada pila es la que se está utilizando en un momento determinado. Hay cuatro pilas de matrices:

- **Pila *modelview*** (GL_MODELVIEW). Esta pila contiene las matrices de transformación de modelado y visualización.

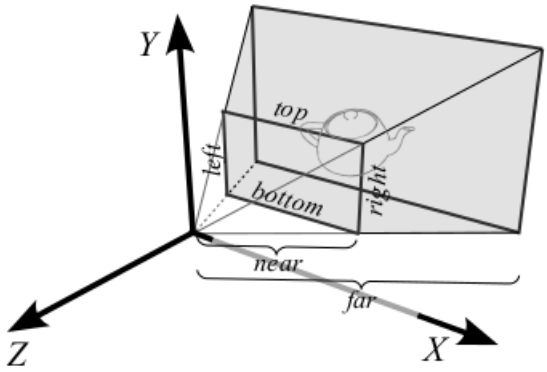


Figura 5.25: Modelo de cámara en OpenGL

- **Pila *Projection*** (GL_PROJECTION). Esta pila contiene las matrices de proyección.
- **Pila *Color*** (GL_COLOR) Utilizada para modificar los colores.
- **Pila *Texture*** (GL_TEXTURE) Estas matrices se emplean para transformar las coordenadas de textura.

Como se ha explicado, es posible cargar matrices de transformación definidas por métodos propios (como es el caso de los algoritmo de autolocalización de cámaras). Esto se realiza con la llamada a función *glLoadMatrix()*. Cuando se llama a esta función se modifica la cima de la pila de matrices activa con el contenido de la matriz que se pasa como argumento. Si interesa multiplicar una matriz definida por el contenido de la cima de la pila de matrices se puede llamar a la función *glMultMatrix()* que postmultiplica la matriz que pasamos como argumento por la matriz de la cima de la pila.

Los algoritmos de autolocalización de cámaras calculan la posición de la cámara respecto al mundo que le rodea, es decir, una ubicación en el espacio (correspondiente a la distancia y rotación relativa entre los objetos o patrones y la cámara real). Por lo tanto una vez conocidos los parámetros de la cámara, tanto intrínsecos como extrínsecos, se expecifica a OpenGL cuáles son la matrices de visualización y proyección. A continuación se muestra un fragmento de código donde se cargan dichas matrices.

```
glMatrixMode(GL_PROJECTION);
GLfloat* projectionGL = convertMatrixType(camera->getProjection());
```

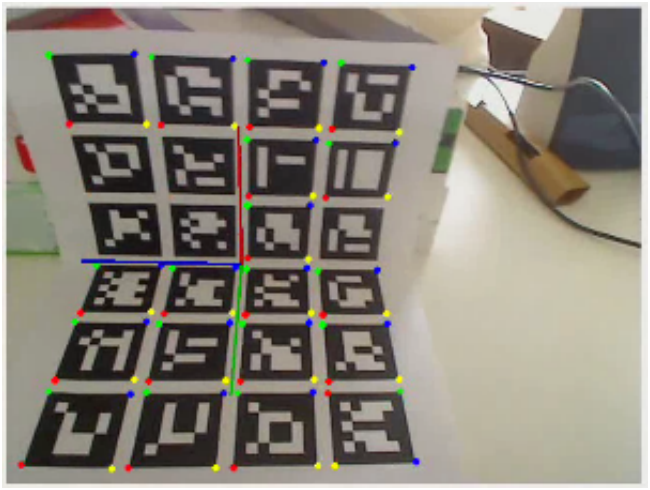


Figura 5.26: Realidad Aumentada sobre los patrones

```
glLoadMatrixf(projectionGL);
delete[] projectionGL;

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
GLfloat* modelviewGL = convertMatrixType(camera->getModelview());
glLoadMatrixf(modelviewGL);
delete[] modelviewGL;
```

5.3.3. Motor gráfico OGRE

En la aplicación del videojuego se ha utilizado el motor gráfico OGRE. Este motor gráfico permite manipular y tratar los gráficos 3D. Ogre permite representar los personajes del juego sobre la imagen real. Para renderizar un objeto con Ogre es necesario conocer su malla y en caso de tener movimiento de algún tipo es necesario el esqueleto. Estos archivos suelen tener extensiones .mesh, para las mallas, y .skeleton, para los esqueletos 3D.

Para cada archivo almacenado hay que describir la estructura de elementos que permiten la manipulación de las escenas Ogre. Existen tres elementos fundamentales: *Entity*, *SceneNode* y *SceneManager*.

- *SceneManager* o controlador de nodos: es el nodo raíz de todo lo que se dibuja en la escena, a través de éste siempre es posible acceder a cualquier entidad creada en la escena.

- *Entity*: corresponde a cada elemento que se puede dibujar en la pantalla. Cualquier modelo, malla u objeto que se cargue es considerado una entidad; sin embargo no son entidades las cámaras o las luces. La posición y la orientación geométrica de las entidades no son una característica de las mismas, estos parámetros son controlados por otra estructura.

- *SceneNode*: se encarga de manipular las características de las entidades. Se trata de un contenedor que engloba las entidades, cámaras y luces para modificar sus propiedades. La posición siempre es relativa a la de los nodos padre.

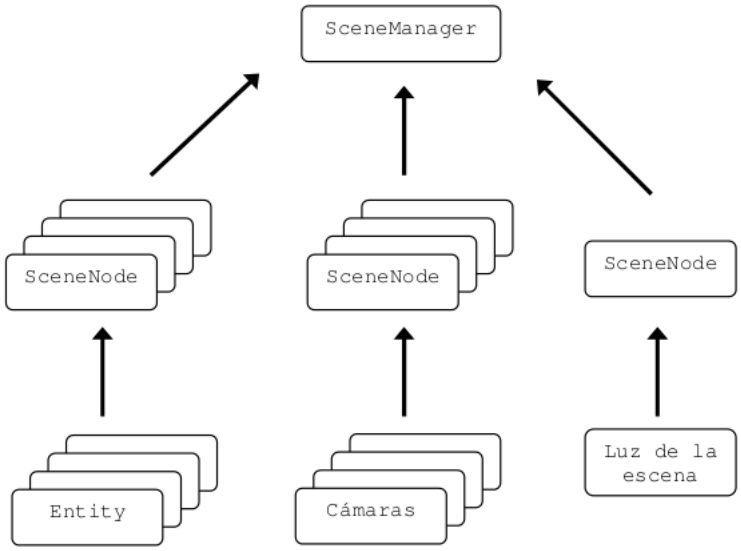


Figura 5.27: Jerarquía de nodos de la escena

En la Figura 5.27 se muestra la jerarquía de nodos de la escena explicada anteriormente. Para posicionar la cámara y *renderizar* los objetos de manera coherente en la imagen se debe modificar el *SceneNode* e indicar la posición y orientación de la cámara (obtenida mediante los algoritmos de autolocalización visual) desde la cual se contempla la escena ficticia. Ogre necesita que esta información se le indique utilizando dos funciones: *setPosition()* que acepta como entrada tres reales que significan la posición en x, y, z en un eje de coordenadas cartesiano; para indicar la orientación se utiliza la función *setOrientation()* que acepta como parámetros cuatro reales correspondiente a un cuaternión. Estas dos funciones será necesario llamarlas en cada iteración.

Ogre permite la captura de todo tipo de eventos: eventos de ratón, de teclado o incluso *joysticks*. Para este experimento se han utilizado cinco teclas para generar eventos: letras,

A, *W*, *S* y *D* que mueven al personaje y la tecla *espacio* que lanza un hechizo.

5.3.4. Integración de OpenCV y Ogre

Para el funcionamiento de este componente ha sido necesaria la integración entre OpenCV y Ogre. Por un lado, OpenCV se utiliza para transformar las imágenes recibidas por la interfaz estandar ICE a la estructura de datos de OpenCV, *cv::Mat*, una extensión para trabajar con matrices de cualquier tamaño y tipo en C++, que permiten acceder cómodamente a los datos de la imagen. Y por otro lado, se muestran las imágenes recibidas en el fondo de una ventana de Ogre

Ha sido necesario crear un plano sobre el que se actualiza el video obtenido de la cámara. El fotograma se muestra como una textura dinámica. Además de desactivar las propidades *depthbuffer* o la iluminación, es importante indicar que el buffer asociado a este material va a ser actualizado con frecuencia, cada vez que se reciba un nuevo fotograma. Posteriormente se crea un rectángulo 2D, que se añadirá al grupo de dibujo. La actualización de la textura asociada al plano se realiza a través de un puntero compartido, protegido mediante exclusión mutua, *pixelbuffer*, especificando el color de cada píxel en valores RGBA.

5.3.5. Videojuego

En la Figura 5.28 se puede observar el diagrama de flujo del sistema de Realidad Aumentada materializado para esta aplicación. El componente VisualSLAM recibe las imágenes de la cámara y proporciona la estimación de la posición y orientación de la cámara. Ésta información es introducida en los objetos de la escena para enriquecer la imagen. Si la RT se cambia, es decir, se mueve la cámara, entonces se observa la escena desde otro ángulo o perpectiva. Esto hace que en la imagen enriquecida, la parte inventada sea coherente con la imagen real a la que se sobrepone. Si la estimación de la RT falla encontes aparecen inconsistencias.

En la Figura 5.26 se puede observar el reconocimiento de patrones explicado en el punto anterior y la Realidad Aumentada que se proyecta sobre la imagen utilizando OpenGL. En esta ocasión se representan los ejes XYZ aumentando la imagen real. En la Figura 5.29 la cámara ésta observando una mesa y sobre ella aparecen los objetos de la Realidad Aumentada. Gracias a la localización y la libreria Ogre se han proyectado los personajes animados 3D que desmuestran las capacidades de esta técnica. Para ver el videojuego en

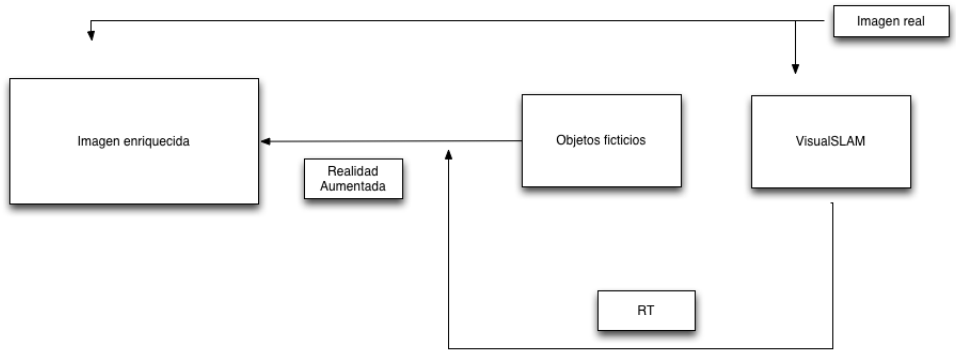


Figura 5.28: Diagrama de flujo para representar Realidad Aumentada

movimiento se puede consultar la wiki del proyecto [3]

Figura 5.29: Videojuego de Realidad Aumentada

5.4. Aplicación Android

Uno de los objetivos de este proyecto es hacer uso de la implementación de PTAM desarrollada en C++ en un dispositivo Android con el menor número posible de

[3]http://jderobot.org/index.php/Ahcorde-tfm#Augmented_reality_GAME_2B_PTAM_2B_OGRE_2B_Engine_video_game2B9

modificaciones al código fuente. Para ello ha sido necesario utilizar el SDK de Android para OpenCV, realizar un *wrapper* entre Java y C++ para intercambiar la información de las imágenes y la posición y orientación de la cámara. Incorporando los cambios en código del *Core 3D* y del procesamiento 2D del componente Visual SLAM (en C++) para que funcione en *Android*. Y por último, representar la Realidad Aumentada en Android.

En Android es necesario indicar en el *AndroidManifest* los permisos y recursos de los que va a hacer uso la aplicación para poder ejecutarse. En este caso se necesitan permisos para la cámara y es necesario indicar la versión de OpenGL, en este caso la 2.0. A continuación se muestran la líneas que lo indican:

```
1 <uses-permission android:name="android.permission.CAMERA"/>
2 <uses-feature android:required="true" android:glEsVersion="0x00020000"/>
```

En la Figura 5.30 se puede observar la estructura de la aplicación VisualSLAM en Android. Es necesario establecer una comunicación entre la máquina virtual de Java y el algoritmo de VisualSLAM (Procesamiento 2D + Core 3D). Ésto se ha realizado a través de un *wrapper* que además también se comunica con el API de OpenCV. También ha sido necesario comunicar la interfaz gráfica ahora en Java con el algoritmo de VisualSLAM (Procesamiento 2D + Core 3D), ya que será éste el que proporcione la información de posición y orientación que es necesaria para representar la Realidad Aumentada en la pantalla del dispositivo utilizando el API de OpenGL.

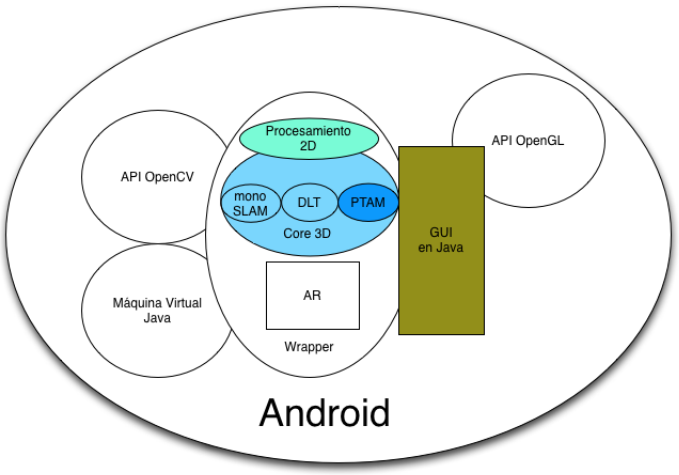


Figura 5.30: Diagrama de bloques Visual SLAM en Android

En primer lugar, es necesario obtener las imágenes de la cámara en una estructura de datos *cv::Mat* de OpenCV para ello se ha utilizado el SDK de OpenCV para Android.

Es necesario que la *actividad* implemente la interfaz *CvCameraViewListener2* y crear un *callback* que se carga al iniciar la aplicación del tipo *BaseLoaderCallback*, que hará que carguen las librerías dinámicas de OpenCV y que cada vez que se reciba un fotograma se ejecute la función *onCameraFrame()*.

5.4.1. Usando código C++ en Java

Como se ha explicado anteriormente, uno de los objetivos es utilizar el código de PTAM desarrollado en la sección 4.4 en C++ para evitar reimplementar todo el código en Java. Para cumplir este objetivo ha sido necesario realizar un *wrapper* que permite intercambiar información entre la máquina virtual de Java y el JNI. La clase *NativeWrapper* se encarga de hacer de intermediario. Esta clase contiene los médotos:

- **native_createTest()**: constructor del objeto C++ que mantendrá el estado del algoritmo guardando la dirección de memoria del puntero que apunta a la clase.
- **native_disposeTest(long cptr)**: se libera la memoria correspondiente a la dirección de memoria que se le indica por parámetro.
- **native_updateImage()**: este método entrega a JNI las direcciones de memoria de los punteros que apuntan a la imagen en color y a la imagen en blanco y negro.
- **native_getTranslation(long cptr, long t)**: método que devuelve la translación calculada por PTAM.
- **native_getRotation(long cptr, long rot)**: método que devuelve la orientación calculada por PTAM.
- **native_touchScreen(long cptr)**: método que indica si la pantalla ha sido pulsada por el usuario. Esa función contiene un temporizador para evitar recibir dos pulsaciones muy próximas en el tiempo.

Una de las principales ventajas de utilizar el *JNI* y por lo tanto código en C++ es la posibilidad de utilizar *exactamente* el mismo código implementado para la aplicación de escritorio en un dispositivo Android. Pero para ello es necesario compilar el código utilizando *Android NDK*, que generará una librería dinámica. A continuación se describen los pasos realizados para compilar el código. Primero es necesario crear una carpeta con el nombre *jni* donde se incluirán todos los ficheros en C++ y dos ficheros *Android.mk* y

Application.mk. Estos dos últimos son los ficheros *Makefile* que controlan la construcción de la librería.

Mientras que *Application.mk* indica los recursos nativos que van a ser utilizados por la aplicación, el *Android.mk* describe qué ficheros son los que se necesitan compilar. El *Application.mk* consta de los siguientes campos:

- **APP_STL**: Indica qué implementación de la librería estandar de C++ es usada.
- **APP_ABI**: Indica el tipo de arquitectura para la cual se compilarán los archivos.
- **APP_CPPFLAGS**: Indica las directivas que se van a pasar al compilador de C++.

Por lo tanto, el fichero *Application.mk* queda como:

```
1 APP_STL := gnustl_static
2 APP_CPPFLAGS := -frtti -fexceptions
3 APP_ABI := armeabi-v7a
4 APP_PLATFORM := android-8
```

Y para el fichero *Android.mk* los campos más importantes son:

- **LOCAL_PATH**: Indica la localización de los archivos.
- **LOCAL_MODULE**: Nombre que le damos a la librería dinámica que se crea en el proceso de compilación
- **LOCAL_SRC_FILES**: Ficheros con el código nativo a compilar.
- **LOCAL_LDLIBS**: Librerías auxiliares.

El fichero *Android.mk* queda como:

```
1 include $(CLEAR_VARS)
2 #se incluyen las dependencias de Android
3 include /Users/ahcorde/programas/OpenCV-2.4.7.1-android-sdk/sdk/native/jni/
    OpenCV.mk
4
5 #nombre
6 LOCAL_MODULE    := ptam_mixed
7 #nombre de los ficheros
```

```
8 LOCAL_SRC_FILES := jni_part.cpp Tracker.ccp KeyFrame.ccp ...
9 #flags para la optimizar
10 LOCAL_LDLIBS += -llog -ldl -O3 -march=nocona -msse3
11 include $(BUILD_SHARED_LIBRARY)
12
13 #se incluyen las cabeceras de Eigen
14 LOCAL_C_INCLUDES += /usr/local/include/eigen3/
```

El fichero que contiene el código en C++ y que forma parte del *wrapper* deben tener un nombre con una serie de características:

- Todos los métodos deben empezar por la palaba *Java*.
- Deben contener el nombre del paquete del proyecto Java (*vision_ptam_source*).
- Nombre de la clase Java desde la que se invoca el código (*NativeWrapper*).
- Nombre el método que se implementa.

Todas las funciones que formen parte del *wrapper* deben contener dos parámetros *JNIENV* env* y *jobject thiz*. El primero es un puntero a una tabla que contiene la interfaz hacia la máquina virtual, incluye las funciones necesarias para interactuar con la máquina virtual de Java y para trabajar con los objetos Java. El segundo parámetro actúa como una referencia *this* al objeto Java en el que se invoca el método. El método que actualiza la imágenes queda de la siguiente forma:

```
1
2 JNIEXPORT void JNICALL
      Java_vision_ptam_source_NativeWrapper_native_1updateImage(JNIEnv *,
      jobject, jlong cptr, jlong addrRgba, jlong addrGRAY){
3     / se realiza los casting necesarios
4     cv::Mat& mRgb = *(cv::Mat*)addrRgba;
5     cv::Mat& mgray = *(cv::Mat*)addrGRAY;
6     PTAM *ptam=reinterpret_cast<PTAM*>(cptr);
7     // se actualiza la imagen
8     ptam->update(mRgb, mgray);
9   }
```

Una vez tenemos los ficheros *Android.mk* y *Application.mk* bien configurados se procede al proceso de compilación. Para realizar esta tarea se ha utilizado el IDE Eclipse, introducido en la sección 3.9.

5.4.2. GUI en Java

En cuanto a la interfaz gráfica es necesario crear una superficie especial en Android en la que se van a dibujar los gráficos denominada *GLSurfaceView*. Hay que añadir un *renderer* que se encargue del dibujar y que cree un contexto de *OpenGL ES*. El *renderer* tiene que implementar la interfaz *GLSurfaceView.Renderer*. Éste será el encargado de dibujar sobre el *GLSurfaceView*. Contiene métodos para dibujar y se debe implementar al menos los siguientes tres métodos:

```
1 //Se ejecuta al crearse la superficie. Establece el entorno de OpenGL
2 public void onSurfaceCreated(GL10 unused, EGLConfig config)
3 //Dibuja sobre la superficie
4 public void onDrawFrame(GL10 unused)
5 //Sucede al cambiar la forma de la superficie
6 public void onSurfaceChanged(GL10 unused, int width, int height)
```

Dentro de la interfaz gráfica de Android es necesario indicar que la superficie donde se van a representar los objetos con *OpenGL* sea translúcida. A continuación se muestra un fragmento de código que realiza lo anteriormente descrito.

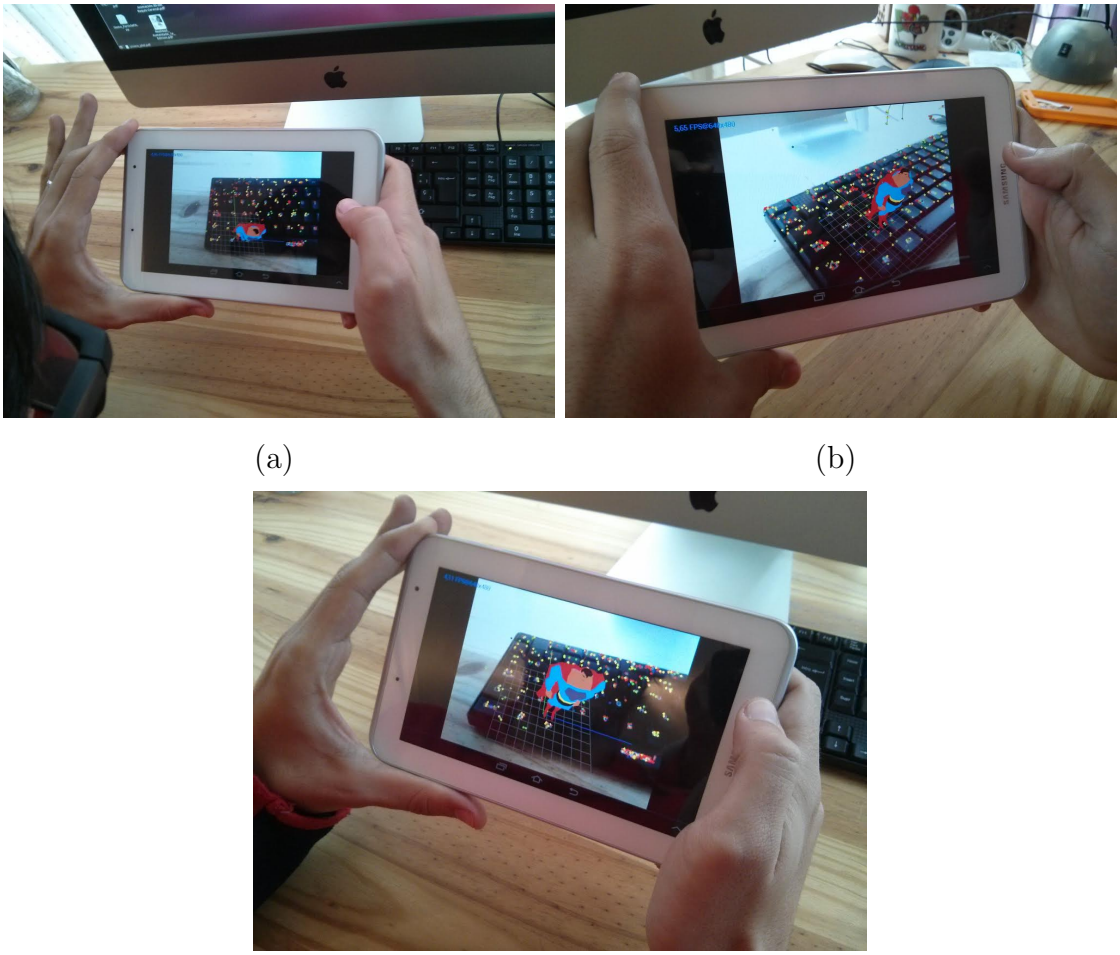
```
1 addContentView(this.mGLSurfaceView,
2 new LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT));
3 mGLSurfaceView.setZOrderMediaOverlay(true);
4 this.mGLSurfaceView.setRenderer(this.renderer);
5 mGLSurfaceView.getHolder().setFormat(PixelFormat.TRANSLUCENT);
```

Los datos de los vértices, color o vértices, del modelo se guardan en *arrays*. Para pasar esta información a *OpenGL*, se hace a través de buffers para maximizar la eficiencia. A continuación se muestra un fragmento de código:

```
1 //Asignar al buffer un tam correspondiente al vector por
2 //la cantidad
3 cbb = ByteBuffer.allocateDirect(vertexData.length * 4);
4 //indicarle que use el orden de bytes que utilice el
5 //dispositivo
6 cbb.order(ByteOrder.nativeOrder());
7 vertexBuffer = cbb.asFloatBuffer();
8 vertexBuffer.put(vertexData).position(0);
```

5.4.3. Experimentos con VisualSLAM en Android

En la Figura 5.31 se muestra el algoritmo PTAM funcionando en un dispositivo Android desde diferentes puntos de vista. La tableta utilizada contiene un procesador Dual-Core a 1GHz y dos cámaras integradas. La cantidad de fotogramas por segundo que es capaz de procesar es aproximadamente *6-7fps*. Teniendo en cuenta que la cantidad de fotogramas por segundo que se pueden mostrar por pantalla sin procesar la imagen es de *14-15fps*, es un número razonable teniendo en cuenta la capacidad de cómputo de la tableta. La estimación de la posición y orientación es correcta pero los movimientos deben ser más lentos.



(a) (b)

(c)

Figura 5.31: PTAM en un dispositivo Android

Capítulo 6

Conclusiones

En los capítulos anteriores se ha descrito la solución propuesta y una serie de experimentos que validan el sistema desarrollado. En este último capítulo se recapitulan las principales conclusiones obtenidas de este Trabajo Fin de Máster y las posibles líneas futuras en las que se puede seguir investigando. Con él que se cierra la presente memoria.

6.1. Conclusiones

Se han diseñado y programado el componente VisualSLAM que incluye algoritmos (DLT y PTAM entre ellos) que estiman en tiempo real la posición y orientación de una única cámara móvil, utilizando exclusivamente las imágenes obtenidas por la cámara. El componente Visual SLAM y los algoritmos programados se han validado experimentalmente haciendo uso de una cámara de videoconferencia convencional y un dispositivo Android. También se ha realizado un videojuego que hace uso de la localización de la cámara para representar los personajes del videojuego coherentes con la escena.

A continuación se revisan los subobjetivos planteados en el capítulo 2 y en que medida se han conseguido cada uno de ellos.

- El primero de los objetivos consistía en el desarrollo y prueba de distintos algoritmos para la autolocalización visual. En el capítulo 4 se exponen los fundamentos teóricos de los tres algoritmos utilizados (DLT, PTAM y monoSLAM). Se estudiaron los fundamentos teóricos de varias soluciones propuestas en la literatura. La respuesta a éste subobjetivo es el diseño y programación en C++ del componente Visual SLAM explicado en la sección 5.1. En el caso de DLT el problema se resuelve detectando los

marcadores en la imagen y resolviendo las ecuaciones utilizando puntos conocidos. El algoritmo de monoSLAM resuelve el problema mediante un filtro extendido de Kalman. Y por último, PTAM utiliza un algoritmo de optimización no lineal y ajuste de haces para obtener la posición y orientación de la cámara. Ha sido necesario el apoyo de funciones de la librería OpenCV (para el tratamiento de la imágenes) y Eigen (para simplificar los cálculos del álgebra lineal). Estos algoritmos tienen entidad propia y actualmente están siendo utilizados en aplicaciones de visión artificial del grupo de Robótica de la URJC en Proyectos Fin de Carrera y en Tesis Doctorales.

- El segundo subobjetivo consistía en validar experimentalmente el sistema desarrollado. En concreto se trata de comprobar si el algoritmo es capaz de localizar la cámara en 3D a partir de las observaciones de la cámara en diferentes escenas. En el cápitulo 5 se describen los distintos experimentos realizados. En primer lugar, se ha comprobado que el sistema funciona y para ello se ha validado de forma cualitativa comprobando la trayectoria de la cámara, robustez frente a movimientos de la cámara, desenfoque en las imágenes y rosbustez frente a oclusiones temporales parciales o totales o zonas sin textura. Se comprobó que el algoritmo PTAM es más robusto que monoSLAM (explicado en la sección 5.2)

  En segundo lugar, se ha desarrollado un propotipo para la empresa Seabery que demuestra la mejora que supone utilizar un algoritmo como monoSLAM frente a DLT, tal y como se describió en la sección 5.2.4.

  Tercero, se eligió el algoritmo desarrollado en este trabajo para realizar un videojuego de Realidad Aumentada. De esta manera se ha demostrado que el sistema tiene una utilidad en sistemas avanzados de Realidad Aumentada en la que no es necesaria la presencia de patrónes gráficos de referencia ni un proceso de postproducción. Se aumenta la imagen con objetos ficticios, un personaje 3D y monstruos que se lanzan hechizos (explicado en la sección 5.3).

- El tercer el subobjetivo era integrar el algoritmo de PTAM en un dispositivo Android realizando el menor número de cambios posibles al código fuente en C++. Este objetivo se ha cumplido como se demuestra en la sección 5.4. El algoritmo es capaz de ejecutar en tiempo real (a menos *fps* que un ordenador convencional) y además muestra un modelo 3D estático, que se puede apreciar desde distintas vistas moviendo el dispositivo móvil por la escena 3D. El código esta compilado utilizado JNI. La interfaz de usuario esta desarrollado en Java y OpenGL para *Android*

Los requisitos que se exigen a las aplicaciones desarrolladas en este Trabajo Fin de

Máster están marcados por lo especificado en la sección 2.2. A continuación se analiza en qué medida se han satisfecho:

Todos los componentes han sido desarrollados usando la plataforma JdeRobot. La solución propuesta está programada utilizando C++ y Java. Es una colección de componentes que se ejecuta sobre JdeRobot y ofrece interfaces ICE para comunicarse con el resto de componentes. Trabajar con esta plataforma ha facilitado el desarrollo, el disponer de componentes y drivers ya existentes han sido de gran utilidad. También proporciona la infraestructura necesaria para programar aplicaciones de procesamiento en tiempo real.

Para el desarrollo de los componentes se han manejado múltiples librerías en el entorno Linux, como OpenCV para el procesamiento de la imagen, Eigen para las funciones algebraicas, OpenGL para la representación 3D o Qt para el desarrollo de las interfaces gráficas.

El requisito de tiempo real se ha cumplido ya que durante todo el desarrollo del proyecto se han realizado pruebas con el sistema funcionando a 25-30 fotogramas por segundo. Los experimentos realizados con el videojuego o la aplicación en Android validan este punto. En los dispositivos Android no se alcanza una tasa tan grande de fotogramas por segundo por las limitaciones hardware del sistema.

En cuanto a precisión de la localización depende directamente del número de puntos de apoyo y de la distancia de éstos a la cámara. La robuztez del algoritmo depende directamente de cómo esté estructurado el entorno en el que se trabaja. Escenas con poca textura no son favorables y se puede llegar a perder la localización. Esto se refleja en la secciones 5.2.2 y 5.2.3.

Respecto a los conocimientos aportados por el proyecto, se puede destacar los adquiridos sobre técnicas de localización de cámaras en tres dimensiones que suponen un gran reto y un campo candente de investigación.

Los objetivos se han satisfecho tras un periodo de trabajo que ha comprendido el aprendizaje de nuevas técnicas de procesado de señal, conceptos de geometría y desarrollo de software, así como un repaso de Álgebra Lineal y interfaces gráficas sofisticadas (OpenGL, OGRE...).

Dado que este Trabajo Fin de Máster es muy heterogeneo en cuanto a técnicas y herramientas auxiliares empleadas han sido necesarios muchas iteraciones, búsqueda de información, resolución de dudas, codificar, probar, depurar, documentar, etc. Este desarrollo ha sido llevadero gracias al modelo de desarrollo en espiral. También se ha profundizado en un gran abanico de utilidades de software libre: la plataforma JdeRobot,

la creación de una aplicación Android de visión utilizando JNI, además de un serie de librerías, aplicaciones y demás herramientas, importantes hoy día en cuanto a software se refiere. Mencionar especialmente el uso de las librerías OpenCV y Eigen que han sido el pilar básico para el desarrollo del trabajo.

Este proyecto sienta las bases para que robots del grupo puedan explorar entornos desconocidos y para indagar más en aplicaciones de Realidad Aumentada. Un factor importante es el uso de una única cámara como sensor para conocer la localización. Esta cámara se ha utilizado para extraer información 3D relevante de la imagen, algo que típicamente se realiza con cámaras estereo o más recientemente con sensores RGBD. Estas otras técnicas pueden servir como complementarias para robustecer el SLAM. Los algoritmos propuestos son aplicables a distintos tipos de cámaras, siempre que éstas se calibren correctamente.

Gracias al desarrollo de este proyecto he seguido formando parte del grupo de desarrolladores de la plataforma JdeRobot que comencé a formar parte desde mi Proyecto Fin de Carrera. Esto implica un trabajo en grupo y el uso de herramientas de control de versiones como *subversion* para la gestión de código. Esto ha sido necesario para la integración de los componentes desarrollados en el presente Trabajo Fin de Máster en el repositorio oficial de JdeRobot. Aportando unas 4000 líneas de código, los del componente *Visual SLAM*.

Este Trabajo Fin de Máster tiene asociado un mediawiki[1] que contiene una bitácora con los progresos conseguidos en hilo temporal y una escaparate con vídeos que muestran los resultados alcanzados. También se dispone de un SVN [2] donde se encuentra disponible el código fuente.

6.2. Trabajos futuros

Algunas posibles mejoras que se pueden realizar sobre este trabajo y que pueden formar parte de nuevas líneas de investigación para otros proyectos relacionados son los siguientes:

- **Inicialización no retrasada**: Como se ha explicado en la sección 4.4, PTAM necesita inicializarse realizando un cierto movimiento 3D con la cámara. Sin embargo, monoSLAM no necesita este paso, simplificando su uso. Esta diferencia es debida a la

[1] http://jderobot.org/index.php/Ahcorde-tfm
[2] https://svn.jderobot.org/users/ahcorde/tfm

distinta naturaleza de las soluciones propuestas. Puede resultar interesante resolver este problema en PTAM.

- **Optimización del código para plataformas móviles**: Debido al tipo de optimización que utiliza PTAM para resolver el problema de localización, ésto supone un gran consumo de CPU. En un ordenador de sobremesa no es un gran problema, sin embargo, en las plataformas móviles donde el uso de la CPU es crítico para el ahorro de la batería puede ser interesante optimizarlo o simplificarlo para mejorar el rendimiento.

- **Integración de colisiones con los objetos reales**: El videojuego desarrollado en los experimentos no tiene en cuenta colisiones con los puntos del mapa 3D que genera el algoritmo de VisualSLAM. Una vía a mejorar puede ser la integración del mapa generado dentro del videojuego y provocar colisiones de los personajes ficticios con objetos y superficies en el mapa generado desde los datos reales. Utilizando un simulador de físicas, como Bullet, permitiría por ejemplo que los personajes del videojuego se puedan caer si traspasan el borde de un mesa.

- **Nuevas aplicaciones**: Haciendo uso de la aplicación VisualSLAM crear nuevas aplicaciones como las descritas en el Capítulo 1. Por ejemplo, Magic Plan sustituyendo los giróscopos por autolocalización visual que permita translaciones y movimientos mayores.

- **Uso de algoritmos más recientes**: Actualmente existen en la literatura científica algoritmos más complejos, como por ejemplo DTAM, que no sólo proporciona la localización de la cámara y un mapa de puntos, si no que también es capaz de realizar un mallado de la escena, mejorando la interacción del mundo real con el virtual.

Anexo

6.3. Filtro de Kalman

El filtro de Kalman (KF) es un algoritmo desarrollado por Rudolf E. Kalman [Kalman, 1960], que permite estimar el estado de un sistema dinámico a partir de observaciones que pueden contener ruido. El KF tiene numerosas aplicaciones en ingeniería, tales como seguimiento, control de vehículos, procesamiento de señales o econometría.

El KF es un estimador probabilístico bayesiano óptimo, y se puede reformular iterativamente de un modo sencillo en sistemas lineales gaussianos. Se formula en dos pasos, un primer paso de predicción y otro de corrección. Para que el KF pueda aplicarse, debemos encontrarnos ante un sistema lineal con unas mediciones y ruido que sigan una distribución gaussiana. En ese caso, el filtro es capaz de ofrecer una estimación del estado del sistema y una medida de la incertidumbre asociada a la estimación.

En muchos casos los filtros de Kalman se utilizan en sistemas dinámicos discretizados en el tiempo. El estado del sistema es una variable n-dimensional x_t. La observación del sistema en cada iteración en un variable l-dimensional z_t que sólo depende de x_t.

Para utilizar el KF es necesario especificar una serie de matrices para adaptarlo al modelo del problema que estemos abordando:

- F_t: El modelo de transición de estado.
- H_t: El modelo de observación.
- Q_t: La covarianza del ruido del proceso.
- R_t: La covarianza del ruido de la observación.

El filtro de Kalman asume que el estado real x_t evoluciona desde el estado en (t-1) mediante la ecuación de dinámica de estado (predicción):

$$x_t = F_t \Delta x_{t-1} + w_t \tag{6.1}$$

siendo F_t el modelo de transición de estado que se aplica al estado previo x_{t-1} y w_t el ruido del proceso con covarianza Q_t.

Por otra parte, asume que la observación z_t en la iteración t del estado real x_t se realiza de acuerdo a (modelo de observación):

$$z_t = H_t \Delta x_t + v_t \tag{6.2}$$

donde H_t es el modelo de observación que convierte el espacio del estado en el espacio observado, mientras que v_t representa el ruido existente de la observación con covarianza R_t.

El KF es un estimador recursivo, es decir, sólo utiliza para la estimación el estado actual el estado de la iteración anterior la observación actual, sin tener en cuenta las observaciones anteriores ni otros estados previos, es decir, asume un sistema markoviano.

El vector de estado x_t sigue una distribución gaussiana n-dimensinal $\widehat{x}_t$ con una matriz de covarianza P. A su vez, el vector de observación sigue una distribución gaussiana l-dimensional $H_t\widehat{x}_{t|t-1}$ que tiene un matriz de covarianza S.

La estimación del estado en cada iteración se realiza en dos pasos, el primer paso es conocido como *predicción*, que se encarga de predecir el estado $\widehat{x}_{t|t-1}$ y su covarianza $\widehat{P}_{t|t-1}$, mientras que el segundo paso se conoce como *correción*, donde se utiliza la observación para refinar el estado estimado y así $\widehat{x}_t$ y P_t. A continuación se muestra el algoritmo completo:

Predicción	
Predicción del estado	$\hat{x}_{t\|t-1} = F_t\hat{x}_{t-1}$
Predicción de la covarianza	$P_{t\|t-1} = F_tP_{t-1}F_t^T + Q_t$
Actualización	
Innovación o residuo	$\tilde{y}_t = z_t - H_t\hat{x}_{t\|t-1}$
Covarianza del residuo	$S_t = H_tP_{t\|t-1}H_t^t + R_t$
Ganacia de Kalman	$K_t = P_{t\|t-1}H_t^tS_t^{-1}$
Estimación actualizada del estado	$\hat{x}_t = \hat{x}_{t\|t-1} + K_t(z_t - H_t\hat{x}_{t\|t-1})$
Covarianza actualizada del estado	$P_t = (I - K_tH_t)P_{t\|t-1}$

6.4. Filtro de Kalman Extendido

Una de las condiciones para utilizar el filtro de Kalman es que el sistema sea lineal. Sin embargo, cuando las ecuaciones de estado u observación (o ambas) no son lineales, no

pueden utilizarse las ecuaciones del filtro de Kalman. En este caso, se puede realizar una aproximación que consiste en linealizar las matrices F_t y H_t en torno al vector de estado actual. Para ello, deben sustituirse estas matrices por sus jacobianas JF_t y JH_T.

Estas nuevas matrices JF_t y JH_T sólo deben sustituirse cuando operen sobre matrices. En caso de operar sobre vectores (en la predicción x_t y el cálculo del residuo) se prefiere utilizar las funciones $f()$ y $h()$ con las operaciones originales, ya que la aproximación conlleva una pérdida de exactitud en la medición, dando como resultado el siguiente algoritmo, conocido con el nombre de Filtro de Kalman Extendido (EKF):

Predicción	
Predicción del estado (pronóstico)	$\hat{x}_{t\|t-1} = f(\hat{x}_{t-1})$
Predicción de la covarianza	$P_{t\|t-1} = JF_t P_{t-1} JF_t^T + Q_t$
Actualización	
Innovación o residuo	$\tilde{y}_t = z_t - h(\hat{x}_{t\|t-1})$
Covarianza del residuo	$S_t = JH_t P_{t\|t-1} JH_t^T + R_t$
Ganancia de Kalman	$K_t = P_{t\|t-1} JH_t^T S_t^{-1}$
Estimación actualizada del estado	$\hat{x}_t = \hat{x}_{t\|t-1} + K_t(_t - h(\hat{x}_{t\|t-1}))$
Covarianza actualizada del estado	$P_t = (I - K_t JH_t) P_{t\|t-1}$

El EKF no es un estimador óptimo, y es menos óptimo cuanto menos lineales sean las funciones de estado y de observación. Además, el EKF cuenta con otras desventajas, como que puede no ser estable cuando la hipótesis de partida no es la correcta, no llegando nunca a aproximar correctamente al vector de estado, y que tienden a subestimar la matriz de covarianza.

Bibliografía

[A. J. Davison and Stasse., 2007] I. D. Reid A. J. Davison, N. D. Molton and O. Stasse. Monoslam: Real-time single camera slam. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2007.

[Azuma, 1997] Azuma. Survey of augmented reality. *Presence Teleoperators and Virtual Environments, VOL 6*, pages 355–385, 1997.

[Canny., 1986] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence, VOL 8*, page 679–698, 1986.

[Cantero, 2013] Juan José García Cantero. Herramienta de calibración de sensores rgbd en jderobot. *Proyecto Fin de Carrera, ETSIT, Universidad Rey Juan Carlos*, 2013.

[Castellanos and Tardós, 1999] JA Castellanos and JD Tardós. *Mobile robot localization and map building: A multisensor fusion approach*. Kluwer Academic Publishers, Boston, 1999.

[Cañas Plaza, 2003] José María Cañas Plaza. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónssssomo*. PhD thesis, Universidad Politénica de Madrid, 2003.

[Civera *et al.*, 2008] Javier Civera, Andrew J. Davison, and J. M. M. Montiel. Inverse depth parametrization for monocular slam, vol 24. *IEEE TRANSACTIONS ON ROBOTICS*, pages 932–945, 2008.

[Civera *et al.*, 2010] Javier Civera, Andrew J. Davison, and J. M. M. Montiel. 1-point ransac for ekf filtering: Application to real-time structure from motion and visual odometry. *Journal of Field Robotics, VOL 27*, pages 609–631, 2010.

[Davison, 2002] Andrew J. Davison. Slam with a single camera. *SLAM/CML Workshop at ICRA 2002*, 2002.

[Davison, 2003] Andrew J. Davison. Real-time simultaneous localization and mapping with a single camera. *In Proc. International Conference on Computer Vision*, pages 1403–1410, 2003.

[Díaz, 2013] Agustín Gallardo Díaz. Herramienta de calibración de cámaras en jderobot. *Proyecto Fin de Carrera, ETSII, Universidad Rey Juan Carlos*, 2013.

[Fischler and Bolles, 1981] M. A. Fischler and R. C. Bolles. Random sample consensus, a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM, VOL 24*, pages 381–395, 1981.

[Hahnel *et al.*, 2003] D. Hahnel, Wolfram Burgard, Dieter Fox, and Sebastian B. Thrun. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. *Intelligent Robots and Systems, 2003 (IROS 2003)*, 2003.

[Harris and Stephens, 1988] C. G. Harris and M. Stephens. A combined corner and edge detector. *In Proceedings of the 4th Alvey Vision Conference*, page 147–151, 1988.

[Hartley and Zisserman, 2000] R. Hartley and A. Zisserman. Multiple view geometry in computer vision. *Cambridge University Press*, 2000.

[H.J.S. Feder and Smith, 1999] J.J. Leonard H.J.S. Feder and C.M. Smith. Adaptive mobile robot navigation and mapping. *International Journal of Robotics Research, VOL 8*, pages 650–668, 1999.

[Kachach, 2008] Redouane Kachach. Calibración automática de cámaras en la plataforma jdec. *Proyecto Fin de Carrera, ETSII, Universidad Rey Juan Carlos*, 2008.

[Kalman, 1960] R.E Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering, VOL 82*, pages 35–45, 1960.

[Klein and Murray., 2007] G. Klein and D. Murray. Parallel tracking and mapping for small ar workspaces. *In Proceedings of the Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234, 2007.

[Konolige and Agrawal, 2008] K. Konolige and M. Agrawal. Frameslam: From bundle adjustment to realtime visual mappping. *IEEE Trans actions on Robotics, VOL 24*, pages 1066–1077, 2008.

[Microsoft, 2011] PhotoTourism. Microsoft. Phototourism. *http://phototour.cs.washington.edu/. Visitada fecha: 3-2-2014*, 2011.

[Milgram and Kishino, 1994] P Milgram and A Kishino. Taxonomy of mixed reality visual displays. *IEICE Transactions on Information and Systems, VOL 77*, pages 1321–1329, 1994.

[Montiel and Zisserman., 2003] D. Ortín. M. M. Montiel and A. Zisserman. Automated multisensor polyhedral model acquisition. *In Proceedings of the IEEE International Conference on Robotics and Automation, VOL 1*, pages 1007–1012, 2003.

[Newcombe *et al.*, 2011] Richard A. Newcombe, Steve J. Lovegrove, and Andrew J. Dtam: Dense tracking and mapping in real-time. *IEEE International Conference on Computer Vision*, pages 2320–2327, 2011.

[Nister, 2003] D. Nister. An efficient solution to the five-point relative pose problem. *Pattern Analysis and Machine Intelligence, IEEE Transactions on, VOL 26*, pages 756–770, 2003.

['Oscar G. Grasa, 2011] J. M. M. Montiel 'Oscar G. Grasa, Javier Civera. Ekf monocular slam with relocalization for laparoscopic sequences. *IEEE International Conference on Robotics and Automation*, pages 4816–4821, 2011.

[Ramos, 2010] Luis Miguel Lóez Ramos. Autolocalización en tiempo real mediante seguimiento visual monocular. *Proyecto Fin de Carrera, ETSIT, Universidad Rey Juan Carlos*, 2010.

[Rosten *et al.*, 2010] Edward Rosten, R. Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *Pattern Analysis and Machine Intelligence, VOL 32*, pages 105–119, 2010.

[Strasdat *et al.*, 2012] Hauke Strasdat, J. M. M. Montiel, and Andrew J. Davison. Visual slam: Why filter? *Image and Vision Computing VOL 30*, pages 65–77, 2012.