Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

MÁSTER OFICIAL EN VISIÓN ARTIFICIAL

Trabajo Fin de Máster

Autolocalización visual robusta basada en marcadores

Autor: Alberto López-Cerón Pinilla

Tutor: Prof. Dr. José María Cañas Plaza

Curso académico 2014/2015

A mi pequeña y a mi pequeño

Agradecimientos

En primer lugar no puedo más que agradecer a José María, tutor de este Trabajo Fin de Máster, su dedicación, su apoyo y su comprensión a lo largo del desarrollo de este proyecto. Gracias por tu soporte, por facilitarme siempre las cosas, por buscar la mejor manera de que realizara este trabajo.

También quisiera dar las gracias a todos los profesores del Máster, sobre todo a Antonio y Juanjo, que me acogieron en un primer momento y que siempre estuvieron disponibles para lo que necesitara.

Me gustaría dar las gracias igualmente a los compañeros del Grupo de Robótica, aunque sólo os conozca por correo electrónico: es alucinante vuestra disponibilidad y capacidad para ayudar y resolver problemas.

No puedo olvidarme de agradecer a mi familia su apoyo constante e incondicional, no sólo en esto, sino a lo largo de toda mi vida. Además, claro está, de su apoyo logístico en los momentos álgidos de actividad: ¡gracias súper-abuelos!

Gracias Jorgetón por tus sonrisas y por tus abrazos, por entender que papá se tenía que quedar en casa, *tabajando...*

Y a ti, Marti, gracias por todo, por estar siempre, por tu apoyo y tu alegría constante, por ser mi luz, por hacerme feliz. ¡Infinito!

Índice general

Índice de figuras

Resumen

Uno de los ámbitos destacados dentro de la visión artificial es el de la autolocalización visual, que consiste en la determinación de la posición y la orientación de la cámara utilizando para ello únicamente las imágenes recibidas de ésta, sin ninguna información adicional. Los algoritmos dedicados a este fin abren la puerta a múltiples aplicaciones dentro de la visión artificial propiamente dicha, como por ejemplo la realidad aumentada, y de otros campos asociados como la robótica.

Este Trabajo de Fin de Máster se encuadra dentro de la autolocalización visual en robótica, siendo su objetivo principal programar y caracterizar un algoritmo de autolocalización visual basado en marcadores. Para ello se ha desarrollado una aplicación que, a partir de la detección de balizas visuales en una imagen, estima la posición y la orientación de la cámara que la capturó, presentando el modelo de cámara resultante en una ventana OpenGL. El algoritmo se ha validado experimentalmente realizando estudios de precisión en dos ámbitos: por un lado en el entorno simulado de Gazebo, haciendo uso de un modelo robótico virtual y sus cámaras asociadas, y por otro en un entorno real con una cámara de videoconferencia.

Para el desarrollo del proyecto se ha utilizado la plataforma JdeRobot y el lenguaje de programación C++, mientras que las librerías utilizadas más importantes han sido las siguientes: AprilTags para la detección de balizas, OpenCV para el procesamiento de imágenes, OpenGL para la representación 3D, Eigen para los cálculos geométricos y Qt como biblioteca gráfica. También se ha hecho uso de Matlab para el análisis de información y la generación de las gráficas de los estudios de precisión.

Capítulo 1

Introducción

Como humanos, percibimos la estructura y los detalles del mundo que nos rodea con aparente facilidad. Pensemos por ejemplo en lo claramente que captamos los distintos aspectos del jarrón de flores de la mesa de al lado. Podemos decir la forma y transparencia de cada pétalo y distinguir cada flor del fondo de la imagen sin esfuerzo. O, mirando un retrato enmarcado de la familia, podemos fácilmente contar (y nombrar) a todas las personas de la fotografía, incluso adivinar sus emociones a partir de la expresión de sus caras.

Todo esto se lo debemos al sentido de la vista y a un cerebro preparado para procesar e interpretar toda la información que recibe. Éste divide la señal visual en muchos canales que suministran diferentes tipos de información, además de que tiene un sistema de atención que identifica, de manera independiente de la tarea, las partes importantes de una imagen a la vez que descarta el examen de otras.

Los científicos llevan décadas tratando de entender cómo funciona el sistema visual y, aunque han podido desentrañar algunos de sus principios, la solución completa a este rompecabezas es todavía difícil de alcanzar. Por otra parte, los investigadores en visión artificial han estado en paralelo desarrollando técnicas matemáticas e informáticas para intentar reproducir esta habilidad humana en ordenadores.

Este trabajo se enmarca dentro de la visión artificial en robótica, más concretamente en el campo de la autolocalización visual. El presente capítulo está dedicado a exponer una visión global de este contexto.

1.1 Visión artificial

Actualmente se dispone de técnicas fiables para calcular con precisión el modelo 3D de un entorno a partir del solapado parcial de múltiples fotografías. Dado un conjunto suficientemente grande de vistas de un objeto, se pueden crear modelos 3D utilizando correspondencia estereoscópica. Se puede realizar el seguimiento de una persona frente a un fondo complejo. Se puede incluso intentar encontrar y nombrar a todas las personas de una fotografía usando reconocimiento facial. Sin embargo, a pesar de todos los avances, el sueño de tener un ordenador interpretando una imagen igual que lo haría un niño de dos años queda todavía lejano.

¿Por qué la visión es tan difícil? En parte porque es un problema inverso, en donde se intenta despejar algunas incógnitas con información insuficiente para especificar completamente la solución. Es por ello que se recurre a modelos probabilísticos y basados en la física para eliminar la ambigüedad entre las posibles soluciones.

La visión artificial, por tanto, intenta describir el mundo observado en una imagen (o varias) y reconstruir sus propiedades, como por ejemplo la forma, la iluminación o la distribución de color. Para ello aplica una transformación a los datos procedentes de una cámara para obtener una decisión o una nueva representación, que permita conseguir un determinado objetivo. Los datos de entrada pueden incluir alguna información contextual como "la cámara está montada en un coche", mientras que la decisión puede ser del tipo "hay una persona en esta escena". Una nueva representación puede significar convertir una imagen en color a escala de grises o eliminar el movimiento de la cámara de una secuencia de imágenes.

Cuando la visión computarizada arrancó en los primeros años 70, fue observada como el componente de percepción visual de una ambiciosa agenda para imitar la inteligencia humana y dotar a los robots de comportamiento inteligente. Al mismo tiempo, algunos pioneros de la inteligencia artificial y la robótica creyeron que resolver el problema de la información visual sería un paso fácil en el camino de resolver problemas mayores como el razonamiento de alto nivel o la planificación. Sabemos desde hace algún tiempo que esto no es precisamente así.

Lo que distinguió a la visión artificial del campo ya existente del procesamiento digital de imágenes fue un deseo de recuperar la estructura tridimensional del mundo a partir de las imágenes, y usar esto como trampolín en pos del entendimiento completo de la escena. Procesar esta descripción es difícil principalmente por la ambigüedad inherente a la imagen: como el proceso de formación de ésta capta sólo dos dimensiones, una infinidad de escenas tridimensionales pueden "explicar" la misma imagen bidimensional.

Por tanto, ninguna imagen contiene suficiente información como para permitir la reconstrucción de la escena tridimensional que la formó.

En los años 80 se puso mucha más atención en técnicas matemáticas más sofisticadas para llevar a cabo análisis cuantitativos de imágenes y escenas. En esta década, Marr introdujo su noción sobre los tres niveles de descripción de un sistema de procesamiento de información visual:

- **Teoría computacional**: ¿Cuál es el objetivo de la computación y cuáles las limitaciones conocidas o que pueden influir en el problema?
- **Representaciones y algoritmos**: ¿Cómo se representan la entrada, la salida y la información intermedia, y qué algoritmos se utilizan para calcular el resultado deseado?
- **Implementación hardware**: ¿Cómo se mapean las representaciones y algoritmos sobre hardware real? Por el contrario, ¿cómo se pueden usar las limitaciones hardware para guiar la elección de la representación y el algoritmo? Con el uso creciente de los chips gráficos (GPUs) y de arquitecturas multi-núcleo, esta cuestión es de nuevo muy relevante.

En los años 90 empezaron a aparecer los primeros ordenadores capaces de procesar imágenes lo suficientemente rápido. Además se comenzó a dividir los posibles problemas en otros más específicos, con lo que a partir de entonces la visión artificial empezó a emplearse para múltiples y diferentes tareas y su desarrollo fue concentrándose en problemas como la segmentación, el reconocimiento de formas o el filtrado de bordes.

Actualmente el abaratamiento del hardware y el aumento exponencial de la capacidad de proceso han favorecido el desarrollo de este campo de investigación, por lo que se pueden encontrar múltiples y variadas aplicaciones. A continuación se presentan una serie de ámbitos donde la visión artificial ha tenido implantación, detallando brevemente algunas aplicaciones de ejemplo:

- **Industria**: las tareas de inspección y control de calidad en sistemas industriales son fundamentales y la visión artificial permite cumplir un máximo de exigencia sin afectar al ritmo de producción. La inspección del producto final en fábricas de embotellado, la medición con visión estereoscópica de tolerancias en alas de avión o el envasado automático son algunos ejemplos.

Figura 1 - Aplicaciones industriales: control de calidad y envasado automático

- **Vigilancia y seguridad**: desde vigilancia en grandes eventos mediante identificación de personas hasta detección de intrusos en estancias privadas, pasando por análisis del tráfico de autopistas, el reconocimiento de matrículas en aparcamientos o la monitorización de piscinas para detectar ahogamientos.
- **Identificación**: el reconocimiento biométrico (de cara, huellas, ojo...) permite sistemas de identificación en los que el uso de llaves, tarjetas de identificación o contraseñas es innecesario.
- **Control de tráfico y seguridad vial**: el reconocimiento automático de matrículas, la detección de obstáculos inesperados, la monitorización del estado del conductor o el reconocimiento de señales son algunos ejemplos de la utilidad de la visión artificial en este ámbito.
- **Deportes**: seguramente la aplicación más conocida es el "Ojo de halcón" utilizado en los partidos de tenis para determinar cuál ha sido la trayectoria de la pelota en una jugada. También se emplea la visión artificial para el análisis biomecánico y la monitorización de deportistas o para añadir información a las imágenes de televisión mediante realidad aumentada, como por ejemplo las distancias de referencia en los saltos de longitud.

Figura 2 - Aplicaciones deportivas: Ojo de halcón y realidad aumentada en el salto de longitud

- **Robótica**: los robots interpretan el entorno que les rodea a través de sus sensores, como por ejemplo las cámaras de vídeo. Con la información que proporcionan pueden localizarse, detectar objetos e incluso seguirlos.
- **Entretenimiento**: el sector de los videojuegos ha contribuido notablemente al desarrollo de la visión artificial. Ejemplos claros de su uso son el dispositivo *Kinect* desarrollado por Microsoft y el *Eye Toy*, desarrollado por Sony para su consola Play Station, para el reconocimiento de los gestos realizados por los jugadores. En la industria cinematográfica se utiliza mucho últimamente la técnica de captura del movimiento de actores para su posterior animación.

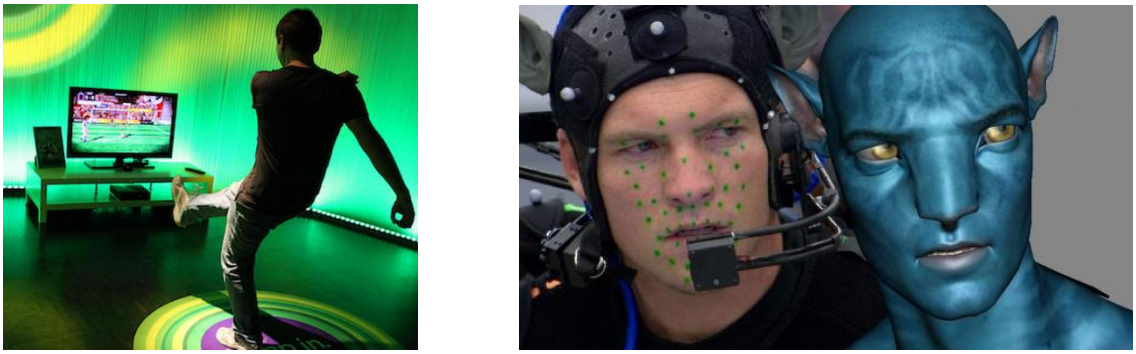

Figura 3 - Aplicaciones de entretenimiento: Kinect y captura de movimiento en "Avatar"

- **Medicina**: algunas aplicaciones dentro del ámbito de la medicina son el registro de imágenes pre-operatorias e intra-operatorias, la detección y caracterización automática de tumores o la mejora de la imagen de microscopía.
- **Mapeo topográfico**: por ejemplo, construcción totalmente automatizada de mapas y de modelos 3D a partir de fotos aéreas. En otro sentido, la aplicación *StreetView*, asociada al servicio de mapas de Google, utiliza visión artificial para el emborronado automático de caras y matrículas.
- **Aplicaciones móviles**: la visión artificial ha encontrado múltiples posibilidades en este sector gracias a la llegada de los teléfonos inteligentes. Como ejemplos se puede nombrar a la aplicación *CamScanner*, que convierte el teléfono en un escáner de bolsillo, y a *Word Lens*, que traduce cualquier texto que se enfoque con la cámara del móvil, mostrando el resultado sobre la propia imagen empleando realidad aumentada.

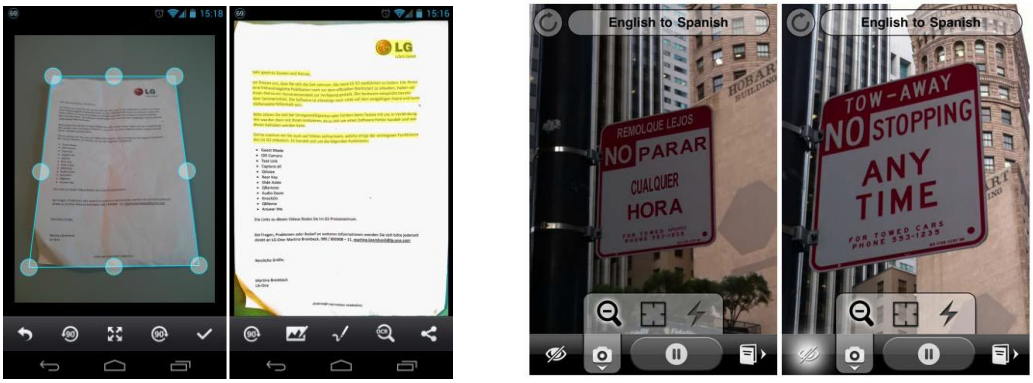

Figura 4 - Aplicaciones móviles: *CamScanner* y *World Lens*

En varios de los ámbitos recién comentados se ha indicado como ejemplo aplicaciones que hacen uso de la realidad aumentada (en inglés *Augmented Reality*, AR). Esta técnica consiste en el insertado en un vídeo de gráficos creados por ordenador, con la correcta posición, escala, orientación y movimiento en relación a los objetos presentes en la toma. Una de las aplicaciones fundamentales de la autolocalización visual es servir de base a esta técnica, ya que el conocimiento preciso en tiempo real de la posición y orientación de la cámara es necesario para poder insertar los objetos ficticios de manera coherente, como si de verdad estuvieran en la escena real. Las aplicaciones de la realidad aumentada son múltiples y muy diversas, existiendo en todo tipo de ámbitos, como por ejemplo, en videojuegos, publicidad, medicina o automoción, además de los ya comentados deportivo y el de aplicaciones móviles.

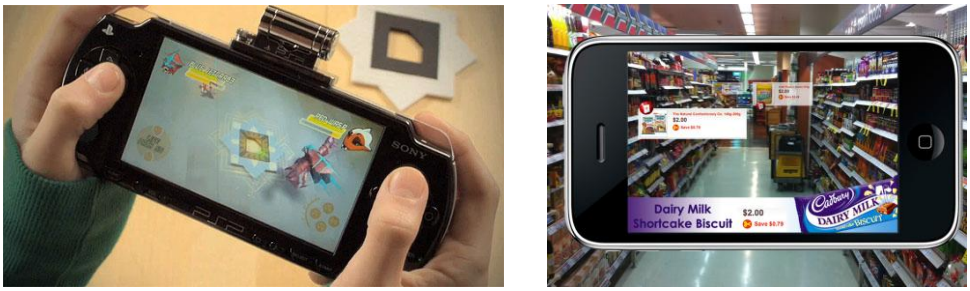

Figura 5 - Ejemplos de AR: Videojuego *Invizimals* y publicidad en supermercado

1.2 Robótica

La robótica es la rama de las ingenierías mecánica, electrónica y de las ciencias de la computación que se encarga del diseño, construcción, operación y programación de robots. Un robot se puede definir como un sistema informático que posee cierto grado de inteligencia, capaz de percibir su entorno y de interaccionar con él. Los robots se utilizan para desempeñar labores de riesgo o que requieren de una fuerza, velocidad o precisión que está fuera del alcance humano, aunque también existen robots cuya finalidad es lúdica o social. En cualquier caso, hay que programarlos para que puedan

percibir su entorno (a través de sensores) y conseguir sus objetivos (mediante actuadores).

Figura 6 - Brazos automáticos en factoría de automoción y sistema quirúrgico Da Vinci

Los ámbitos en los que se utilizan los robots son muy diversos, siendo seguramente el industrial donde su uso está más extendido desde hace más tiempo, ya que realizan el trabajo de forma mucho más precisa y barata que los humanos. Un ejemplo son los brazos robóticos utilizados por la industria de la automoción o los sistemas que empaquetan alimentos en las fábricas de comida envasada. Por otra parte, en medicina se utilizan robots en el ámbito quirúrgico: el sistema Da Vinci por ejemplo permite practicar cirugías de alta complejidad poco invasivas y con precisión.

Otro de los campos donde destaca el uso de sistemas robóticos es en el de las misiones espaciales, donde son utilizados para explorar otros planetas sin la necesidad de la presencia humana. Uno de los pioneros fue el robot *Sojourner*, que exploró la superficie de Marte durante unos tres meses.

En los últimos años se han desarrollado diferentes robots especializados en la movilidad en un terreno, ya sea conocido o desconocido, para lo que hacen uso de distintos sensores que les permiten localizar su posición y navegar a un destino solicitado. Un ejemplo es el coche autónomo de Google.

Figura 7 - Robot *Sojourner* y coche de Google

Otro ejemplo que destaca por su movilidad en superficies son los robots aspiradores como *Roomba* o el Dyson 360 Eye, que limpian la casa de manera automática. Éste último utiliza una cámara para autolocalizarse.

Un tipo de robots muy de moda últimamente son los llamados drones, vehículos aéreos no tripulados, a los que, ya sean autónomos o guiados, se les da todo tipo de utilidades, tanto en el ámbito civil como en el militar.

Los robots más llamativos seguramente sean los androides, que imitan la morfología, el comportamiento y el movimiento de los seres humanos, como *Asimo*, fabricado por Honda. Las perspectivas futuras en la robótica pasan por la construcción de robots de propósito general que interaccionen con los humanos y sean capaces de aprender y adaptarse.

Figura 8 - Robot *Asimo*

Con respecto a la visión dentro del ámbito de la robótica, la utilización de cámaras como uno más de los sensores de un robot (incluso el único) es cada vez mayor, ya que se trata de dispositivos potencialmente muy ricos en cuanto a información proporcionada y muy baratos. Por contra, extraer información útil del gran torrente de datos que suministran es bastante complejo.

1.3 Autolocalización visual

Uno de los campos destacados dentro de la visión artificial es el de la autolocalización visual, que consiste en la determinación de la posición y la orientación de la cámara utilizando para ello únicamente las imágenes recibidas de ésta, sin ninguna información adicional. La autolocalización general puede emplear diversos tipos de sensores (laser, ultrasonidos, encoders...) y combinarlos para obtener mejores resultados, pero, si bien el problema es más complicado en el caso únicamente visual, también es verdad que las cámaras son dispositivos mucho más baratos y extendidos.

El problema de la autolocalización visual ha sido abordado por dos comunidades científicas a la vez. Por un lado, la de visión artificial denominó el problema como *Structure from Motion* (SfM), donde la información es procesada por lotes. Por otro, la

comunidad robótica denominó el problema como SLAM (*Simultaneous Localization and Mapping*) visual y tiene restricciones como el tiempo real o la robustez.

1.3.1 Structure from Motion

Dentro de la visión artificial, el 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 observada y las ubicaciones y orientaciones de las cámaras desde donde se tomaron las imágenes.

Tiene su origen en la fotogrametría, que en la segunda mitad del siglo XIX tuvo como objetivo el extraer información geométrica de las imágenes a partir de un conjunto de características identificadas manualmente por el usuario. Lo que mayoritariamente se ha venido investigando al respecto es cómo conseguir la completa automatización del proceso, llegándose a un grado de madurez donde alguno de los algoritmos tiene aplicación comercial, como el *PhotoTourism* de Microsoft.

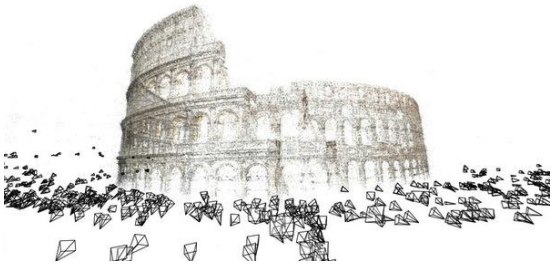

Figura 9 - PhotoTourism de Microsoft

Dadas múltiples vistas, un punto tridimensional puede ser reconstruido mediante triangulación, con el pre-requisito de la calibración de la cámara. La teoría geométrica de SfM permite calcular las matrices de proyección y los puntos 3D utilizando sólo correspondencia entre los puntos de cada imagen.

1.3.2 SLAM

En paralelo al SfM, la comunidad robótica ha abordado la estimación de la posición de un robot móvil y de su entorno desde un punto de vista diferente: el SLAM estima, por un lado, el movimiento (la posición y orientación en cada momento) del robot y, por otro, construye un mapa de los alrededores del mismo a partir del flujo de datos de entrada proporcionado por uno o varios sensores. En muchas ocasiones las mediciones de odometría se incluyen en la estimación, aunque en los últimos años los algoritmos de visión por ordenador han madurado lo suficiente como para tener una posición dominante entre los sensores utilizados por los robots (y en los esquemas de fusión sensorial).

El SLAM monocular o *MonoSLAM* se refiere a la utilización de una única cámara como sensor predominante en la realización del mapeo y la localización. Se ha centrado en el enfoque secuencial de procesado de secuencias de vídeo en tiempo real, siendo esto consecuencia directa de su aplicación: un robot móvil necesita una estimación de su posición de manera continua con el fin de introducirla en el bucle de control.

Figura 10 - Fotograma de vídeo de demostración de MonoSLAM de Andrew Davison

MonoSLAM fue propuesto y desarrollado por Andrew Davison [Davison, 2002] [Davison, 2003] en el año 2002 y, a diferencia de otras técnicas como visión binocular o sensado mediante laser, necesita que la cámara se mueva por el entorno para poder percibir la profundidad. El algoritmo utiliza un filtro extendido de Kalman para estimar la posición y orientación de la cámara, así como una serie de puntos en el espacio 3D. 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 tres puntos. A partir de ese momento el algoritmo es capaz de estimar la posición y la orientación de la cámara y de generar nuevos puntos para crear el mapa y servir como apoyo a la propia localización de la cámara.

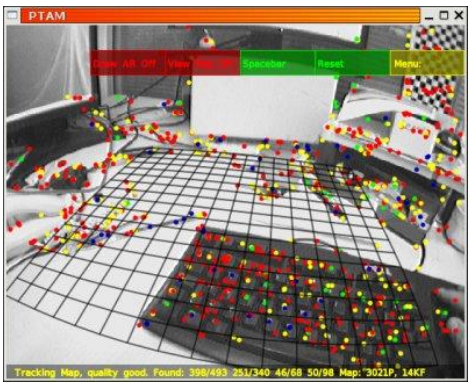

Figura 11 - Algoritmo PTAM en funcionamiento

Otro algoritmo que busca los mismos objetivos que *MonoSLAM*, pero desde un punto de vista totalmente diferente es el llamado PTAM (*Parallel Tracking and Mapping*, seguimiento y mapeado paralelo), desarrollado por George Klein [Klein & Murray,

2007] en 2007. El mayor problema de los algoritmos basados en *MonoSLAM* es que su tiempo de ejecución aumenta exponencialmente con el número de puntos ya que en cada iteración se actualiza tanto la posición de cada elemento del mapa (mapeado) como la posición actual de la cámara (seguimiento). Esto hace que aunque el seguimiento pueda mantenerse en tiempos de ejecución estables, el mapeado hace que a partir de un número de puntos el algoritmo no pueda ejecutarse en tiempo real.

PTAM no hace uso de técnicas de filtrado como *MonoSLAM*, sino que realiza una optimización mediante ajustes de haces, técnica que tiene su origen en el SfM. Sí comparte con él la necesidad de detectar puntos de interés y emparejamiento con los anteriores. El algoritmo parte de la idea de que sólo es necesario funcionar en tiempo real en la parte de seguimiento, mientras que el mapeado no tiene por qué hacerlo ni ser tan eficiente. Por tanto, las dos tareas funcionan en hilos separados de forma asíncrona. También hace uso de cuadros clave (*keyframes*) que se utilizan tanto para localizarse como para crear el mapa de puntos.

1.3.3 Marcadores visuales

En los últimos años, muchas de las aproximaciones a la localización y navegación de robots se han basado en marcadores visuales o en imágenes sin restricciones. Trabajar con estas últimas es una tarea bastante complicada debido a la complejidad y a la demanda computacional de los algoritmos, además de que suelen necesitar, aunque cada vez menos gracias a los avances de los microprocesadores, de hardware dedicado capaz de procesar las tareas en tiempo real.

Por otra parte, la localización en interiores se puede basar en marcadores activos o pasivos. Los activos, como por ejemplo los basados en infrarrojos, se suelen utilizar con hardware simple porque se pueden trazar fácilmente con sensores especiales. Las soluciones basadas en este tipo de marcadores no demandan una carga de tiempo y computación tan alta ya que los marcadores son fáciles de encontrar. Sin embargo, presentan la desventaja de la fuente de alimentación, ya que cada marcador necesita de una, lo que complica la instalación y el mantenimiento. Por otra parte, los marcadores visuales pasivos (señales artificiales diseñadas para ser fácilmente reconocibles y distinguibles las unas de las otras) son más baratos y fáciles de instalar, pero más difíciles computacionalmente de detectar, por lo que deben ser elegidos cuidadosamente. Las características principales que deben diferenciar a un marcador pasivo (también llamado baliza o etiqueta) del alrededor son el contraste y la forma: el primero debe ser lo más alto posible y la segunda debe ser considerablemente distinta de las demás formas cercanas. Como ejemplo se puede nombrar los marcadores de *AprilTags* o

ArUco, que son códigos similares a los QR, pero diseñados para almacenar menos información de manera más robusta.

Figura 12 - Ejemplos de balizas: AprilTags y ArUco

Es posible que la aplicación más conocida de los sistemas basados en marcadores visuales pasivos sea la realidad aumentada, donde, como ya se ha comentado, gráficos generados por ordenador se insertan coherentemente en la imagen captada por la cámara. En otros casos se han utilizado para mejorar la interacción entre humanos y robots, por ejemplo permitiendo dar órdenes con sólo enseñar una carta con el marcador correspondiente.

Por otra parte, la evaluación del rendimiento y la comparativa de sistemas robóticos se ha convertido en una cuestión de importancia para la comunidad investigadora, para lo que los sistemas basados en marcadores visuales son particularmente útiles. Por ejemplo, se pueden utilizar para generar trayectorias de verdad absoluta y bucles de control cerrados. De manera similar, estos marcadores artificiales pueden hacer posible la evaluación de algoritmos de SLAM ejecutándolos bajo algoritmos de control.

1.3.4 Autolocalización visual en el Grupo de Robótica de la URJC

En el Grupo de Robótica de la Universidad Rey Juan Carlos se han desarrollado diversos trabajos que sirven de antecedente a este Trabajo de Fin de Máster (TFM).

El Proyecto de Fin de Carrera (PFC) de Alberto López [López A. , 2010], presentado en 2005, trata sobre la autolocalización de robots móviles empleando dos algoritmos diferentes, uno basado en mallas de probabilidad y otro con filtro de partículas. Eduardo Perdices [Perdices, 2010] trabaja en 2009 con el robot Nao, utilizado en la liga de fútbol robótico *RoboCup*, para el que realiza una serie de algoritmos de localización dentro del campo, utilizando una cámara como único sensor. En 2010 Darío Rodríguez [Rodríguez, 2010] y Luis Miguel López [López L. M., 2010] también presentan trabajos relacionadas con la autolocalización: el primero desarrolla un algoritmo que fusiona la información obtenida de un sensor laser y una cámara, mientras que el segundo diseña e implementa un algoritmo de *MonoSLAM*.

En 2014, Alejandro Hernández [Hernández, 2014] presenta un TFM en el que muestra el trabajo realizado con dos algoritmos de autolocalización, uno basado en *Direct Linear Transformation* (DLT) y otro en PTAM. José Manuel Villarán presenta a principios de 2015 un PFC en el que desarrolla un algoritmo basado en marcadores y más recientemente Daniel Azuara [Azuara, 2015] y Yazmin Cumberbirch [Cumberbirch, 2015] trabajan en algoritmos de autolocalización visual para su PFC que les permiten experimentar con realidad aumentada.

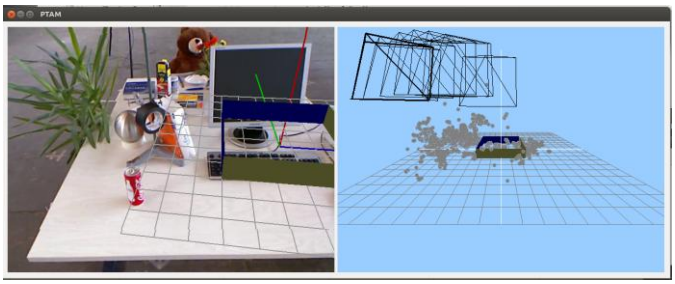

Figura 13 - Interfaz gráfica de PTAM en el TFM de Alejandro Hernández

El Grupo también ha colaborado últimamente con la empresa DTA, que fabrica vehículos para mover cargas en el interior de fábricas, desarrollando un algoritmo de autonavegación basado en marcadores visuales.

El presente Trabajo de Fin de Máster pretende unirse a los trabajos comentados en la programación y caracterización de técnicas de autolocalización, en este caso basadas en marcadores visuales. En los capítulos siguientes se profundizará en la presentación y explicación de la solución adoptada.

Esta memoria presenta en siete capítulos todos los aspectos relacionados con el trabajo realizado, siendo este primero el introductorio. El siguiente fija los objetivos planteados, mientras que el tercero presenta el entorno sobre el que se ha realizado el trabajo. En el capítulo cuatro se exponen los fundamentos teóricos en los que se basan los algoritmos desarrollados, en el cinco el desarrollo software y en el seis los experimentos llevados a cabo. Por último, las conclusiones extraídas y las posibles líneas de trabajo futuras se presentan en el capítulo siete.

Capítulo 2

Objetivos

En el presente capítulo se realiza una descripción del problema planteado y se exponen los objetivos del proyecto, los requisitos que se deben cumplir y la metodología utilizada para el desarrollo.

2.1 Descripción del problema

El objetivo principal de este Trabajo de Fin de Máster es el programar y caracterizar técnicas de autolocalización visual basadas en marcadores para su uso en robots, además de validarlas tanto en un entorno simulado como en uno real. Este objetivo principal se ha articulado en los siguientes dos subobjetivos:

- Programación de un algoritmo de autolocalización visual basado en marcadores que consiga estimar en cada momento la posición y orientación de la cámara utilizando únicamente las imágenes recibidas, haciendo uso del conocimiento previo sobre la ubicación de los marcadores dentro del escenario. El algoritmo incluirá técnicas de fusión tanto espacial (varias balizas) como temporal (memoria en la estimación).
- Caracterización de la solución adoptada mediante un estudio de precisión que recoja la desviación entre la posición estimada y la verdadera en función de distintas variables: distancia al marcador, ángulos entre el marcador y la cámara y posición del marcador en la imagen, con distinto número de marcadores y distintas orientaciones de los mismos. El estudio se realizará tanto en un entorno simulado como en uno real.

2.2 Requisitos

Además de los objetivos recién comentados, la solución implementada debe satisfacer los siguientes requisitos:

- El sistema hará uso de la plataforma de desarrollo JdeRobot, que es el entorno de trabajo del Grupo de Robótica de la Universidad Rey Juan Carlos. Las aplicaciones serán desarrolladas en C++.
- Las aplicaciones desarrolladas se ejecutarán en el entorno GNU/Linux Ubuntu 12.04.
- Para la autolocalización se hará uso únicamente del flujo de vídeo proporcionado por las cámaras.
- El algoritmo implementado debe cumplir lo siguiente:
  - Ser lo suficientemente eficiente como para ser ejecutado en tiempo real consiguiendo una localización precisa.
  - Ser lo suficientemente robusto como para permitir el movimiento suave de la cámara sin perder la localización.

2.3 Metodología de desarrollo

El modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos, que 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 unos de ellos representa una fase del proyecto. Dentro de cada ciclo del modelo en espiral se pueden diferenciar cuatro partes principales que pueden verse en la siguiente figura, donde cada una de las partes tiene un objetivos distinto:

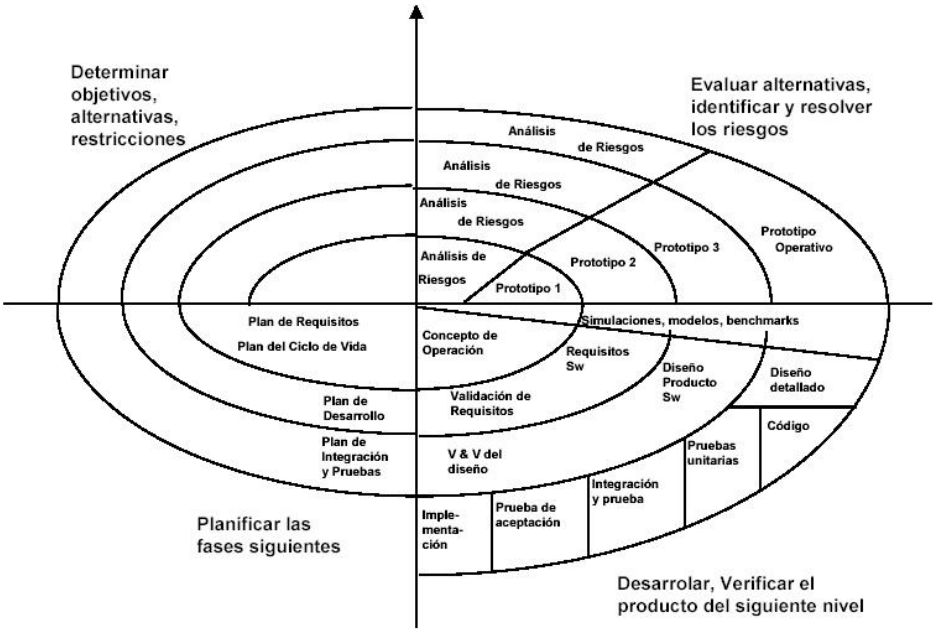

Figura 14 - Modelo en espiral (Barry Boehm, 1986)

- **Determinar objetivos**: se establecen las necesidades que debe cumplir el sistema en cada iteración, teniendo en cuenta los objetivos finales. Típicamente, el coste del ciclo y su complejidad aumentará según avancen las iteraciones.
- **Evaluar alternativas**: determina las diferentes formas de alcanzar los objetivos establecidos en la fase anterior, utilizando distintos puntos de vista: el rendimiento en espacio y tiempo, las formas de gestionar el sistema, etc. Además se consideran explícitamente los riesgos, intentando mitigarlos lo más posible.
- **Desarrollar y verificar**: se desarrolla el producto siguiendo la mejor alternativa para 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 mediante las pruebas realizadas se planifica la siguiente iteración revisando los posibles errores cometidos a lo largo del ciclo y comenzando un nuevo ciclo en espiral.

A continuación se presenta el plan de trabajo del presente Trabajo de Fin de Máster, describiendo las diferentes etapas que componen el proyecto, relacionadas con los ciclos recién comentados. A lo largo del desarrollo se han mantenido reuniones periódicas con el tutor en las que se comentaban los problemas encontrados y se establecían los objetivos a corto plazo. En la *mediawiki* del proyecto[1], dentro del portal de JdeRobot, se puede encontrar más información sobre los avances que se han ido alcanzando ilustrados con imágenes y vídeos.

[1]http://jderobot.org/Alopezceron-tfm

2.3.1 Plan de trabajo

- **Familiarización con JdeRobot**: esta fase tiene como objetivo el aprender a utilizar la plataforma JdeRobot y las aplicaciones y *drivers* que contiene. También se prepara el entorno de desarrollo con la instalación del software necesario y sus dependencias.
- **Familiarización con el simulador Gazebo y los componentes asociados**: en este caso lo que se pretende es aprender a utilizar Gazebo y a crear mundos virtuales que utilizar en él. También se realiza la instalación del componente de tele-operación de drones voladores *uav_viewer* que se integra con Gazebo para el control del *ArDrone.*
- **Desarrollo del algoritmo de autolocalización visual basado en marcadores**: el objetivo de esta etapa realizar la implementación del algoritmo de autolocalización. Para ello se programa la parte que se encarga de la recepción del flujo de vídeo y la que permite el pintado sobre un mundo virtual 3D, en el que se mostrarán las ubicaciones estimadas y reales. Además se implementa el módulo que estima la posición y ubicación de la cámara en cada momento, a partir de las detecciones del módulo de detección de balizas y de un desarrollo propio apoyado en funciones de OpenCV.
- **Estudio de precisión en entorno virtual**: el objetivo de esta fase es llevar a cabo un estudio de precisión utilizando la ventaja que ofrece el simulador al poder disponer de los datos reales de ubicación del cuadricóptero.
- **Estudio de precisión en entorno real**: en esta etapa se adapta el programa para recibir imágenes de una cámara real y para permitir el estudio de precisión posterior, en el que se realizan una serie de experimentos equivalentes a los del caso virtual para validar en un escenario real la solución implementada.

Capítulo 3

Infraestructura

Este capítulo está dedicado a describir los elementos hardware y software que se han utilizado en este Trabajo de Fin de Máster, así como las herramientas que han sido de ayuda para alcanzar los objetivos planteados.

3.1 Elementos hardware

El desarrollo y prueba de los programas realizados se ha llevado a cabo en un ordenador personal al que se le ha conectado una cámara web para los trabajos relacionados con vídeo real. El ordenador tiene un procesador Intel(R) Core(TM) i7-2600k CPU @ 3.40 GHz y 8 GB de RAM, mientras que el sistema operativo utilizado ha sido Ubuntu 12.04, que está basado en GNU/Linux y que se distribuye como software libre.

La cámara con la que se ha trabajado ha sido la Logitech C270, que se conecta al PC a través de un puerto USB. La resolución utilizada ha sido 640x480 y la tasa de refresco de 25 cuadros por segundo.

Figura 15 - Cámara web Logitech C270 y baliza AprilTags construida

Otros elementos físicos que, sin ser hardware, han sido utilizados son los marcadores visuales, para cuya construcción se han imprimido varias de las etiquetas de AprilTags en folios DIN A4 que se han pegado en libros de tapas duras.

3.2 JdeRobot

Este Trabajo de Fin de Máster ha sido desarrollado haciendo uso del proyecto de software libre JdeRobot[1], que es un entorno de programación multiplataforma mantenido por el Grupo de Robótica de la Universidad Rey Juan Carlos. El objetivo de esta plataforma es el de proporcionar una herramienta que facilite el desarrollo de aplicaciones para robótica y visión artificial.

JdeRobot proporciona un entorno de programación basado en componentes distribuidos donde la aplicación se construye haciendo uso de diferentes componentes asíncronos que se ejecutan concurrentemente y que se comunican mediante un esquema cliente-servidor. Además 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.

El acceso a los dispositivos hardware desde el programa de control queda muy simplificado ya que, por ejemplo el obtener medidas de un sensor se reduce a llamar a una función local. Para ello se dispone de una serie de componentes que conectan con sensores y actuadores reales como de distintos *plugins* para Gazebo, que es un simulador de escenarios considerablemente potente y fiable. La comunicación entre componentes, ya sea en entorno real o simulado, se soporta sobre el *middleware* ICE, que permite transmitir información por protocolo TCP/IP, siendo transparente para el desarrollador tanto la arquitectura como el lenguaje de programación utilizada en el otro extremo de la conexión. ICE (*The Internet Communication Engine*) es un software orientado a objetos desarrollado por la empresa ZeroC[2] bajo una licencia doble: GNU GPL y una licencia propietaria. Su uso permite abstraerse de tareas de bajo nivel como abrir conexiones, serializar datos o reintentar conexiones fallidas.

La versión de la plataforma JdeRobot utilizada en este trabajo ha sido la 5.2.4.

3.2.1 Cuadricóptero virtual

En el presente proyecto se ha hecho uso, para el trabajo en entorno simulado, del modelo de cuadricóptero virtual (*ArDrone*) y los drivers asociados creados por Daniel Yagüe en su Proyecto de Fin de Carrera [Yagüe, 2015]. Gracias a este trabajo se dispone de una serie de *plugins* para Gazebo que proporcionan a las aplicaciones

[1] http://jderobot.org/

[2] http://www.zeroc.com/

JdeRobot acceso a los sensores y actuadores del cuadricóptero mediante una serie de interfaces ICE.

La aplicación desarrollada aplica los algoritmos sobre el flujo de imágenes de las dos cámaras de las que dispone el cuadricóptero, una frontal y otra ventral, con lo que se estima la posición y orientación de cada una de ellas. La desviación de los cálculos se puede registrar ya que se dispone también de la información verdadera a través de uno de los *plugins* comentados, lo que ha sido de mucha utilidad para la realización del estudio de precisión.

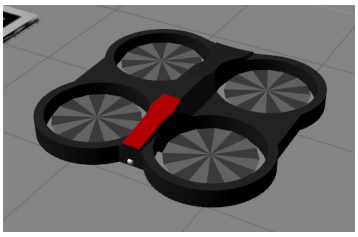

Figura 16 - Cuadricóptero virtual

3.2.2 Componente uav_viewer

Para la tele-operación del cuadricóptero se ha utilizado el componente uav_viewer, implementado por Alberto Martín [Martín, 2014] para el control de robots aéreos reales, pero que gracias al trabajo de Yagüe también permite la tele-operación de drones simulados. Este componente proporciona una manera sencilla de controlar los actuadores del cuadricóptero con el que se conecte y de obtener información de sus sensores más importantes, como altitud, orientación, velocidad o imágenes, para lo que dispone de una interfaz gráfica.

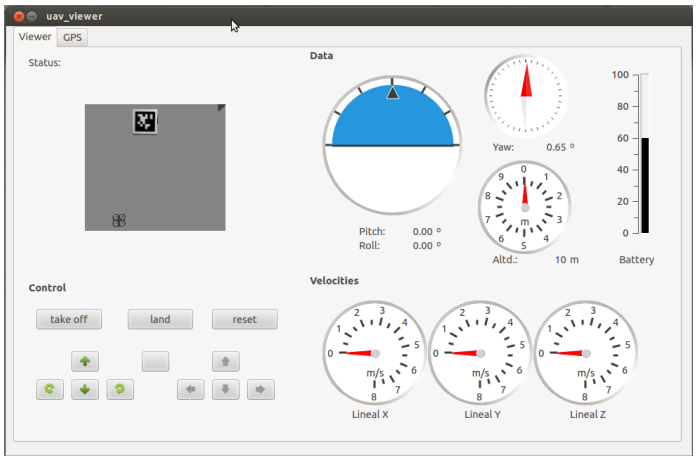

Figura 17 - Interfaz gráfico de usuario de uav_viewer

3.2.3 Progeo

Progeo es el nombre de una librería de geometría proyectiva incluida dentro de JdeRobot. Incluye funciones para relacionar los puntos 2D de la imagen con los puntos 3D del mundo, siendo las llamadas *project* y *backproject* las que se han utilizado en este trabajo. La primera proporciona el punto de la imagen donde proyecta un punto del mundo dado, por lo que se ha utilizado para pintar sobre la imagen proyecciones del mundo real. La segunda realiza la operación inversa, calculando el rayo de retroproyección correspondiente a un punto de la imagen dado, con lo que se ha utilizado para el pintado de las cámaras en el mundo 3D virtual.

3.3 Gazebo

Los simuladores de escenarios facilitan enormemente el desarrollo de aplicaciones en robótica ya que acortan el tiempo de aprendizaje y proporcionan un entorno seguro donde probar el sistema desarrollado antes de su puesta en marcha "real". También permiten no depender de la disponibilidad de los componentes hardware a utilizar en la versión final.

En este Trabajo de Fin de Máster se ha hecho uso de Gazebo[1], cuyo desarrollo comenzó en 2002 con la meta de crear un simulador de alta fidelidad para espacios exteriores. Esta plataforma de software libre bajo licencia GPL permite crear mundos donde insertar objetos y distintos modelos de robots, así como simular los sensores y actuadores con una física determinada, con lo que se consiguen entornos coherentes y realistas.

En este proyecto se ha utilizado la versión 1.8.1 de Gazebo junto con el cuadricóptero virtual comentado, para lo que también se ha creado un mundo con diferentes marcadores visuales repartidos por el espacio.

[1] http://gazebosim.org/

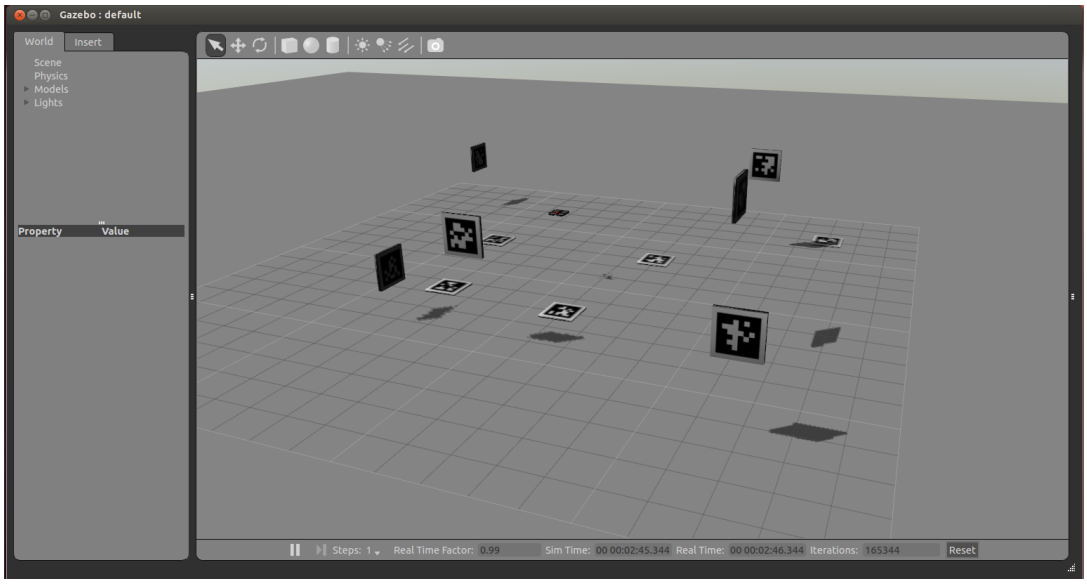

Figura 18 - Mundo virtual con ArDrone y balizas AprilTags en Gazebo

3.4 OpenCV

OpenCV es una librería de visión artificial originalmente desarrollada por Intel, aunque actualmente está mantenida por Willow Garage[1] e Itseez[2]. Su uso está muy extendido gracias a que es multiplataforma (existen versiones para Linux, Windows, MAC OS, iOS y Android) y a que está publicada bajo licencia BSD, lo que permite utilizarla libremente para propósitos comerciales o de investigación. Es por ello que se ha utilizado en infinidad de aplicaciones desde su aparición en 1999.

La librería proporciona un gran número de funciones que abarcan distintas áreas dentro del proceso de visión, como reconocimiento de objetos, calibración de cámaras o visión estereoscópica. Se ha programado teniendo en cuenta sobre todo la eficiencia, haciendo uso de C y C++ optimizados y aprovechando las capacidades que proveen los procesadores multi-núcleo. Además puede utilizar el sistema de primitivas de rendimiento integradas de los procesadores Intel, que son un conjunto de rutinas de bajo nivel específicas de estos procesadores y que ofrecen un gran rendimiento.

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

[2]http://itseez.com/

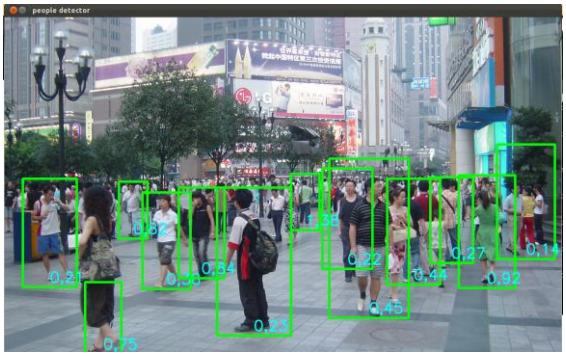

Figura 19 - Detección de personas con OpenCV

La versión con la que se ha trabajado ha sido OpenCV 2.4.3, haciendo uso de sus funciones para la calibración de cámaras, captura de vídeo y resolución del problema *pnp*, además de las utilidades de pintado sobre la imagen.

3.5 OpenGL

OpenGL[1] (acrónimo del inglés *Open Graphics Library*) es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Proporciona una serie de funciones que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Se utiliza en ámbitos como la realidad virtual, la representación científica o la simulación de vuelo. También se usa en desarrollo de videojuegos, donde compite con Direct3D en plataformas Microsoft Windows.

Los fabricantes de hardware, a partir de la especificación, crean bibliotecas de funciones que se ajustan a los requisitos de la misma, utilizando aceleración hardware cuando es posible. Estas implementaciones deben superar unos tests de conformidad para que puedan ser calificadas como conformes a OpenGL. Hay varias implementaciones para múltiples plataformas hardware y software como Linux, Windows y MacOS.

Los propósitos esenciales de OpenGL son dos:

- 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, requiriendo que todas las implementaciones soporten la funcionalidad completa de OpenGL.

En el presente trabajo se ha utilizado OpenGL para visualizar en 3D la posición y orientación estimadas y reales de las cámaras, además de las de las balizas para tener una referencia. Al utilizar OpenGL se aprovecha la GPU de la tarjeta gráfica para

[1]http://www.opengl.org/

realizar las operaciones anteriores, con lo que se libera a la CPU del ordenador de este trabajo y la visualización no ralentiza significativamente la velocidad de ejecución de la aplicación.

3.6 Biblioteca Qt

La biblioteca Qt es una herramienta multiplataforma ampliamente usada para desarrollar aplicaciones con interfaz gráfica de usuario, aunque también se puede utilizar para programas sin interfaz gráfica como herramientas para la línea de comandos o consolas para servidores.

Es un proyecto de software libre que se distribuye bajo los términos de la licencia LGPL y donde participan desarrolladores de la comunidad libre y de las empresas Nokia y Digia, así como de algunas otras. Utiliza C++ de forma nativa, aunque puede ser utilizado en otros lenguajes de programación a través de *bindings*.

Qt funciona en todas las principales plataformas, teniendo su API métodos que dan soporte para diferentes funcionalidades: acceso a bases de datos mediante SQL, uso de XML, gestión de hilos, soporte de red, estructuras de datos... Aplicaciones como Google Earth, Skype, VLC media player o Virtual Box utilizan Qt.

En Qt se hace uso del mecanismo de señales y *slots* para la comunicación entre diferentes partes del programa. Una señal es un evento que un componente de la aplicación puede emitir en un momento dado, mientras que un slot es una función a la que se llama como respuesta a una determinada señal. El programador puede utilizar las señales y slots predefinidos o crear nuevos, y conectarlos de la manera que crea conveniente. Este mecanismo es una de las características centrales de Qt y seguramente la parte que más lo diferencia de las funcionalidades proporcionadas por otros *frameworks*.

En este proyecto se ha hecho uso de la biblioteca Qt en su versión 4.8.6 para la programación del interfaz gráfico de usuario, donde se presenta el vídeo capturado con la detección de las balizas y donde se permite al usuario el realizar distintas acciones y selecciones.

3.7 Biblioteca Eigen

Eigen[1] es una biblioteca C++ de alto nivel para álgebra lineal, distribuida bajo la licencia MPL2. Facilita la realización de operaciones con matrices y vectores y también ofrece métodos de resolución y reducción de sistemas lineales.

[1]http://eigen.tuxfamily.org/

En las aplicaciones desarrolladas se ha utilizado la versión 3.0.5 para realizar las operaciones relacionadas con vectores y matrices, es decir, las transformaciones de rotación y traslación necesarias para el cálculo de la posición 3D de las cámaras.

3.8 Biblioteca Aruco

Aruco es el nombre de una librería desarrollada por el grupo de investigación "Aplicaciones de la visión artificial", de la Universidad de Córdoba, que ofrece la funcionalidad mínima necesaria para crear aplicaciones de realidad aumentada mediante la detección de marcadores visuales. En la aplicación desarrollada para este Trabajo de Fin de Máster se hace uso de la estructura que proporciona para la lectura de ficheros yml de calibración. Se probó su funcionamiento, pero para los marcadores visuales se decidió finalmente utilizar la tecnología de *AprilTags*, explicada a continuación.

3.9 AprilTags

AprilTags es un sistema de detección de marcadores visuales presentado por Edwin Olson [Olson, 2011], de la Universidad de Michigan, que utiliza marcadores 2D (balizas) del estilo de código de barras. Para ello, describe un método para detectar de manera robusta las balizas y propone un algoritmo de segmentación basado en gradientes locales que consigue que las líneas se estimen con precisión. Por otra parte, describe un nuevo sistema de codificación que aborda problemas específicos de los sistemas de códigos de barras 2D: robustez frente a la rotación y frente a los falsos positivos que las imágenes naturales puedan provocar. Al ser *AprilTags* el sistema de marcadores elegido, resulta de gran importancia en el presente Trabajo de Fin de Máster, por lo que se presenta en detalle a continuación.

El sistema *AprilTags* consta de dos componentes principales: el detector de balizas y el sistema de codificación. El primero, cuyo trabajo es el de estimar la posición de los marcadores que pueda haber en la imagen, intenta encontrar regiones de cuatro lados (*quads*) que tengan un interior más oscuro que su exterior. Los marcadores tienen bordes blancos y negros para facilitar esta tarea. El detector está diseñado para tener una tasa de falsos negativos muy baja, por lo que tiene una alta tasa de falsos positivos, pero se apoya en la codificación para reducir esta tasa a niveles aceptables y utilizables. El proceso consta de diferentes fases, descritas a continuación:

3.9.1 Detección de marcadores

Para la detección de líneas en la imagen se computa el gradiente de dirección y magnitud de cada píxel y se agrupan los que tengan direcciones y magnitudes similares. Haciendo uso de mínimos cuadrados ponderados se ajusta un segmento de línea a los píxeles de cada componente y su dirección se determina por el gradiente.

En este punto ya se ha obtenido de una imagen un conjunto de segmentos con dirección. La siguiente tarea es encontrar secuencias de segmentos que formen una forma cuadrada, es decir, lo que se ha llamado un *quad*. El reto está en hacerlo a la vez que se es robusto frente a posibles oclusiones y a ruido en las segmentaciones de las líneas.

El enfoque se basa en una búsqueda recursiva *depth-first*, con una profundidad de cuatro: cada nivel del árbol de búsqueda añade un borde al *quad*. En el nivel uno se consideran todos los segmentos. En los niveles del dos al cuatro se consideran todos los segmentos que empiecen lo suficientemente cerca de donde acabó el segmento previo y que obedezcan a un orden en sentido de las agujas del reloj. La robustez frente a oclusiones y los errores de segmentación se gestionan ajustando el umbral de "suficientemente cerca": aumentando el umbral se pueden manejar huecos significativos alrededor de los bordes.

Una vez que se han encontrado cuatro líneas se crea un candidato a *quad*, siendo sus esquinas las intersecciones entre las líneas que lo componen.

Por último, se computa la matriz de homografía que proyecta puntos 2D en coordenadas homogéneas en el sistema de referencia del marcador al sistema de coordenadas 2D de la imagen. Para ello se hace uso del algoritmo DLT (*Direct Linear Transformation*).

3.9.2 Descodificación de *payload*

La siguiente tarea es la lectura de los bits que forman el marcador, para lo que se procesan las coordenadas de cada bit (o *quad*, relativas al marcador), transformándolas a coordenadas de imagen (utilizando la homografía) y umbralizando los píxeles resultantes. Para ser robusto frente a la iluminación, que puede variar no sólo de un marcador a otro sino incluso dentro del propio marcador, se utiliza un umbral que varía espacialmente.

3.9.3 Sistema de codificación

Una vez que los datos procedentes de un *quad* han sido descodificados, es trabajo del sistema de codificación el decidir si éste es válido o no. Los objetivos de un sistema de codificación son los siguientes:

- Maximizar el número de códigos distinguibles.
- Maximizar el número de errores de bit que se pueden detectar o corregir.
- Minimizar la tasa de falsos positivos entre marcadores.
- Minimizar el número total de bits por marcador (y con ello su tamaño).

Estos objetivos están a menudo en conflicto, por lo que los códigos al final hacen uso de una solución de compromiso. *AprilTags* describe un sistema de codificación basado en *lexicodes* que puede generar códigos para cualquier tamaño de marcador (por ejemplo 3x3, 4x4, 5x5) y mínima distancia de Hamming. Su enfoque garantiza explícitamente la mínima distancia de Hamming para las cuatro rotaciones de cada marcador y elimina los marcadores de baja complejidad geométrica.

Figura 20 - Balizas AprilTags

En la figura anterior se pueden observar ejemplos de balizas pertenecientes a distintas familias de AprilTags, además de que se indica el número de bits frente a la mínima distancia de Hamming entre marcadores válidos.

3.9.4 Biblioteca

El propio Edwin Olson proporciona la biblioteca AprilTags-C[1], que es una implementación en C del algoritmo que propone para la detección de las balizas. Por otra parte, el profesor de la universidad Carnegie Mellon, Michael Kaess, pone a disposición de quien lo desee otra implementación del algoritmo, AprilTags-C++[2], en este caso escrita en el lenguaje de programación C++. Ambas bibliotecas se encargan de la detección de cualquier baliza AprilTags en una imagen dada, informando de su identificador único y de su posición en la imagen (la posición de sus cuatro esquinas).

Durante la realización del presente Trabajo de Fin de Máster se han probado ambas bibliotecas, para finalmente utilizar la implementación basada en C++ para la detección de los marcadores visuales presentes en las imágenes.

[1]https://april.eecs.umich.edu/wiki/index.php/AprilTags-C

[2]http://people.csail.mit.edu/kaess/apriltags/

3.10 Qt Creator

Qt Creator es un entorno de desarrollo integrado (*Integrated Development Environment*, IDE) creado por la empresa Trolltech para el desarrollo de aplicaciones que utilicen las bibliotecas Qt. Es multiplataforma y forma parte del SDK del *framework* de desarrollo de aplicaciones Qt.

Este entorno de desarrollo integra Qt *Designer*, una herramienta para diseñar y construir interfaces gráficas de usuario (GUIs) que permite componer y personalizar los diferentes formularios, así como probarlos con diferentes estilos y propiedades directamente en el editor. Los formularios creados se integran con código utilizando el mecanismo de señales y *slots*. Por otra parte, el editor de código soporta el resaltado de sintaxis para varios lenguajes de programación y el auto-completado.

Para la depuración, Qt Creator incluye un *plugin* que hace de interfaz entre el entorno de desarrollo y los depuradores externos soportados, (GDB o LLDB, por ejemplo) mostrando la información que proporcionan de manera simplificada.

Qt Creator ha sido el entorno de desarrollo integrado utilizado para la programación de las aplicaciones principales de este Trabajo de Fin de Máster.

3.11 Eclipse

Eclipse[1] es un entorno de desarrollo integrado compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", en contraposición a las aplicaciones "Cliente-liviano" basadas en navegadores. Emplea *plugins* para proporcionar su funcionalidad, de manera que permite la adición o desactivación de funcionalidades según el usuario lo requiera. Fue liberado originalmente bajo la *Common Public License*, pero ahora es distribuido bajo la *Eclipse Public License*, siendo ambas licencias de software libre.

Durante la realización del proyecto se ha modificado la aplicación de calibración de cámaras proporcionada por OpenCV para poder calibrar una cámara virtual, para lo que se ha hecho uso del entorno de desarrollo Eclipse en su versión Juno.

3.12 Matlab

Matlab (abreviatura de *Matrix Laboratory*, "laboratorio de matrices") es una herramienta de software matemático que ofrece un entorno de desarrollo integrado con

[1] http://www.eclipse.org/

un lenguaje de programación propio (lenguaje M). Está disponible para las plataformas Unix, Windows, Mac OS X y GNU/Linux.

Entre sus prestaciones básicas se hallan la manipulación de matrices, la representación de datos y funciones y la implementación de algoritmos. En este trabajo se ha utilizado para la creación de las gráficas de los estudios de precisión llevados a cabo.

Capítulo 4

Fundamentos teóricos

Este capítulo está dedicado a repasar los fundamentos teóricos en que se basan los diferentes procedimientos utilizados durante la realización de este Trabajo de Máster. Más concretamente, se presentan conceptos de geometría tridimensional, del modelo de cámara *pin-hole* y de calibración de cámaras. También se introduce el problema *perspective-n-point* y se explica brevemente la teoría relacionada con el filtro de Kalman.

4.1 Geometría 3D: Coordenadas homogéneas

Una escena 3D se define por los puntos, líneas y planos que la componen, que necesitan ser ubicados dentro de un sistema de referencia, con lo que cualquier punto del espacio queda identificado por una terna $(x,\ y,\ z)$.

Por otra parte, un sistema de coordenadas homogéneo es el resultante de añadir una dimensión extra a un sistema de referencia dado, con lo que, dadas las coordenadas cartesianas de un punto tridimensional $(x,\ y,\ z)$, las coordenadas homogéneas de ese punto se definen como $(k \cdot x,\ k \cdot y,\ k \cdot z,\ k)$, donde k es constante y distinto de cero (suele usarse k igual a uno). En sentido contrario, para pasar de coordenadas homogéneas a cartesianas se dividen las tres primeras coordenadas por la cuarta y se elimina ésta. Las coordenadas homogéneas hacen que las traslaciones sean transformaciones lineales y permiten representar puntos en el infinito. Son muy útiles además para poder expresar el encadenamiento de transformaciones geométricas mediante productos matriciales.

4.2 Transformaciones lineales y matrices

La variación de la posición y/o el tamaño de los objetos con respecto a los sistemas de referencia se lleva a cabo mediante transformaciones lineales, siendo las más comunes las siguientes: traslación, cambio de escala, rotación y reflexión.

En informática se suele utilizar la notación matricial para describir las transformaciones lineales de los objetos, siendo la convención más empleada la que expresa el punto que se quiere transformar como un vector que se multiplica por una matriz de transformación.

La composición de matrices consiste en la multiplicación de matrices en un orden determinado y es, como ya se ha comentado, una de las principales razones para trabajar con coordenadas homogéneas. El orden en que se multiplican es importante ya que por lo general el producto de matrices no es conmutativo. Sin embargo, en la composición de matrices pueden intervenir tantos factores (matrices) como se requieran. Así, siendo M la matriz compuesta resultante de la composición de las matrices A, B y C (es decir, M = A · B · C), al multiplicar un punto por esta matriz se obtendrá el mismo punto final que resultaría de multiplicarlo sucesivamente por las tres matrices que la componen.

En 3D, la expresión general de una matriz compuesta es por lo general de la siguiente forma:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & d_x \\ a_{21} & a_{22} & a_{23} & d_y \\ a_{31} & a_{32} & a_{33} & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Donde la submatriz A_{ij} representa el cambio de escala y rotación neta y D_i el vector de desplazamiento neto.

Traslaciones

La traslación de un objeto consiste en desplazarlo cierta distancia en una dirección determinada. en 3D, el sistema de referencia homogéneo tendrá cuatro dimensiones, por lo que la traslación de un punto P quedará indicada como:

$$P' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = T \cdot P = \begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Siendo T la matriz de traslación en 3D. El vector $(d_x\ d_y\ d_z)$ se conoce como vector de traslación.

Rotaciones

Para realizar el giro de un objeto en 3D se ha de establecer un eje de rotación así como el ángulo y el sentido alrededor de dicho eje. Las rotaciones en 3D se realizan normalmente aprovechando la base trigonométrica de las rotaciones en 2D, es decir, descomponiendo los giros en sus componentes ortogonales.

En la figura siguiente se muestra un giro 2D alrededor del eje Z de un punto P situado inicialmente en las coordenadas (x, y, z) y que luego se ha girado una ángulo α pasando a estar en las coordenadas (x', y', z'). Al ser un giro en dos dimensiones la componente z no varía, por lo que z es igual a z'.

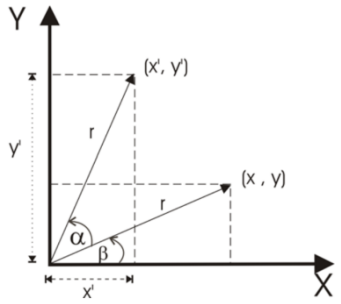

Figura 21 - Giro 2D alrededor del eje Z

De la figura anterior se pueden deducir las fórmulas trigonométricas que producen directamente las nuevas coordenadas:

$$x' = x \cdot \cos(\alpha) - y \cdot sen(\alpha)$$

$$y' = x \cdot \text{sen}(\alpha) + y \cdot cos(\alpha)$$

$$z' = z$$

Finalmente, interpretando las tres ecuaciones anteriores en un sistema de coordenadas homogéneo se puede expresar la rotación como una multiplicación matricial de la siguiente manera:

$$P' = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = R_z \cdot P = \begin{pmatrix} \cos(\alpha) & -sen(\alpha) & 0 & 0 \\ sen(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Las matrices de rotación alrededor del eje X y el eje Y se pueden obtener de la misma manera:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -sen(\alpha) & 0 \\ 0 & sen(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos(\alpha) & 0 & sen(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -sen(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Una rotación 3D cualquiera puede expresarse como el resultado de tres rotaciones consecutivas alrededor de los tres ejes de coordenadas: $R = R_x \cdot R_y \cdot R_z$

Además de las matrices de rotación, existen otras alternativas para representar giros, como por ejemplo los ángulos de Euler, que son tres coordenadas angulares (α, β, y ϒ) que sirven para especificar la orientación de un sistema de referencia de ejes ortogonales (xyz), respecto a otro sistema de referencia también de ejes ortogonales (XYZ). Se basa en escoger un plano en cada uno de los sistemas (el xy y el XY) y utilizar su intersección, llamada línea de nodos, para definir los tres ángulos: α es el ángulo entre el eje x y la línea de nodos, β es el ángulo entre el eje z y el eje Z y ϒ es el ángulo entre la línea de nodos y el eje X.

Otra forma de representación son los cuaterniones, que son una extensión de los números reales similar a la de los números complejos. Fueron introducidos por W. R. Hamilton en 1843 después de que él y otros matemáticos buscaran durante muchos años un sistema numérico que describiera puntos del espacio tridimensional en forma similar a como los números complejos describen puntos del plano. Los cuaterniones son números hipercomplejos de la forma $a + bi + cj + dk$, donde a, b, c, d son números reales y las tres unidades imaginarias i, j, k tienen cuadrado igual a -1.

$$i^2 + j^2 + k^2 = -1$$

También se puede expresar cualquier rotación tridimensional como una secuencia de tres rotaciones seguidas sobre los ejes x, y y z. A estos tres ángulos se les denomina, *roll*, *pitch* y *yaw*, respectivamente. En este trabajo se han utilizado éstos últimos y las matrices de rotación para calcular y representar la posición y orientación de la cámara estimadas por nuestro algoritmo. Los cuaterniones se utilizan en la estructura que entrega el componente desarrollado, ya que ésta tiene un cuaternión para expresar la orientación.

4.3 Modelo de cámara Pin hole

El modelo de cámara *pinhole* (agujero de aguja en inglés) describe la relación matemática entre las coordenadas de un punto 3D y su proyección en el plano de imagen de una cámara *pinhole* ideal, en la que la apertura se describe como un punto y no hay lente que focalice la luz. Este modelo no incluye, por ejemplo, las distorsiones geométricas o el emborronado de objetos no enfocados provocados por las lentes y las

aperturas de tamaño finito. Tampoco tiene en cuenta que la mayoría de las cámaras sólo tienen coordenadas de imagen discretas. Esto significa que este modelo sólo se puede usar como aproximación de primer orden del mapeo de una escena 3D a una imagen 2D. Su validez depende de la calidad de la cámara y, en general, disminuye desde el centro de la imagen hacia los bordes ya que los efectos de la distorsión aumentan.

Como se ha comentado, el modelo asume que todos los rayos de luz pasan por un único punto, infinitamente pequeño, con lo que se recoge la idea de una proyección cónica. La base del modelo se encuentra en el de cámara oscura, en el que los rayos de luz entran por un agujero minúsculo e impactan en la pared contraria, formando una imagen invertida de la escena que la caja tiene enfrente.

El modelo se puede utilizar con las cámaras actuales teniendo presente las limitaciones comentadas, ya que la precisión es aceptable y el modelo cuadra a pesar del uso de lentes.

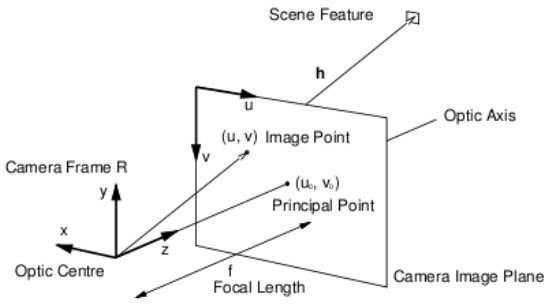

Figura 22 - Modelo de cámara pin-hole

La figura anterior muestra la geometría relacionada del modelo, que consta de los siguientes elementos básicos:

- Un sistema de coordenadas 3D ortogonal cuyo origen es el centro óptico. Los tres ejes del sistema son X, Y y Z, siendo éste último el denominado eje óptico (o eje principal) y el que apunta en la dirección de la cámara.
- Un plano de imagen donde los puntos 3D del mundo se proyectan en dirección al centro óptico. Es paralelo a los ejes X e Y y está localizado a una distancia f del centro, siendo f la distancia focal de la cámara.
- Un punto en la intersección del eje óptico y el plano de imagen, al que se denomina punto principal o centro de la imagen.
- Un punto en algún lugar del mundo 3D (x, y, z) referido a los ejes X, Y y Z.
- El rayo de retroproyección del punto 3D en la imagen, que es la línea recta que pasa por el punto y el centro óptico.

- La proyección del punto 3D en el plano de imagen que está determinado por la intersección del rayo de retroproyección con el plano de imagen.
- Un sistema de coordenadas 2D en el plano de imagen, con los ejes u y v paralelos a X e Y.

Idealmente, la proyección de un punto (x, y, z) en el plano de imagen, es decir, (u, v), se calcula según la siguiente ecuación:

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

En sentido contrario la solución a la correspondencia no es única, ya que dado un punto de la imagen, existen infinitos puntos del mundo que se proyectan en él, y son todos los que pertenecen a la recta que une el centro de proyección con el punto tridimensional.

En los sistemas informáticos el origen de coordenadas de una imagen suele estar situado en la esquina superior izquierda, por lo que la ecuación quedaría de la siguiente forma, donde m y n son la anchura y la altura de la imagen, respectivamente:

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

Sin embargo, aun suponiendo que la lente fuera ideal, si ésta no está perfectamente alineada con el plano de proyección el centro óptico no tiene por qué encontrarse en el centro de la imagen sino en un punto genérico (u0, v0). Además, se puede dar el caso de que la imagen esté ligeramente achatada, lo que se modela con una distancia focal para el eje X y otra distinta para el eje Y, con lo que el modelo básico queda de la siguiente manera:

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

Esta última ecuación, que representa la proyección de los puntos 3D del mundo en el espacio 2D de la imagen se puede representar también en coordenadas homogéneas, invirtiendo los ejes u y v para quitar el signo menos:

$$P_{im} = K \cdot P_w^{cam} \tag{4.4}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}_w^{cam} \tag{4.5}$$

Por tanto, para obtener el punto de la imagen donde se encuentra proyectado un punto del mundo basta con multiplicar la matriz de coordenadas 3D de ese punto por la matriz denominada K, que no es otra que la de parámetros intrínsecos de la cámara. Estos parámetros son los que definen la geometría interna y la óptica de la cámara, tal y como se ha venido explicando. Son constantes en tanto no varíen las características y posiciones relativas entre la óptica y el sensor imagen. La distorsión radial y el *skew* son también parámetros intrínsecos, pero en este caso no se consideran ya que las cámaras que hemos utilizado no introducen grandes distorsiones en ellos.

Sin embargo, en el caso de utilizar cámaras móviles no se puede asumir que el foco está en el origen de coordenadas y que la cámara apunta a la dirección positiva del eje Z. El punto en 3D de las ecuaciones anteriores (P_w^{cam}) está referenciado al sistema de coordenadas de la cámara, pero en nuestro caso lo que se quiere es expresarlo con respecto a otro sistema de referencia absoluto (P_w^{abs}) que no tiene por qué ser el de la cámara. Para ello se tiene que aplicar una rotación y una traslación al sistema de referencia original y a la matriz responsable de este cambio de base se la denomina matriz de rotación y traslación extrínseca, $RT_{ext} = R \cdot T$. Por tanto, un punto referenciado al sistema de la cámara se puede expresar de la siguiente manera:

$$P_w^{cam} = RT_{ext} \cdot P_w^{abs} \tag{4.6}$$

Combinando esta última con (4.4) se obtiene la ecuación general de la proyección de cualquier punto 3D del mundo sobre el plano imagen:

$$P_{im} = K \cdot RT_{ext} \cdot P_w^{abs} \tag{4.7}$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}_w^{cam} \tag{4.8}$$

La estimación de los parámetros extrínsecos, es decir, el cálculo de los coeficientes de la matriz RT, permite la autolocalización de la cámara móvil, objetivo principal de este Trabajo de Fin de Máster.

4.4 Calibración de cámaras

Se llama calibración al proceso mediante el cual se calculan los parámetros intrínsecos y extrínsecos que aparecen en las ecuaciones del modelo de cámara, que, como se ha explicado en el apartado anterior, permiten construir las relaciones geométricas entre la

escena real 3D y sus imágenes 2D. Estas ecuaciones modelan el funcionamiento de la cámara, en una posición y con una orientación particulares.

Para ello se utiliza normalmente una plantilla con numerosos puntos distinguibles de la que se conocen sus posiciones 3D y las proyecciones 2D correspondientes, que se sustituyen en las ecuaciones del modelo, siendo las incógnitas los parámetros de la cámara. Como paso final se debe resolver el sistema de ecuaciones resultante para obtener los parámetros buscados, por lo que cuantos más puntos se usen más precisa será la calibración.

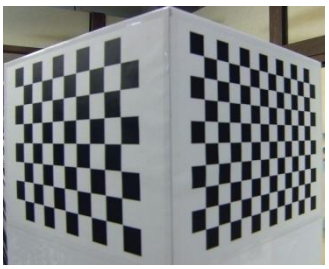

Figura 23 - Ejemplo de plantilla de calibración

El origen de coordenadas del mundo se elige en algún punto de la plantilla para referir a él los puntos 3D de la misma que se van a utilizar para calibrar. Por su parte, los puntos 2D de correspondencia se pueden extraer manualmente o automáticamente. El primer método es útil cuando la cámara va a estar fija y hay que calibrar sólo una vez (o muy pocas veces). Del segundo existen diversas maneras que evitan el tener que indicar manualmente los puntos de la imagen que se corresponden con los puntos 3D de calibración (detección de bordes, rectas, intersecciones, esquinas...).

Diversos autores han desarrollado algoritmos de calibración, que se podrían clasificar, generalmente, en dos categorías: calibración fotogramétrica y auto-calibración. En la primera el proceso se realiza observando la plantilla comentada anteriormente, mientras que en la segunda no se usa ningún objeto de calibración, sino que se mueve la cámara en una escena estática y se calculan las correspondencias entre por lo menos tres imágenes, lo que permite recuperar los parámetros buscados.

En este Trabajo de Fin de Máster se ha utilizado la utilidad de calibración de OpenCV para el cálculo de los parámetros intrínsecos de las cámaras utilizadas. El algoritmo utilizado por esta utilidad se basa en la técnica descrita por Zhang [Zhang, 2000], que sólo requiere que la cámara observe un patrón plano (un damero de ajedrez, por ejemplo) en por lo menos dos orientaciones diferentes. Se trata de una solución de forma cerrada seguida de un refinamiento basado en un criterio de máxima verosimilitud. Este método se encuentra a medio camino entre la calibración

fotogramétrica y la auto-calibración, ya que usa información métrica 2D en vez de información 3D o puramente implícita.

4.5 El problema *perspective-n-point*

El problema indicado en el título de este apartado, también conocido como el problema PnP, es uno de los clásicos problemas en visión artificial y fotogrametría. La estimación de la posición y orientación basándose en puntos de correspondencia se ha estudiado intensamente en las últimas décadas y es esencial en numerosos campos de aplicación.

Dicho problema se podría plantear formalmente de la siguiente manera: dado un conjunto de correspondencias entre n puntos 3D de referencia y sus proyecciones en la imagen, encuéntrese la posición y la orientación de la cámara calibrada con respecto a dichos puntos de control. Es decir, lo que se quiere determinar es la transformación relativa entre el sistema de referencia 3D y el de la cámara: la matriz de rotación-traslación que transfiere el sistema de coordenadas del mundo al de la imagen.

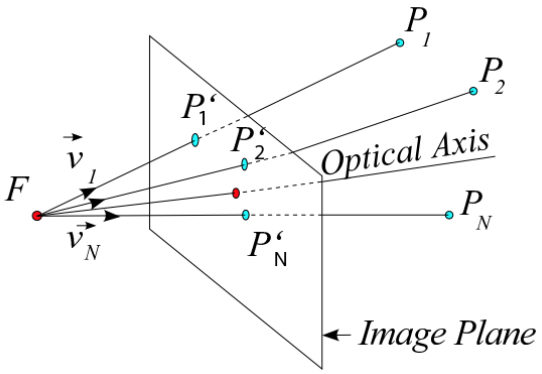

Figura 24 - El problema PnP

Las primeras investigaciones al respecto se concentraron principalmente en el caso de n < 6 debido a lo siguiente: por un lado, para el caso n ≥ 6, PnP se puede formular como un problema de estimación lineal de mínimos cuadrados y se puede resolver fácilmente utilizando el método DLT (*Direct Linear Transformation*); sin embargo cuando el caso es el contrario (n < 6), la esencia del problema se convierte en no lineal y su resolución se dificulta considerablemente. Por la otra parte, el número de puntos de referencia de los que se dispone es normalmente limitado (y menor que 6).

Hay básicamente dos categorías de métodos para resolver el problema en el caso de n $<$ 6: métodos de forma cerrada y métodos iterativos de optimización. Los primeros convierten normalmente el problema en una ecuación polinomial, con lo que basándose en las soluciones a la ecuación se pueden encontrar simultáneamente todas las

soluciones al problema. Sin embargo, como el orden del polinomio suele ser alto, estos métodos no son numéricamente estables cuando los datos son ruidosos. Los métodos iterativos tratan el problema de otra forma, intentando resolverlo mediante la minimización de una función de coste debidamente definida. La gran fuerza de estos métodos reside en que son normalmente extremadamente rápidos y precisos, pero sólo pueden encontrar una solución factible cada vez (cuando n $<$ 6 no se puede garantizar que la solución sea única).

Para la resolución de este problema en este proyecto, se ha hecho uso de la función proporcionada por la librería OpenCV *solvepnp*, a la que se le pasan como argumentos los puntos de control, sus proyecciones y los parámetros intrínsecos de la cámara y que proporciona como resultado la matriz de rotación y el vector de traslación estimados. Esta función permite seleccionar de entre tres métodos para llevar a cabo la resolución:

- Iterativo: se busca la posición y orientación que minimiza el error de reproyección, que es la suma del cuadrado de las distancias existentes entre las proyecciones suministradas y las calculadas con la solución correspondiente. Utiliza la optimización de Levenberg-Marquardt.
- P3P: se basa en el método presentado por Gao, Hou, Tang y Cheng [Gao, Hou, Tang, & Cheng, 2003], que proporciona una descomposición triangular completa del sistema de ecuaciones del caso P3P (tres correspondencias).
- EPnP: utiliza un método no iterativo presentado por F.Moreno-Noguer, V.Lepetit y P.Fua [Lepetit, Moreno-Noguer, & Fua, 2008], que expresa los n puntos 3D de referencia como una suma ponderada de cuatro puntos de control virtuales, con lo que el problema se reduce a estimar las coordenadas de estos puntos. Esto se consigue expresando estas coordenadas como suma ponderada de los vectores propios de una matriz 12 x 12 y resolviendo un pequeño número de ecuaciones cuadráticas para seleccionar los pesos adecuados. El método contempla también operaciones adicionales para mejorar la precisión sin un aumento significativo del tiempo.

En la aplicación desarrollada para este Trabajo de Fin de Máster se ha utilizado el primer método comentado, el iterativo.

4.6 Filtro de Kalman

El filtro de Kalman es un algoritmo desarrollado por Rudolf E. Kalman [Kalman, 1960] que sirve para estimar el estado de un sistema dinámico a partir de observaciones ruidosas. Si todo el ruido es gaussiano, es un estimador óptimo en el sentido de que minimiza el error cuadrático medio de la estimación. Además es recursivo, permitiendo

el proceso de nuevas muestras según van llegando utilizando para la estimación actual sólo el estado de la iteración anterior.

En otras palabras, el filtro de Kalman trata de estimar el estado x de un proceso en un instante de tiempo k utilizando una medida z:

$$x_k = A \cdot x_{k-1} + B \cdot u_k + w_{k-1}$$

$$z_k = H \cdot x_k + v_k$$

Las variables aleatorias w_k y v_k representan el ruido del proceso y el ruido de la medida, respectivamente. Se asume que son independientes la una de la otra, además de blancas y con distribución de probabilidad normal:

$$p(w) = N(0, Q)$$

$$p(v) = N(0, R)$$

La matriz A relaciona el estado actual con el estado en el instante anterior, mientras que la matriz B relaciona la entrada de control opcional con el estado. Por último, la matriz H relaciona el estado con la medida.

Por tanto, el algoritmo de Kalman estima el estado de un proceso haciendo uso de realimentación: se estima el estado en un instante basándose en el estimado del instante anterior y se realimenta con la medida (ruidosa) que se obtiene del sistema. Es por ello que las ecuaciones del filtro se separan en dos grupos: por un lado las ecuaciones de actualización temporal y por el otro las ecuaciones de actualización de la medida. Las primeras son las responsables de proyectar hacia delante (en el tiempo) el estado actual y la covarianza del error para obtener los estimados a priori del siguiente paso temporal. Las segundas se hacen cargo de la realimentación, incorporando una nueva medida al estimado a priori para obtener un "mejorado" estimado a posteriori.

Las ecuaciones de actualización temporal también se pueden ver como ecuaciones predictivas, mientras que las de actualización de la medida se pueden interpretar como ecuaciones correctoras. De hecho, el algoritmo final se asemeja a un algoritmo predictor-corrector de resolución de problemas numéricos, como se muestra en la figura a continuación:

Figura 25 - Bucle predicción-corrección del algoritmo de Kalman

Las ecuaciones completas se presentan a continuación:

Predicción

$$\hat{x}_k^- = A \cdot \hat{x}_{k-1} + B \cdot u_{k-1}$$

$$P_k^- = A \cdot P_{k-1} \cdot A^T + Q$$

Donde:

- $\hat{x}_k^-$ es el estimado a priori.
- $\hat{x}_{k-1}$ el estimado a posteriori.
- P_k^- es la covarianza de error a priori.
- P_{k-1} es la covarianza de error a posteriori.

Corrección

$$K_k = \frac{P_k^- \cdot H^T}{H \cdot P_k^- \cdot H^T + R}$$

$$\hat{x}_k = \hat{x}_k^- + K_k \cdot (z_k - H \cdot \hat{x}_k^-)$$

$$P_k = (I - K_k \cdot H) \cdot P_k^-$$

Donde:

- $(z_k - H \cdot \hat{x}_k^-)$ es el residuo o innovación.
- K_k es la ganancia de Kalman (factor de ganancia que minimiza el error a posteriori).

Capítulo 5

Desarrollo

Este capítulo está dedicado a la presentación y explicación del desarrollo informático llevado a cabo en el presente Trabajo de Fin de Máster, que ha dado lugar a la aplicación llamada *cam_autoloc*. Ésta se encarga de procesar el flujo de imágenes de una cámara para detectar balizas *AprilTags* en ellas, gracias a lo cual obtiene una estimación instantánea de la posición y orientación de la misma que se representa sobre un mundo virtual 3D. El programa permite el registro en un fichero de texto (log) de la posición estimada en todo momento, además de la verdadera si está disponible en un entorno virtual. Si se trabaja con una cámara real también permite el almacenamiento de esta información, pero los datos verdaderos deben ser introducidos manualmente a través del interfaz gráfico de usuario.

5.1 Diseño global

El objetivo principal de la aplicación es implementar un algoritmo de autolocalización visual basado en marcadores de *AprilTags*, por lo que se podría representar como una caja negra a la que se le pasa un flujo de imágenes como entrada para obtener la posición y orientación en cada momento de la cámara que las capta como salida. Para ello, la caja negra debe inicializarse con cierta información a priori: los parámetros de calibración intrínsecos de la cámara y la posición y orientación de las balizas en el mundo. Opcionalmente se le puede también suministrar la información de pose verdadera, que junto con la estimada forma el contenido del fichero de log.

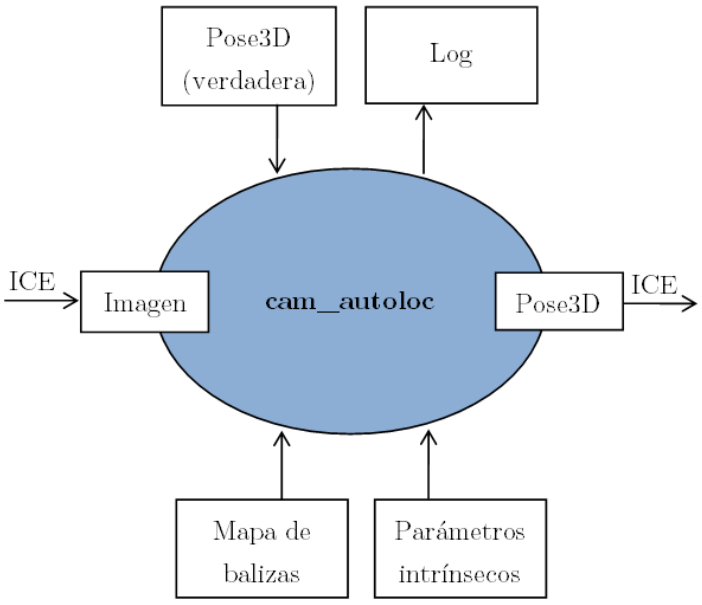

Figura 26 - Diagrama de caja negra

El programa dispone además de un interfaz gráfico que presenta el vídeo capturado junto con varios controles que permiten cierta interacción, como por ejemplo la activación y desactivación de la ventana que muestra el mundo virtual 3D donde se representan la posición verdadera y la estimada. En un apartado posterior se explica este interfaz más en detalle.

Internamente, el componente lleva a cabo el proceso de autolocalización siguiendo el proceso indicado en la figura siguiente, que muestra los grandes bloques funcionales del algoritmo.

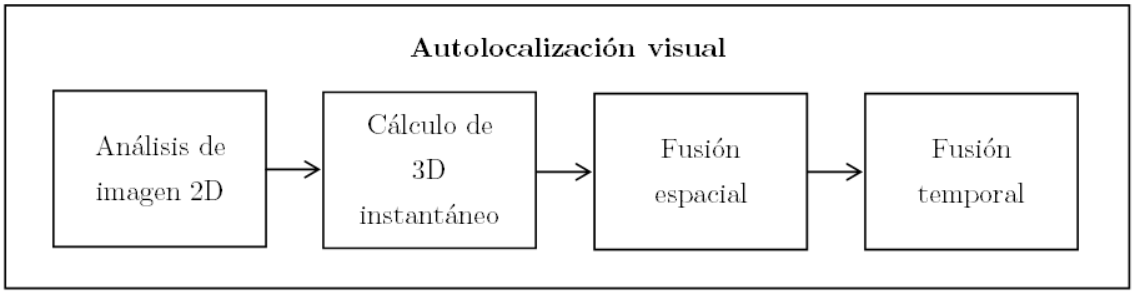

Figura 27 - Diagrama de bloques

Como se puede ver, el proceso comienza con la detección de las balizas en la imagen, sigue con el cálculo de la ubicación dada por cada una de ellas y termina con la fusión de las estimaciones obtenidas, primero de las proporcionadas por todas las detecciones en un momento dado (fusión espacial) y finalmente de la actual con las pasadas (fusión temporal).

La arquitectura software de la aplicación desarrollada se ha diseñado de forma que las distintas funcionalidades representadas en la Figura 27 se reparten entre módulos interconectados por un componente principal, la clase *MainWindow*.

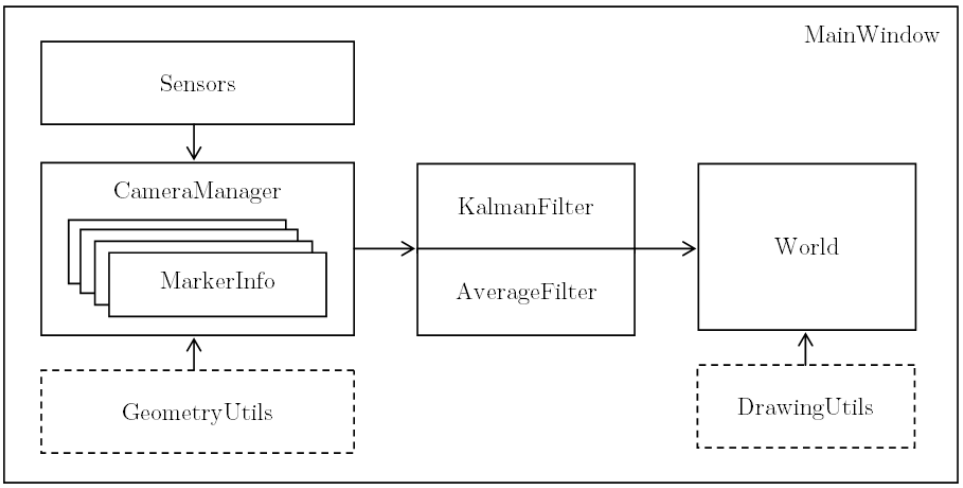

Figura 28 - Diagrama de bloques de las clases principales

Este módulo principal, que contiene instancias de las clases *CameraManager* y *World,* se encarga de transmitirle imágenes a la primera y representar los resultados obtenidos en la segunda. Además, implementa instancias de los dos filtros de fusión temporal disponibles (Kalman o media ponderada), permitiendo seleccionar la aplicación de uno u otro (o ninguno).

Aparte del hilo principal de la aplicación, donde se gestionan los eventos del interfaz gráfico y se muestra el vídeo capturado, la clase dispone de un hilo dedicado al proceso de la imagen y otro al pintado del mundo virtual sobre una ventana OpenGL. A continuación se muestra un extracto del código ejecutado por el primer hilo comentado.

```
m_Image = m_Sensors->getImage();
if (m_CameraManager->ProcessImage(m_Image)) //Si se han detectado balizas
{
   if (m_TemporalFusionFilter == TEMPORAL_FUSION_FILTER_KALMAN)
   {
      Pose p = m_KalmanFilter->
                GetFilteredPose(m_CameraManager->GetEstimatedPose());
      m_CameraManager->SetEstimatedPose(p.GetRT());
   }
   else if (m_TemporalFusionFilter == TEMPORAL_FUSION_FILTER_WEIGHTED_AVERAGE)
   {
      Pose p = m_WeightedAverageFilter->
                GetFilteredPose(m_CameraManager->GetEstimatedPose());
      m_CameraManager->SetEstimatedPose(p.GetRT());
   }
}
```

El método *ProcessImage* se encarga tanto de la detección en 2D de las balizas presentes en la imagen, como del paso a 3D absoluto, por lo que a su salida (si se ejecuta con

éxito) ya se dispone de la estimación instantánea de la ubicación de la cámara, basada sólo en los marcadores visuales y accesible a través de la función *GetEstimatedPose*. Si se activa la aplicación de alguno de los dos filtros de combinación temporal de observaciones (*KalmanFilter* y *AverageFilter* en la Figura 28), la estimación resultante se re-asigna correspondientemente, estableciendo la estimación de salida definitiva del sistema.

Por otra parte, el hilo encargado de la gestión del mundo virtual pinta sobre éste, representados por ejes de coordenadas, el sistema de referencia del mundo y los marcadores visuales presentes en la escena, además de la representación de la pose verdadera y estimada, como se puede ver a continuación.

```
//Ejes de referencia del mundo
DrawingUtils::drawAxis( cvPoint3D32f(0.0, 0.0, 0.0),
                        cvPoint3D32f(1.0, 0.0, 0.0),
                        cvPoint3D32f(0.0, 1.0, 0.0),
                        cvPoint3D32f(0.0, 0.0, 1.0),
                        4.0f);

//Balizas
for (std::map<int, MarkerInfo*>::const_iterator iter =
      CameraManager::MARKERS.begin();
      iter != CameraManager::MARKERS.end();
      ++iter)
{
       DrawingUtils::drawAxis(iter->second->GetPosition(),
                              iter->second->GetAxisX(),
                              iter->second->GetAxisY(),
                              iter->second->GetAxisZ(),
                              2.0f);
}

//Cámara calculada en azul
DrawingUtils::drawCamera(m_CameraManager->GetEstimatedCamera(),
                         cvPoint3D32f(0.0, 0.0, 1.0));

//Cámara real en rojo
DrawingUtils::drawCamera(m_CameraManager->GetRealCamera(),
                         cvPoint3D32f(1.0, 0.0, 0.0));
```

Una vez presentada una visión global del sistema, los apartados siguientes explican más concretamente los distintos pasos del proceso y la manera en que se han implementado.

5.2 Análisis de imagen 2D

El primer paso que realiza la aplicación es la captura y el análisis de la imagen bidimensional recibida, donde se lleva a cabo la búsqueda de los posibles marcadores presentes en la escena.

5.2.1 Obtención de imagen

La clase *Sensors* está pensada para obtener la información proporcionada por los sensores de un robot a través de los interfaces ICE correspondientes. En este caso permite a la aplicación el obtener el flujo de vídeo de la cámara y, en el caso de trabajar en el entorno virtual, es la encargada de proporcionar la información verdadera de posición y orientación que le llega a través de un *plugin* de Gazebo.

```
class Sensors
{
    private:
        cv::Mat image;
        Ice::CommunicatorPtr ic;
        jderobot::CameraPrx cameraCprx;
        jderobot::Pose3DPrx p3dprx;
        jderobot::Pose3DDataPtr pose3DDataPtr;
        ...

    public:
        Sensors(Ice::CommunicatorPtr ic);
        virtual ~Sensors();
        cv::Mat getImage();
        jderobot::Pose3DDataPtr getPose3DData();
        void update();
        ...
};
```

5.2.2 Detección de marcadores

MarkerInfo es el nombre de la clase que representa al marcador visual, por lo que contiene su identificador único (dentro de la familia de *AprilTags* correspondiente), su tamaño (longitud de su lado) y su posición y orientación en el mundo. Esta última información se almacena en forma de matriz de rotación, pero de manera doble: pose del marcador con respecto al origen del mundo y pose del mundo con respecto al marcador.

```
class MarkerInfo
{
    private:
        int m_Id;
        Eigen::Matrix4d m_WorldRT; //Baliza con respecto al mundo
        Eigen::Matrix4d m_MarkerRT; //Mundo con respecto a la baliza
        int m_Size;
         ...

    public:
        MarkerInfo(int id, double x, double y, double z, double rotX,
                   double rotY, double rotZ, double size);
        ...
};
```

En la inicialización del programa se construye un mapa de objetos de esta clase, cada uno construido con la información correspondiente de las balizas que componen la

escena. Este mapa, que es accesible por el resto de módulos y que utiliza como clave el identificador único de la baliza, se genera a partir de un fichero de configuración llamado *markers.txt*, lo que facilita el poder utilizar el programa con diferentes configuraciones de escena.

El proceso de la imagen recibida comienza transmitiendo la imagen en escala de grises al detector de balizas (de la librería AprilTags-C++), que devuelve un *array* con las detecciones encontradas. Cada detección informa del identificador del marcador y de la posición de las cuatro esquinas en la imagen con respecto a la esquina superior izquierda.

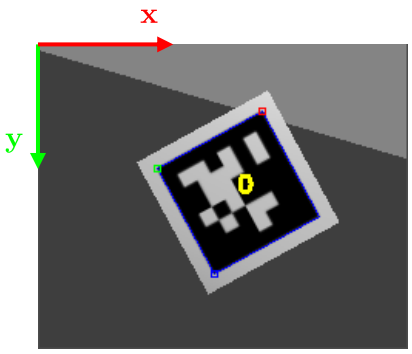

Figura 29 - Detección de marcador visual

A continuación se presenta una muestra de la parte del código donde se obtienen las detecciones recién comentadas:

```
//Conversión a escala de grises
cv::Mat imageGray;
cv::cvtColor(image, imageGray, CV_BGR2GRAY);

//Marcadores detectados
std::vector<AprilTags::TagDetection> detections =
               m_TagDetector->extractTags(imageGray);

//Proceso de las detecciones
for (std::vector<AprilTags::TagDetection>::iterator iter =
         detections.begin(); iter != detections.end(); ++iter)
{
    if (MARKERS.find(iter->id) != MARKERS.end())
    {
       ...
    }
}
```

5.3 Cálculo de 3D instantáneo

El proceso de cálculo de la posición y orientación estimadas por cada marcador visual detectado se explica en este apartado, que presenta en primer lugar el conjunto de utilidades que se han implementado para facilitar la programación del algoritmo en sí.

5.3.1 Utilidades geométricas

El módulo *GeometryUtils* proporciona un conjunto de clases y funciones que resultan de utilidad en los distintos cálculos geométricos y en la representación de la escena virtual 3D.

Line

La clase *Line* está diseñada para representar una línea recta en el espacio tridimensional, que queda determinada al conocerse un punto P(x_0, y_0, z_0) de la misma y un vector director u(u_1, u_2, u_3), mediante sus ecuaciones paramétricas:

$$x = x_0 + u_1 \cdot t$$

$$y = y_0 + u_2 \cdot t$$

$$z = z_0 + u_3 \cdot t$$

Por tanto, dados dos puntos A y B de la recta, las ecuaciones quedarían de la siguiente manera:

$$x = A_x + (A_x - B_x) \cdot t$$

$$y = A_y + (A_y - B_y) \cdot t$$

$$z = A_z + (A_z - B_z) \cdot t$$

Esta clase presenta un constructor al que se le pasan los dos puntos como argumentos y una función que devuelve el punto de la recta correspondiente a un determinado valor de t.

```
class Line
{
    private:
        float x0, y0, z0;
        float u1, u2, u3;

    public:
        Line(HPoint3D a, HPoint3D b);
        HPoint3D GetPoint(float t) const;
};
```

Plane

Un plano queda determinado cuando se conoce un punto P(x_0, y_0, z_0) del mismo y un vector n(A, B, C) perpendicular a él (al plano), mediante la denominada ecuación normal:

$$A \cdot (x - x_0) + B \cdot (y - y_0) + C \cdot (z - z_0) = 0$$

Por tanto, dados dos puntos O y N, siendo O un punto del plano y ON un vector perpendicular al mismo, la ecuación quedaría de la siguiente manera:

$$(N_x - O_x) \cdot (x - O_x) + \left(N_y - O_y\right) \cdot \left(y - O_y\right) + (N_z - O_z) \cdot (z - O_z) = 0$$

Estos dos puntos son los parámetros que se le pasan al constructor de la clase *Plane*.

```
class Plane
{
    private:
        float a, b, c;
        float x0, y0, z0;

    public:
        Plane(HPoint3D o, HPoint3D n);
};
```

Pose

La clase *Pose* representa la posición y orientación en 3D de la cámara, por lo que se construye con información de las coordenadas y de los ángulos *roll*, *pitch* y *yaw* con respecto a un sistema de referencia. El propio constructor de la clase calcula con los datos suministrados la matriz de rotación y traslación correspondiente, que queda almacenada y accesible.

```
class Pose
{
    private:
        Eigen::Matrix4d m_RT;
        double m_Weight;
        ...

    public:
        Pose();
        Pose(float x, float y, float z, float h, float roll,
             float pitch, float yaw);
        Pose(const Eigen::Matrix4d& rt);
        const Eigen::Matrix4d& GetRT() { return m_RT; }
        double GetWeight() { return m_Weight; }
        void SetWeight(double weight) { m_Weight = weight; }
        ...
};
```

Además, debido a que resulta útil para el desarrollo del programa, como se explicará más adelante, esta clase tiene un campo adicional que representa el peso, la importancia de esta posición y orientación.

Funciones

Las siguientes funciones, que hacen uso de las clases recién explicadas, son las que resultan de mayor importancia dentro de la clase *GeometryUtils*:

- ***GetPointOfLine***: dados dos puntos de una recta calcula el punto a una distancia determinada del primero.
- ***GetPointOfLineAndPlane***: calcula el punto de intersección entre una recta y un plano.
- ***BuildRTMat***: dadas unas coordenadas x, y, z y unos ángulos *roll*, *pitch* y *yaw*, calcula la matriz de rotación y traslación correspondiente.
- ***RotationMatrixToRPY***: obtiene los ángulos *roll*, *pitch* y *yaw* correspondientes de una matriz de rotación.
- ***RPYToRotationMatrix***: convierte una terna de ángulos *roll*, *pitch* y *yaw* en la matriz de rotación correspondiente.
- ***GetError*** en x, y, z, *roll*, *pitch*, *yaw*: dados dos objetos de tipo *Pose*, obtiene la diferencia en el parámetro indicado.
- ***GetRadialError***: calcula la distancia euclídea entre las posiciones de las dos poses pasadas como parámetros.
- ***GetAngularError***: calcula una medida de la diferencia entre las orientaciones de las dos poses pasadas como parámetros, la raíz cuadrada de la suma de las diferencias al cuadrado de *roll*, *pitch* y *yaw*.

Las dos primeras funciones se utilizan para el pintado sobre el mundo virtual 3D, mientras que las tres siguientes se usan en diversas partes de la aplicación para facilitar el paso de coordenadas y ángulos a matrices y viceversa. Es en los estudios de precisión donde se hace uso de las tres últimas.

5.3.2 Estimación de pose

La clase *CameraManager* es la que implementa el algoritmo de autolocalización, para lo que contiene atributos que referencian la cámara estimada y el detector de balizas *AprilTags*. La estructura que se utiliza para referenciar la cámara es *TPinHoleCamera* de la librería Progeo de JdeRobot, donde se pueden almacenar tanto los parámetros intrínsecos como los extrínsecos de una cámara modelada como *pin-hole*. Se hace uso de esta misma estructura para referenciar la cámara verdadera, con lo que en todo momento se puede acceder a la información referente a una u otra. En el caso del detector *AprilTags*, se configura para la detección de balizas de la familia 36h11.

En el constructor de la clase se realiza la inicialización de los datos necesarios, obteniendo los parámetros de calibración intrínsecos de la cámara de un fichero externo llamado *camera.yml*. Para la lectura de este fichero y el almacenamiento de su información se ha utilizado la estructura *CameraParameters*, de la librería de realidad aumentada Aruco. Como ya se ha comentado, la inicialización de la posición y

orientación de las balizas en el mundo (el mapa de objetos *MarkerInfo*) se realiza de manera estática en el arranque del programa.

```
class CameraManager
{
    public:
        static const double MARKER_SIZE;
        static const std::map<int, Ardrone::MarkerInfo*> MARKERS;

    private:
        TPinHoleCamera m_RealCamera;
        TPinHoleCamera m_EstimatedCamera;
        aruco::CameraParameters m_CameraParameters;
        Pose m_RealPose;
        Pose m_EstimatedPose;
        AprilTags::TagDetector* m_TagDetector;
        ...

    public:
        CameraManager(const std::string& calibFile);
        virtual ~CameraManager();

        Pose GetRealPose();
        Pose GetEstimatedPose();
        void SetRealPose(double x, double y, double z, double h,
                         double roll, double pitch, double yaw);
        void SetRealPose(const Eigen::Matrix4d& rt);
        void SetEstimatedPose(double x, double y, double z, double h,
                              double roll, double pitch, double yaw);
        void SetEstimatedPose(const Eigen::Matrix4d& rt);
        bool ProcessImage(cv::Mat& image);
        ...

};
```

La función principal de la clase *CameraManager* es la llamada *ProcessImage*, a la que se le pasa una imagen como único parámetro y que en primer lugar detecta las balizas presentes en la misma según se explica en el apartado 5.2.2.

A continuación se recorre el *array* de detecciones, buscando cada identificador en el mapa de marcadores, con lo que si se encuentra se da paso a los cálculos geométricos. Estos comienzan con el uso de la función *solvePnP* de OpenCV que, como ya se ha comentado, calcula la pose relativa entre la cámara y un sistema de referencia dado a partir de la correspondencia de puntos 2D de la imagen con puntos 3D del mundo referenciados a ese sistema. Por tanto, se construye la estructura de puntos 2D con las puntos de las esquinas indicadas en la detección y se le pasa a *solvePnP* junto con los puntos 3D de las esquinas (referenciados al centro de la baliza y en el orden correspondiente a como se han pasado en 2D) y la matriz de parámetros intrínsecos de la cámara. Como resultado se obtienen el vector de traslación y el vector de rotación (este último según el formato de Rodrigues) que determinan la posición y orientación del marcador con respecto al sistema de referencia de la cámara.

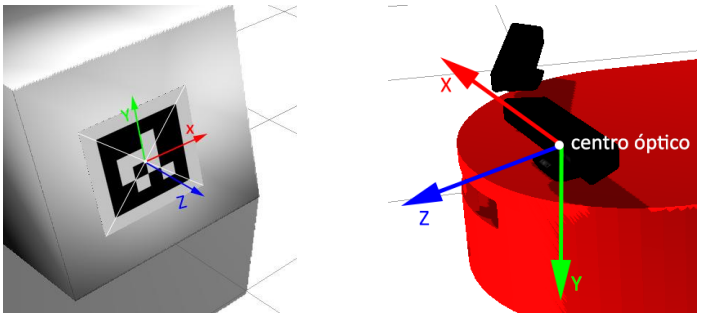

Figura 30 - Sistemas de referencia de la baliza y de la cámara

El paso siguiente es convertir estos dos vectores en una matriz de rotación y traslación, para lo que se hace lo siguiente: con la función *Rodrigues* de OpenCV, se obtiene la matriz de rotación 3x3 desde la representación más compacta (1x3) que ha devuelto *solvePnP*. La matriz de rotación y el vector de traslación se combinan para obtener la matriz 4x4 de rotación y traslación:

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}, \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \rightarrow \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

El resultado es la matriz RT de la pose del marcador con respecto a la cámara y se almacena en una estructura *Matrix4d* de la biblioteca Eigen. Para cambiar al sistema de referencia del marcador no hay más que calcular la inversa de esta matriz:

$$RT_{CámaraMarcador} = RT^{-1}_{MarcadorCámara}$$

La biblioteca Eigen proporciona la función *inverse* para el cálculo de esta inversa, así como facilita el producto de matrices, necesario para el siguiente paso: pasar del sistema de referencia del marcador al sistema de referencia del mundo, operación posible gracias al conocimiento de la pose del marcador con respecto a este último sistema:

$$RT_{MundoCámara} = RT_{MundoMarcador} \cdot RT_{MarcadorCámara}$$

En este punto ya se dispone de la estimación de la posición y orientación absolutas de la cámara, obtenida a partir de la detección de un marcador. La tabla siguiente muestra un extracto del código fuente donde se lleva a cabo la parte fundamental del algoritmo.

```
//Esquinas detectadas en coordenadas de imagen
cv::Mat imgPoints(4,2,CV_32FC1);
...

//Resolución PnP
cv::Mat rvec, tvec;
rvec.create(3,1,CV_32FC1);
tvec.create(3,1,CV_32FC1);
```

```
cv::Mat raux, taux;
cv::solvePnP(m_MarkerPoints, imgPoints, m_CameraMatrix,
             m_DistortionCoeffs, raux, taux);
raux.convertTo(rvec, CV_32F);
taux.convertTo(tvec, CV_32F);

//Paso a matriz de rotación-traslación
cv::Mat R;
cv::Rodrigues(rvec, R);
Eigen::Matrix4d RT_CM; //RT marcador con respecto a la cámara
RT_CM(0,0)=R.at<float>(0,0);
RT_CM(0,1)=R.at<float>(0,1);
RT_CM(0,2)=R.at<float>(0,2);
RT_CM(0,3)=tvec.at<float>(0,0);
RT_CM(1,0)=R.at<float>(1,0);
RT_CM(1,1)=R.at<float>(1,1);
RT_CM(1,2)=R.at<float>(1,2);
RT_CM(1,3)=tvec.at<float>(1,0);
RT_CM(2,0)=R.at<float>(2,0);
RT_CM(2,1)=R.at<float>(2,1);
RT_CM(2,2)=R.at<float>(2,2);
RT_CM(2,3)=tvec.at<float>(2,0);
RT_CM(3,0)=0;
RT_CM(3,1)=0;
RT_CM(3,2)=0;
RT_CM(3,3)=1;

//RT cámara con respecto al marcador
Eigen::Matrix4d RT_MC = RT_CM.inverse().eval();

//RT cámara con respecto al mundo
Eigen::Matrix4d RT_WC = MARKERS.at(iter->id)->GetWorldRT()*RT_MC;

...
```

Además del algoritmo de autolocalización en sí, la clase *CameraManager* proporciona varios métodos que resultan de utilidad para funcionalidades adicionales:

- ***GetEstimatedProjectedPoint***: obtiene la proyección en la cámara estimada de un punto 3D referenciado al sistema de coordenadas del mundo, devolviendo las correspondientes coordenadas 2D en el sistema de referencia de la imagen.
- ***DrawEstimatedPoint***: dibuja sobre la imagen, con un color dado, la proyección de un punto del mundo sobre la cámara estimada.
- ***GetRealProjectedPoint***: obtiene la proyección en la cámara real de un punto 3D referenciado al sistema de coordenadas del mundo, devolviendo las correspondientes coordenadas 2D en el sistema de referencia de la imagen.
- ***DrawRealPoint***: dibuja sobre la imagen, con un color dado, la proyección de un punto del mundo sobre la cámara real.
- ***DrawProjectedRealMarker***: haciendo uso de las funciones "reales" anteriores, dibuja sobre la imagen la proyección del borde del marcador sobre la cámara real.

Las dos primeras funciones, referentes a la cámara estimada, se utilizan para poder realizar una pequeña prueba de realidad aumentada, permitiendo el pintado sobre la imagen de un objeto virtual, como se explica más adelante. Por su parte, las funciones referentes a la cámara verdadera son de utilidad en el estudio de precisión en entorno real.

5.4 Fusión de estimaciones

En una imagen pueden coincidir más de una baliza al mismo tiempo, por lo que el paso siguiente consiste en primer lugar en realizar la fusión de todas las estimaciones obtenidas en la misma. En segundo lugar, existe la posibilidad de aplicar una fusión temporal a las estimaciones obtenidas, lo que quiere decir que estimaciones pasadas puedan tener influencia en la estimación actual

5.4.1 Fusión espacial

La fusión espacial de información implementada consiste en una media ponderada de las coordenadas y ángulos de todas las estimaciones, las cuales se traducen en objetos de clase *Pose* a los que se les asigna un cierto peso (mayor cuanto más cerca está el marcador correspondiente) y que se almacenan en un *array* para su proceso posterior. La clase *Pose* permite acceder tanto a la matriz RT como a las coordenadas y ángulos *roll*, *pitch*, *yaw* correspondientes.

El cálculo de la media ponderada, para lo que se recorre el *array* de poses comentado, es directo en el caso de las coordenadas espaciales; sin embargo, el caso de los ángulos requiere de un tratamiento especial al ser éstos valores "circulares" (por ejemplo, la media entre 358º y 2º es 0º, no 180º). Para este cálculo lo que se ha realizado es la arcotangente de la suma de los senos del ángulo correspondiente dividida entre la suma de los cosenos.

$$ratio_i = \frac{peso_i}{peso_{total}} \tag{5.1}$$

$$[x, y, z]_{fusión} = \sum([x_i, y_i, z_i] \cdot ratio_i) \tag{5.2}$$

$$\alpha_{fusión} = atan\left(\frac{\sum(sen(\alpha_i) \cdot ratio_i)}{\sum(\cos(\alpha_i) \cdot ratio_i)}\right) \tag{5.3}$$

Por lo tanto, después del tratamiento de la información proporcionada por las detecciones de los diferentes marcadores presentes en la imagen, finalmente se dispone

de una estimación de la posición y orientación instantáneas de la cámara con respecto al sistema de referencia 3D que se haya elegido.

5.4.2 Fusión temporal

Además de la fusión espacial instantánea recién comentada, existe la posibilidad de realizar una fusión temporal entre estimaciones, haciendo de este modo que estimaciones pasadas tengan influencia en el cálculo de la actual. Para ello se han desarrollado dos técnicas: por un lado un filtro de Kalman y por otro una segunda media ponderada.

Filtro de Kalman

Una de las maneras de realizar la fusión temporal de información es mediante un filtro de Kalman, que es lo que implementa la clase llamada *KalmanFilter*.

```
class KalmanFilter
{
    private:
        double m_Dt;
        Eigen::VectorXd m_Xhat;
        Eigen::MatrixXd m_A;
        Eigen::MatrixXd m_H;
        Eigen::MatrixXd m_Q;
        Eigen::MatrixXd m_R;
        Eigen::MatrixXd m_P;
        Eigen::MatrixXd m_I;

    public:
        KalmanFilter();
        virtual ~KalmanFilter();

        void Reset();
        Pose GetFilteredPose(const Pose& zpose);
};
```

El estado del filtro implementado es un vector de 12 posiciones que incluye las coordenadas y velocidades tanto espaciales como angulares.

$$s = (x, y, z, roll, pitch, yaw, vx, vy, vz, wroll, wpitch, wyaw)$$

Por tanto, la matriz A, que relaciona el estado anterior con el siguiente es la que se presenta a continuación, donde t es el diferencial de tiempo (el tiempo entre medida y medida):

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & t & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & t & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & t & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & t & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & t & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

La medida se corresponde con la estimación realizada por la clase *CameraManager*.

$$z = (x, y, z, roll, pitch, yaw)$$

Por lo que la matriz H, que relaciona el estado con la medida, es la siguiente:

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Lo que se busca con la inclusión del filtro de Kalman es la atenuación del efecto provocado por la posible variabilidad de la detección 2D de las balizas: al encontrarnos en un entorno discreto (la imagen, de píxeles cuadrados), la variación de un solo píxel en la detección de una esquina provoca un salto de un cierto valor en la estimación de la posición y la orientación. Esta variación de un píxel arriba o abajo es totalmente normal en la detección de balizas *AprilTags*, pero también podría darse el caso de detecciones esporádicas erróneas con lo que el salto sería más acusado. Es por ello que los valores de covarianza del ruido del proceso y del ruido de la medida se deben buscar de tal forma que consigan suavizar el efecto comentado sin que el seguimiento de la estimación se vea retardado. Para conseguir el valor óptimo de estos dos parámetros se han realizado diversas pruebas realizando diferentes combinaciones de los mismos.

La clase dispone de una función llamada *GetFilteredPose* a la que se le pasa la estimación actual (la medida) en forma de objeto de clase *Pose*, devolviendo la estimación filtrada en el mismo formato.

Media ponderada

La segunda forma de fusión temporal que se ha implementado ha sido una media ponderada donde se aplican mayores pesos a las estimaciones más cercanas en el tiempo. La clase *WeightedAverageFilter* implementa un filtro de estas características, con lo que contiene un buffer cuyo tamaño se puede especificar en el constructor. Este

tamaño determina el número de estimaciones pasadas que se consideran, en este caso cinco. A la función *GetFilteredPose* se le pasa la estimación de pose actual y devuelve la filtrada, calculando la media de ángulos y coordenadas de la misma manera que en el caso de fusión espacial, pero ahora con los pesos ajustados según la "frescura" de las estimaciones.

```
class WeightedAverageFilter
{
    private:
        unsigned int m_Size;
        std::vector<Pose, Eigen::aligned_allocator<Pose> > m_Poses;
        std::vector<double> m_Weights;
        std::vector<double> m_Sumatories;

    public:
        WeightedAverageFilter(unsigned int size);
        virtual ~WeightedAverageFilter();

        void Reset();
        Pose GetFilteredPose(const Pose& zpose);
};
```

5.5 Interfaz gráfico de usuario

Como se ha comentado, la aplicación dispone de un interfaz con el que el usuario puede interaccionar para realizar diversas funciones, entre ellas la observación de la imagen capturada y de la representación de la cámara estimada en el mundo virtual 3D.

5.5.1 Utilidades de pintado

Las clases *World* y *DrawWorld* son las encargadas de gestionar el pintado del mundo virtual 3D sobre una ventana OpenGL. Además permiten la interacción de la ventana con el ratón para poder desplazarse por el mundo. Para facilitar este dibujado se ha implementado la clase *DrawingUtils*, que proporciona un conjunto de funciones estáticas. Son las siguientes:

- ***DrawLine***: dibuja una línea recta entre dos puntos dados, con un grosor y un color determinados.
- ***DrawSphere***: dibuja una esfera en un punto dado, con un grosor y un color determinados.
- ***DrawAxis***: dibuja los ejes determinados por un origen y tres puntos de referencia, con un grosor determinado.
- ***DrawCamera***: dados los parámetros intrínsecos y extrínsecos de una cámara, dibuja una representación de la misma, con aspecto de pirámide con la punta cortada, ya que se realiza de la siguiente manera:

- Se calcula el rayo de retroproyección del centro óptico y se dibuja un segmento del mismo de dos metros de longitud desde el foco.
- Se calculan los rayos de retroproyección de las cuatro esquinas de la imagen.
- Se calculan el plano imagen y el plano paralelo a éste a una distancia de un metro.
- Se calculan los ocho puntos de corte de estos planos con los rayos de retroproyección de las esquinas.
- Se dibujan los segmentos de los rayos de retroproyección contenidos entre los cuatro puntos de corte de un plano y los cuatro del otro, con lo que quedan conformadas las aristas de la pirámide.
- Se dibujan los segmentos paralelos dos a dos que unen los puntos de corte y que están contenidos en los planos.

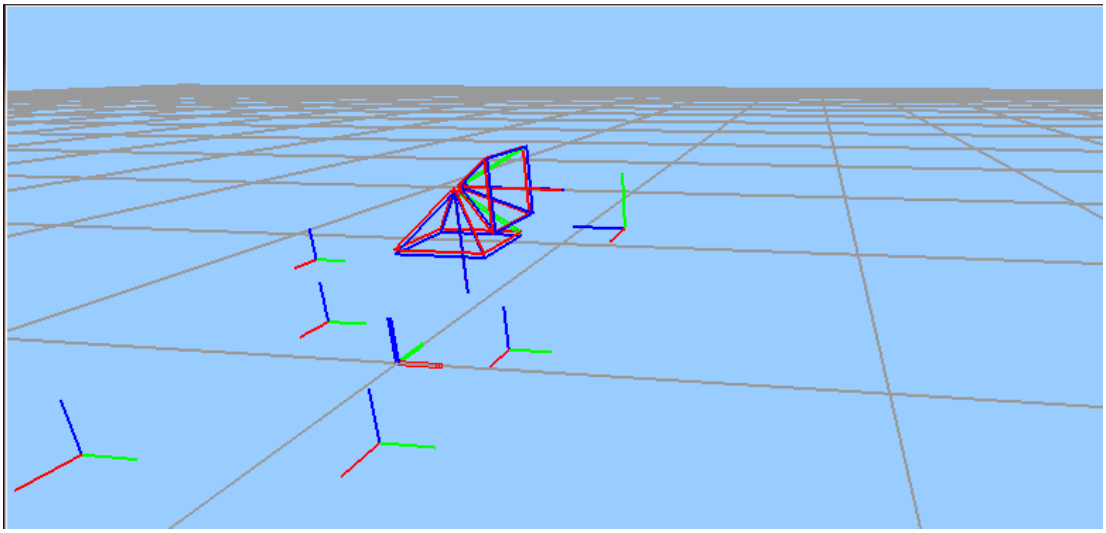

Figura 31 - Representación de cámaras estimada (azul) y verdadera (rojo)

En la figura anterior se puede ver una captura de la ventana del mundo virtual donde se presenta de manera gráfica la información 3D correspondiente. En este ejemplo se ven representadas, de la manera explicada en el punto anterior (con el eje óptico y la pirámide de visión) las dos cámaras del cuadricóptero virtual. La cámara verdadera se pinta en rojo mientras que la estimada se pinta en azul. Además, en el mundo virtual 3D se representan las balizas de la escena dibujando sus sistemas de referencia: el eje x en rojo, el eje y en verde y el eje z en azul. La rejilla gris del suelo representa el plano xy, es decir el plano del suelo, mientras que los ejes más gruesos que se pueden ver en una de sus intersecciones representan el sistema de referencia del mundo.

5.5.2 Menú de usuario

El aspecto que presenta la ventana principal de la aplicación es la que se puede observar en la figura siguiente:

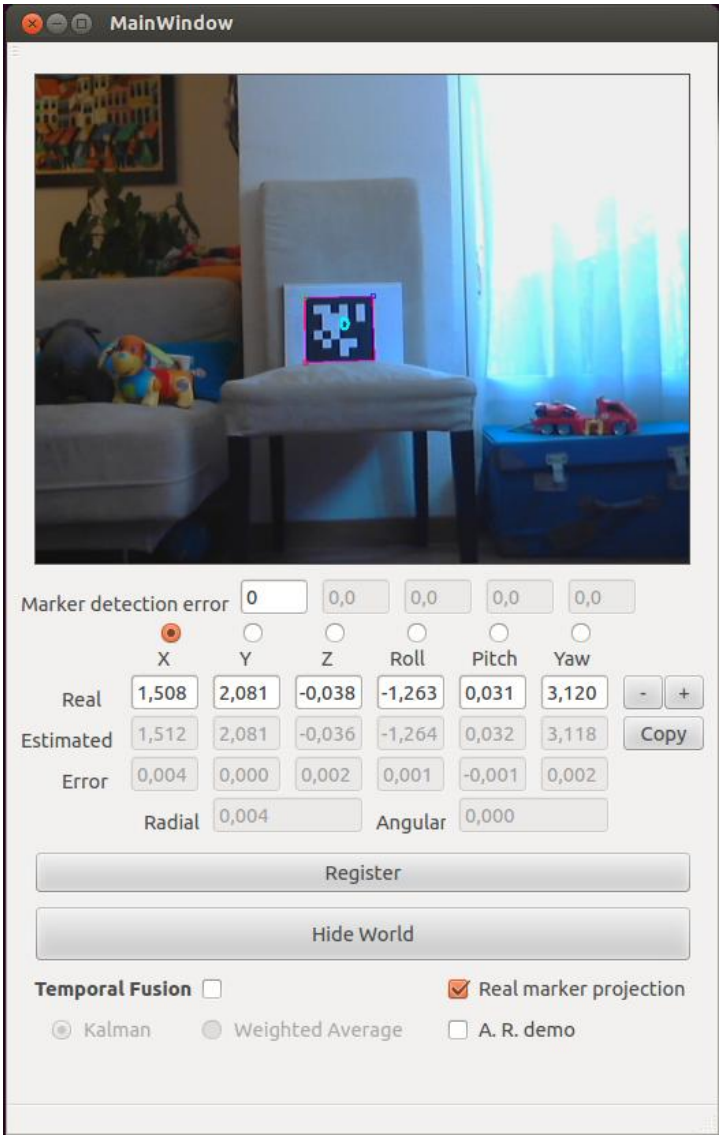

Figura 32 - Interfaz gráfica de usuario

La parte superior está dedicada a presentar la imagen capturada sobre la que se superpone una representación de las balizas detectadas, recuadrándolas en color y marcando sus vértices.

Por otra parte, la zona central de la ventana contiene la información referente a la posición y orientación de la cámara, tanto verdadera como estimada, además de la diferencia entre una y otra. La información verdadera se puede introducir a mano si se está trabajando en un entorno real, ya que a priori no se dispone de ésta como puede pasar en el entorno simulado.

En la zona inferior izquierda el menú permite la activación de la fusión temporal, así como la selección de entre las dos posibles técnicas, mientras que la parte inferior derecha permite activar dos funcionalidades adicionales. La primera es una pequeña demostración de realidad aumentada que al activarse proyecta sobre la imagen los puntos 3D contenidos en un fichero llamado *object.txt*, según la pose estimada en cada momento.

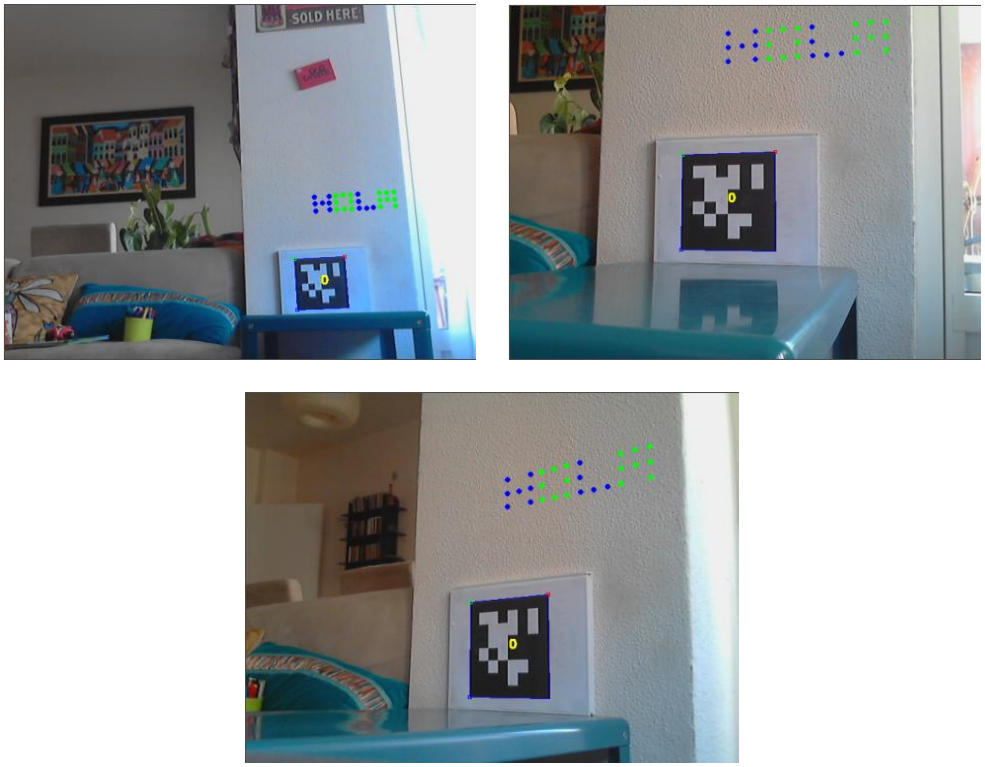

Figura 33 - Ejemplo de realidad aumentada en cam_autoloc

En la figura anterior se muestran varios momentos de una ejecución con la demostración activada, en la que se proyecta sobre la imagen un cartel de bienvenida supuestamente colocado sobre la pared. Se puede observar cómo la posición, orientación y tamaño del cartel se mantienen coherentes a lo largo de las distintas ubicaciones tomadas por la cámara.

La segunda funcionalidad adicional pinta sobre la imagen la proyección de la baliza indicada según la pose verdadera introducida, lo que es de utilidad para el estudio de precisión en entorno real, tal y como se explica en el capítulo siguiente.

Capítulo 6

Estudio de precisión

Para evaluar la precisión del algoritmo de autolocalización desarrollado se han llevado a cabo una serie de experimentos en los que se ha registrado la información verdadera junto con la estimada para posteriormente analizarla y realizar una comparativa. Los datos de comparación han sido el error radial y el error angular. El primero se define como la distancia euclídea existente entre la posición estimada y la verdadera, mientras que el segundo es una combinación de los errores en los tres posibles ángulos y se ha definido como la raíz cuadrada de la suma de las diferencias al cuadrado de *roll*, *pitch* y *yaw*. La convención utilizada para la definición de estos tres ángulos es la siguiente: *roll* es un giro alrededor del eje x de la baliza, *pitch* alrededor del eje y y *yaw* alrededor del eje z.

Los experimentos se han realizado sin hacer uso de la fusión temporal, en primer lugar con el cuadricóptero volador *ArDrone* en el entorno virtual de Gazebo, que dispone de una cámara ventral y otra frontal, y después en un entorno real con una cámara de videoconferencia.

6.1 Experimentos en entorno virtual

Para la primera parte de los experimentos se ha trabajado con el simulador robótico Gazebo, en el que se dispone de la información verdadera de posición y orientación de la cámara gracias a uno de los *plugins* para el mismo desarrollados por Daniel Yagüe [Yagüe, 2015]. Los datos de posición están referenciados al origen del mundo en Gazebo, el mismo que se utiliza en la aplicación, pero los ángulos *roll*, *pitch* y *yaw* que se leen del *plugin* están expresados como ángulos de navegación del drone, por lo que hay que transformarlos para expresarlos como ángulos de orientación con respecto al

sistema de referencia absoluto: el ángulo de *yaw* está desfasado 90º, el de *pitch* se corresponde con el de *roll*, mientras que el de *roll* es el pitch desfasado 180º.

Además, la ubicación recibida del simulador es la del propio cuadricóptero, no la de sus cámaras, por lo que, con la matriz RT correspondiente a la información anterior y con la que determina la ubicación relativa de la cámara en el drone, se calcula la matriz RT deseada:

$$RT_{MundoCámara} = RT_{MundoDrone} \cdot RT_{DroneCámara}$$

En este ámbito cada prueba ha consistido en hacer volar de distinta manera el cuadricóptero por una escena determinada (con distinto número y posición de las balizas), fijando algunos de los parámetros de ubicación de la cámara (x, y, z, *roll*, *pitch* y *yaw*) y variando otros, recogiendo el error tanto en distancia como en ángulo.

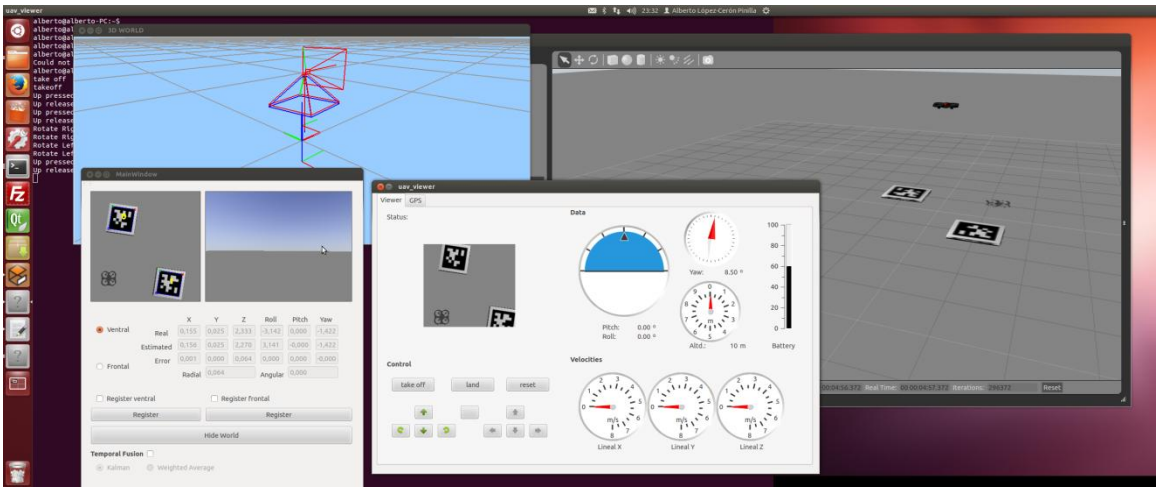

Figura 34 - Escenario de pruebas en entorno virtual

6.1.1 Pruebas de ángulo *yaw* y distancia

El objetivo de estas pruebas es conocer la dependencia del error cometido con la distancia y el *yaw* existentes con respecto al marcador. También es importante observar la variación del error según se utilicen distinto número de marcadores y en distintas posiciones, para lo que se han realizado experimentos con uno, dos, cuatro y cinco marcadores, estando todos en el mismo plano (en el suelo y paralelas a él) en los tres primeros casos y cuatro balizas en un plano y otra en un plano perpendicular en el último. En todos los casos se ha utilizado la cámara ventral y la prueba ha consistido en colocar el cuadricóptero sobre las balizas e ir alejándolo y acercándolo con distintas orientaciones, girándolo sobre su eje vertical hasta dar una vuelta completa.

Una baliza

En este primer caso se ha utilizado una única baliza observada por la cámara en el centro de la imagen.

Figura 35 - Baliza a dos distancias y a dos ángulos *yaw* distintos

Las siguientes gráficas muestran el error radial y el error angular frente a la distancia y frente al ángulo al mismo tiempo.

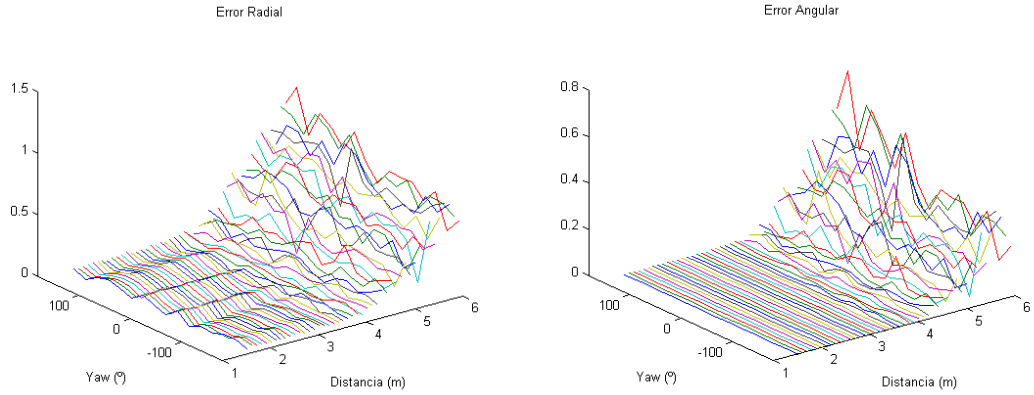

Figura 36 - Errores frente a distancia y *yaw*, 1 baliza

Lo primero que salta a la vista en la Figura 36 es cómo se degrada la estimación a partir de una determinada distancia (unos 4 metros), lo que ocurre tanto para el error radial como para el angular e independientemente de la orientación. Se observa que el error aumenta según aumenta la distancia, pero suavemente en un primer momento (de 1 a 4 metros el error radial medio se mantiene por debajo de 10 cm y el angular medio por debajo de 0.02º), hasta que la estimación se degrada completamente.

La dependencia con al ángulo *yaw* es menor, sobre todo en el error angular, manteniéndose el valor de error, hasta los 4 metros comentados, en cotas similares en todo el rango de posibles ángulos (±180º ya que en este caso se puede dar una vuelta completa al cuadricóptero y se detecta la baliza constantemente).

Dos balizas

Como se ha comentado, una cuestión interesante es saber si utilizando más balizas la estimación mejora. En este experimento se usan dos de ellas en disposición diagonal, de forma que la cámara las observa en las esquinas de la imagen cuando se encuentra cerca de ellas.

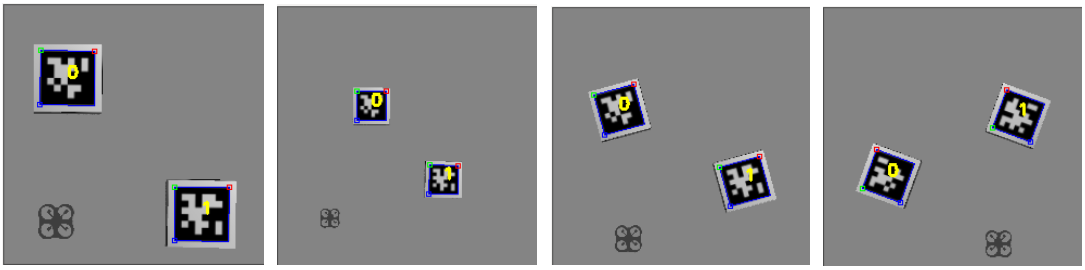

Figura 37 - Dos balizas a dos distancias y a dos ángulos *yaw* distintos

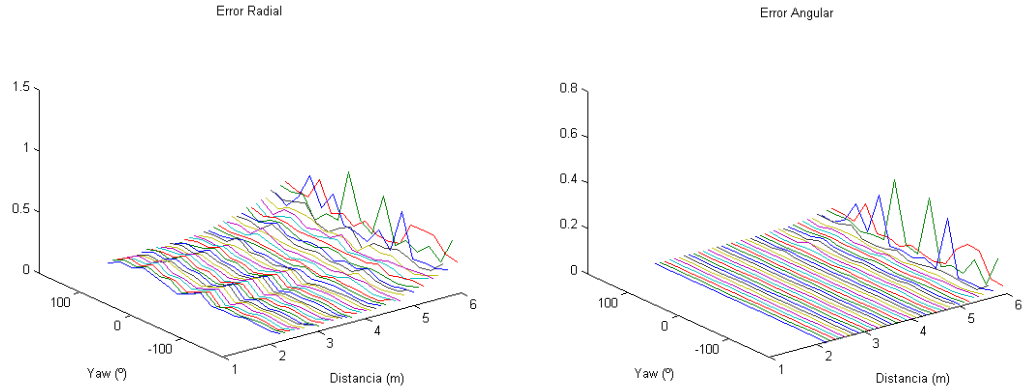

Figura 38 - Errores frente a distancia y *yaw*, 2 balizas

Al analizar las gráficas de la Figura 38, las conclusiones a las que se puede llegar con respecto a la dependencia con la distancia y el *yaw* son equivalentes a las del caso anterior (incremento del error con la distancia y poca dependencia de éste con el ángulo), si bien es cierto que el análisis de la influencia del número de balizas arroja unos datos interesantes: en primer lugar, el rango del error ha bajado tanto en el radial como en el angular, y en segundo, la distancia a la que se produce el salto de degradación es mayor.

El error radial medio ronda los 5 cm hasta los 4 m de distancia y no supera los 10 cm hasta los 5 m (a esa primera distancia el error medio del caso anterior era de 10 cm, mientras que a la segunda subía hasta los 40 cm). Por otra parte, el error angular medio no supera los 0.02º hasta los 5 m de distancia. Es decir, con una sola baliza la estimación se degradaba considerablemente a partir de 4 m, mientras que con dos marcadores, a los 5 m el error todavía se mantiene en cotas aceptables.

Cuatro balizas

En este caso son cuatro las balizas utilizadas, todas en el mismo plano y con sus bordes paralelos.

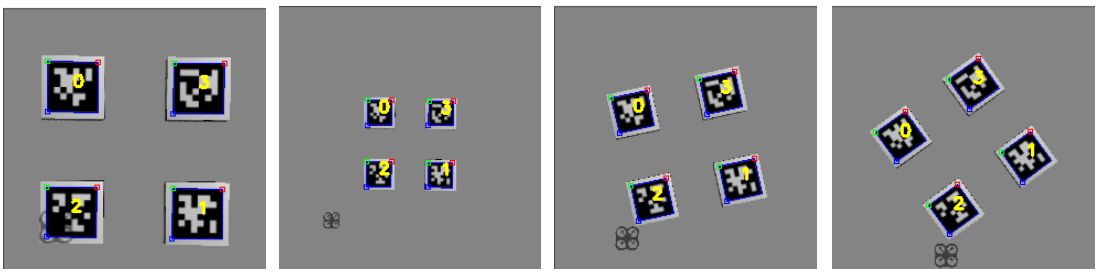

Figura 39 - Cuatro balizas a dos distancias y a dos ángulos *yaw* distintos

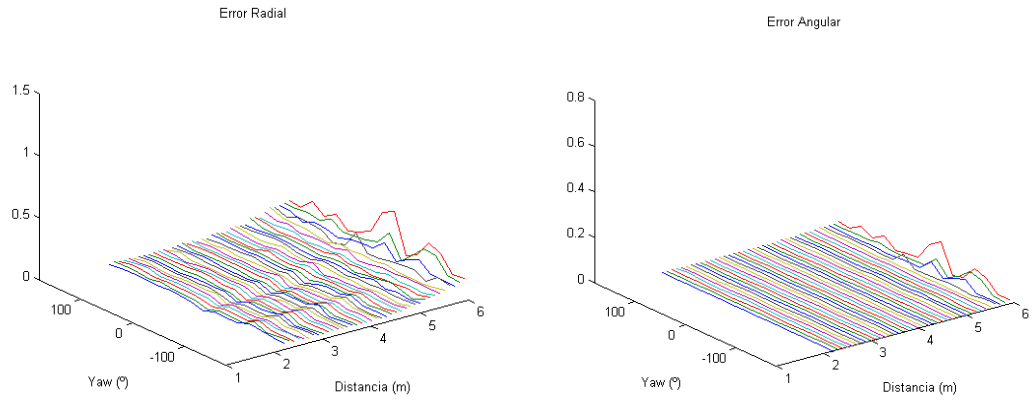

Figura 40 - Errores frente a distancia y *yaw*, 4 balizas

De nuevo se observa algo parecido a lo ya comentado, disminuyendo aún más el error y suavizándose el salto de degradación. La mejora con respecto al caso de dos balizas se nota sobre todo a partir de 4,5 m, ya que hasta esa distancia los valores de error son parecidos. A una distancia de 5 metros el error radial medio en este caso es de unos 8 cm (cuando en el caso anterior ya empezaba a superar los 10), mientras que el error angular medio es de 0.01º (y con dos balizas se encontraba por 0.02º).

Comparativa de número de balizas

Además de la presentación frente al ángulo y a las distancia a la vez, se ha realizado el análisis del error en relación a cada parámetro independientemente. Por ejemplo, las siguientes gráficas muestran el error cometido frente a la distancia, independientemente del ángulo existente entre la cámara y las balizas, para lo que se ha calculado, en cada distancia, la media del error en todo el rango de *yaw*.

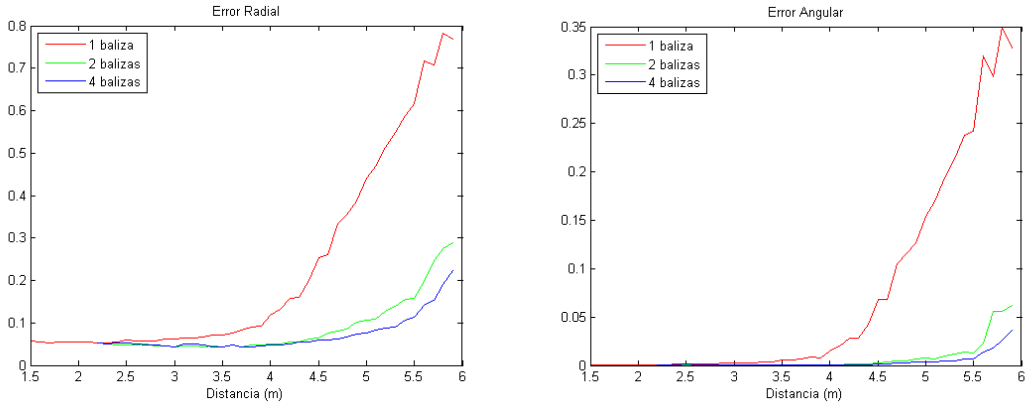

Figura 41 - Errores frente a distancia

En las gráficas de la Figura 41 se aprecia todavía más claramente dos de las observaciones realizadas en los apartados anteriores: el error aumenta con la distancia y disminuye con el número de balizas. Esto último se puede apreciar también en las gráficas de error frente a *yaw*, donde, análogamente al caso anterior, se ha calculado para cada ángulo *yaw*, la media del error en todo el rango de distancias.

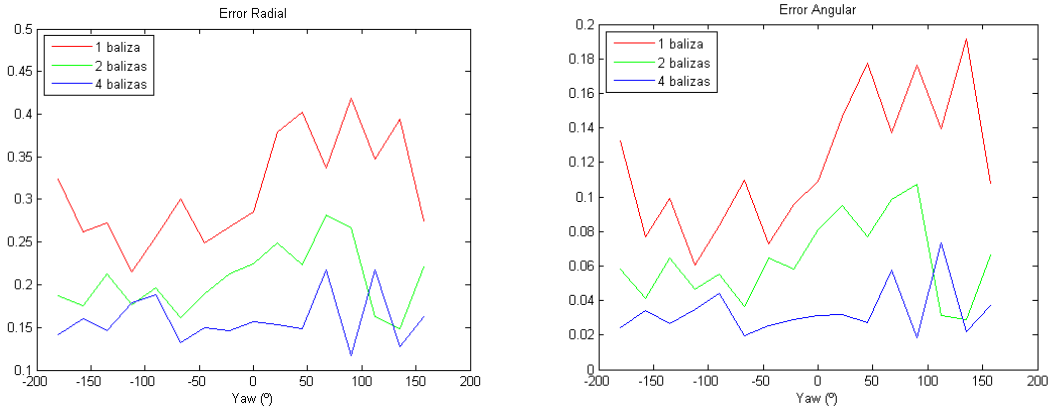

Figura 42 - Errores frente a *yaw*

Cuatro balizas en un plano y una baliza perpendicular

En este experimento se ha querido observar la influencia de la colocación de una baliza en un plano perpendicular al plano en que se encuentran las demás. Para ello, en primer lugar se ha vuelto a realizar la medición con cuatro balizas, pero esta vez no paralelas al suelo, ya que en ese caso, la baliza adicional queda totalmente perpendicular y no se puede ver desde la cámara ventral del drone.

Figura 43 - Cuatro balizas inclinadas a dos distancias y a dos ángulos *yaw* distintos

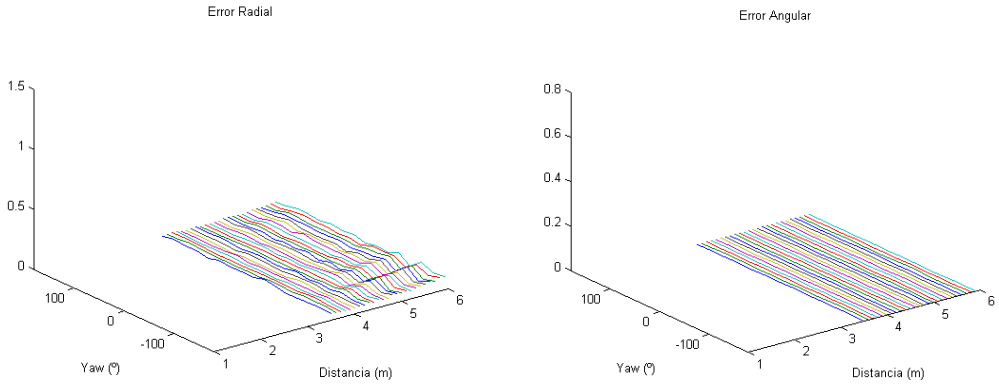

Figura 44 - Errores frente a distancia y *yaw*, 4 balizas no paralelas al suelo

Comparando la Figura 44 con la Figura 40, ambas correspondientes al caso de 4 balizas, pero inclinadas en un caso y paralelas al suelo en el otro, se puede apreciar una mejora en las distancias más altas, no apareciendo picos de error, por lo que se puede llegar a pensar que la inclinación de las balizas afecta favorablemente a la estimación. Esta dependencia se estudia en apartados posteriores.

Por tanto, en este caso se observa un comportamiento aceptable en todo el rango de distancias y ángulos estudiados, sólo sobrepasando los 10 cm de error radial medio a los 6 m de distancia, manteniéndose el error angular por debajo de 0.001º.

En segundo lugar, se ha añadido la quinta baliza y se ha vuelto a realizar el experimento.

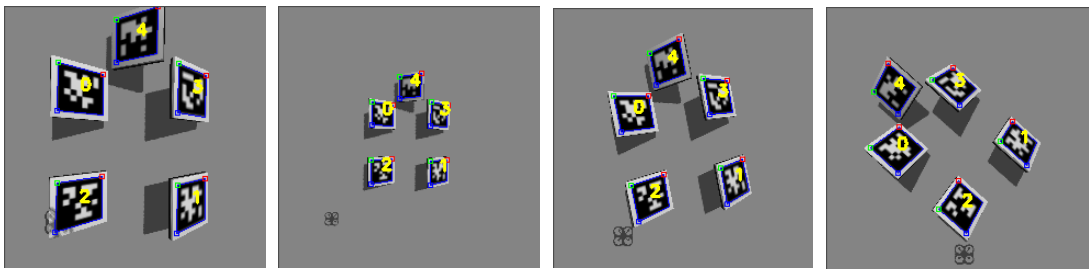

Figura 45 - Cuatro balizas y una perpendicular a dos distancias y a dos ángulos *yaw* distintos

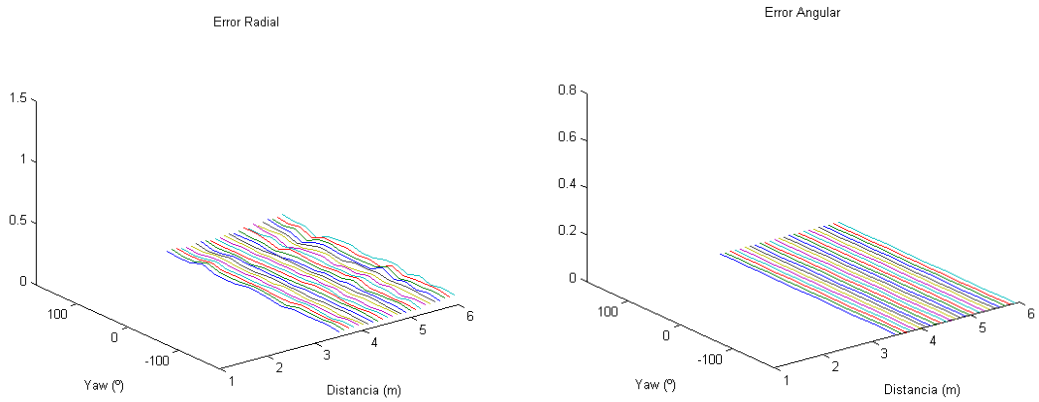

Figura 46 - Errores frente a distancia y *yaw*, 4 balizas coplanares y 1 perpendicular

Inicialmente se observa un comportamiento parecido comparando la Figura 45 y la Figura 46. Para apreciar mejor las diferencias se presentan las figuras siguientes, que comparan directamente los resultados obtenidos en ambos casos, pero por cada parámetro de estudio independientemente:

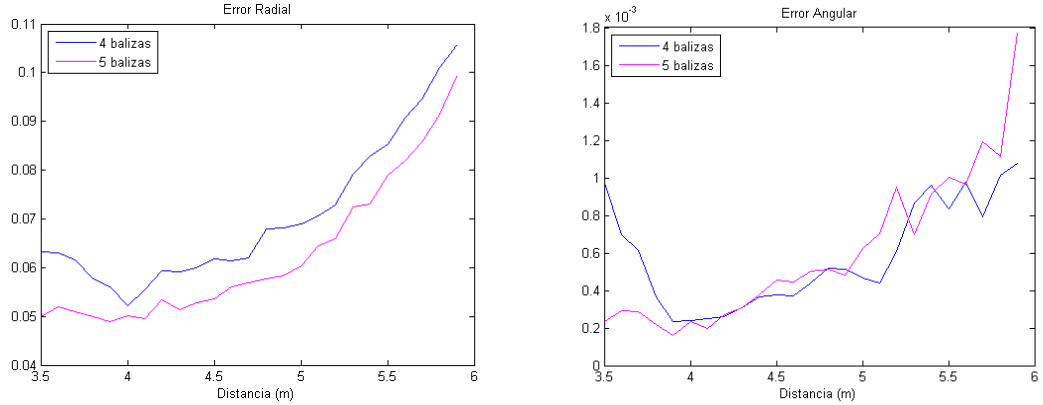

Figura 47 - Errores frente a distancia, 4 y 5 balizas

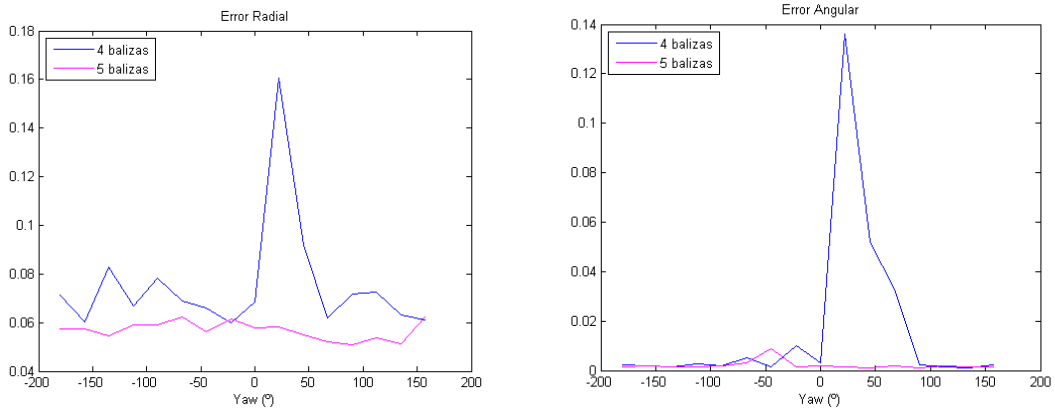

Figura 48 - Errores frente a *yaw*, 4 y 5 balizas

En las comparativas del error radial (parte izquierda de la Figura 47 y de la Figura 48) se comprueba lo observado hasta ahora: el valor es menor utilizando la quinta baliza, lo

que se aprecia tanto frente a la distancia como frente al ángulo *yaw*. El error angular parece equivalente en ambos casos, aunque en cada una de las dos gráficas se observa un aspecto digno de comentar. Frente a la distancia (Figura 47 derecha), el error angular con cinco balizas comienza por debajo del error con cuatro, pero a partir de 5 m esta tendencia sorprendentemente parece invertirse. Por otra parte, el error angular frente al *yaw* (Figura 48 derecha) toma valores muy parecidos en ambos casos, menos en el rango de 0º a 100º, donde el valor para cuatro balizas presenta un pico considerable. Esto puede tener su explicación en una mala detección puntual de las esquinas de una o varias de las balizas.

Otro aspecto a considerar al incluir una baliza perpendicular es separar el error radial en error en XY y en error en Z.

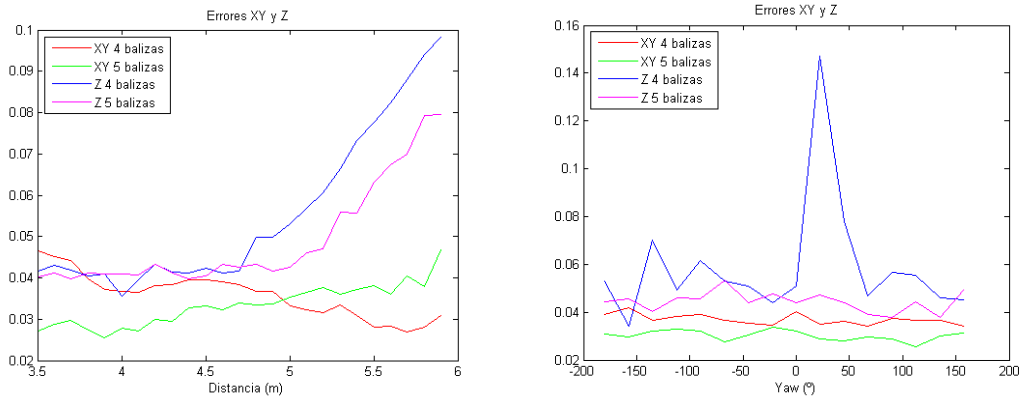

Figura 49 - Error XY y Error Z frente a distancia y *yaw*, 4 y 5 balizas

Lo primero que se aprecia en la parte izquierda de la Figura 49 es que lo que más afecta al aumento del error radial con la distancia es el error en Z, manteniéndose el error en XY en una media de unos 4 cm en todo el rango de distancias estudiadas, mientras que el error en Z empieza a subir considerablemente a partir de 4,5 m. Y en segundo lugar, se puede observar cómo la entrada de la quinta baliza afecta favorablemente al error en Z, manteniéndose en valores inferiores en el rango de distancias más altas.

Con respecto a los errores frente al ángulo *yaw*, presentados en la parte derecha de la figura anterior, también se aprecia la mejora con el marcador perpendicular, siendo los errores (tanto el XY como el Z) menores en todo el rango de ángulos estudiados.

6.1.2 Pruebas de ángulo *pitch* y distancia

En este caso se pretende observar cómo se comporta la estimación si se varía la distancia a la baliza y el ángulo *pitch* existente entre ésta y la cámara. Aumentar o disminuir este ángulo desde 0 hace que la cámara vea la baliza cada vez más escorada.

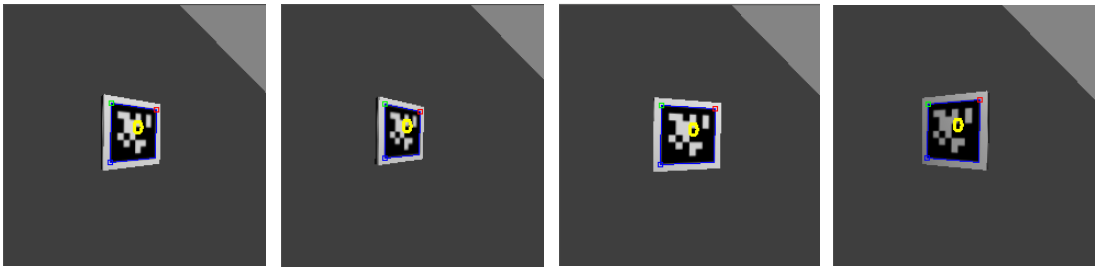

Figura 50 - Baliza a diferentes ángulos *pitch*

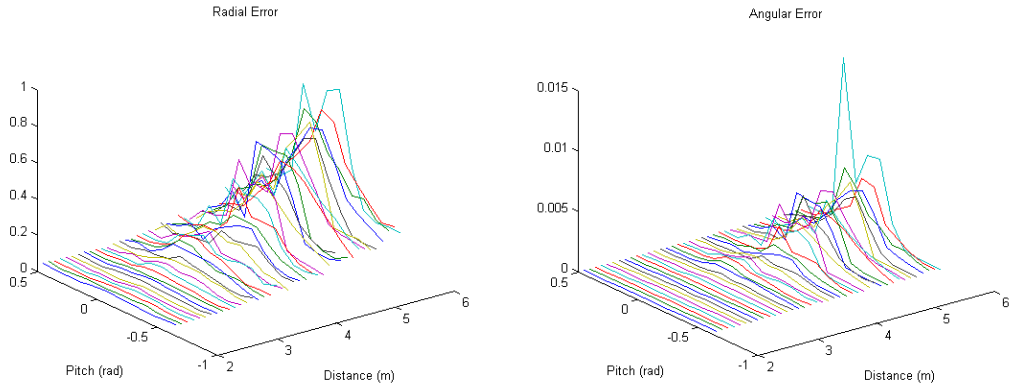

Figura 51 - Errores frente a distancia y *pitch*

Al contrario que en el caso explicado en el apartado anterior, aquí no se ha desplazado el cuadricóptero, sino que se ha colocado una baliza y se ha ido modificando su orientación, ya que de esta manera resulta más cómodo el control del ángulo deseado. En las gráficas de la Figura 51 ya se aprceia la dependencia con la distancia observada en el caso anterior, lo que se puede comprobar en las gráficas siguientes.

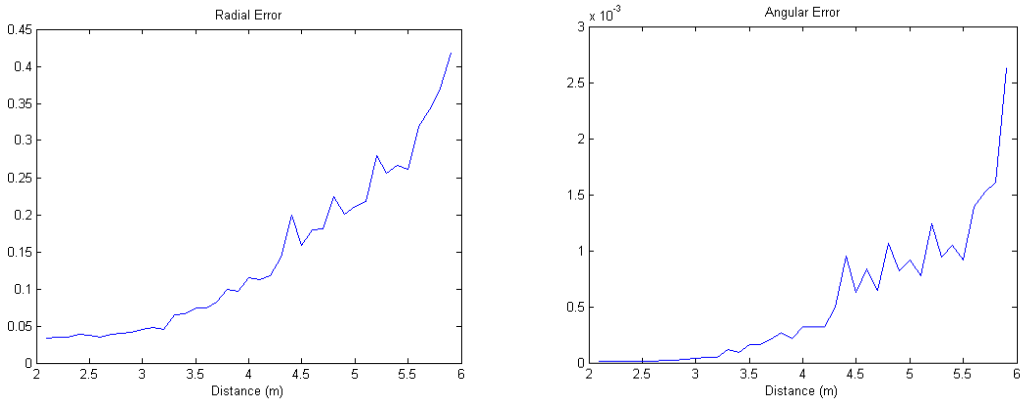

Figura 52 - Errores frente a distancia

Sin embargo, en las gráficas bidimensionales también se aprecia una diferencia con el caso del ángulo *yaw* y es que el aumento del error no es uniforme para todo el rango de valores de *pitch*. Esto se aprecia más claramente en las siguientes gráficas.

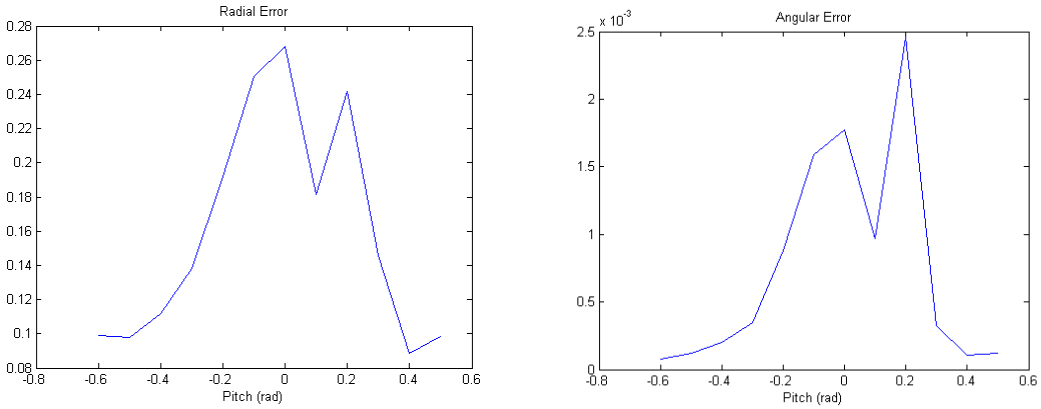

Figura 53 - Errores frente a *pitch*

Lo que se extrae de estas gráficas es que el error es mayor cuando el *pitch* es pequeño, es decir, cuando el paralelismo del plano de la baliza y el de imagen es grande. Además, el tener un cierto *pitch* entre la cámara y la baliza atenúa el efecto de la distancia, obteniéndose mejores estimaciones que en el caso de la baliza paralela a la misma distancia. Por tanto, se puede concluir que, a la hora de escoger la orientación de las balizas a utilizar en un sistema, es recomendable el darles una cierta orientación y evitar que la cámara las pueda observar totalmente paralelas a su plano imagen.

6.1.3 Pruebas de ángulo *roll* y distancia

De nuevo, el objetivo es determinar la dependencia de la estimación con la distancia y un determinado giro, en este caso el *roll*. El experimento es parecido al del apartado anterior: se ha colocado una baliza en el entorno virtual de Gazebo y se ha ido modificando su orientación para que la cámara la fuera observando con distintos ángulos de *roll*, esto es, con diferentes inclinaciones (con un *roll* de 0 la baliza se encuentra paralela al plano de imagen de la cámara).

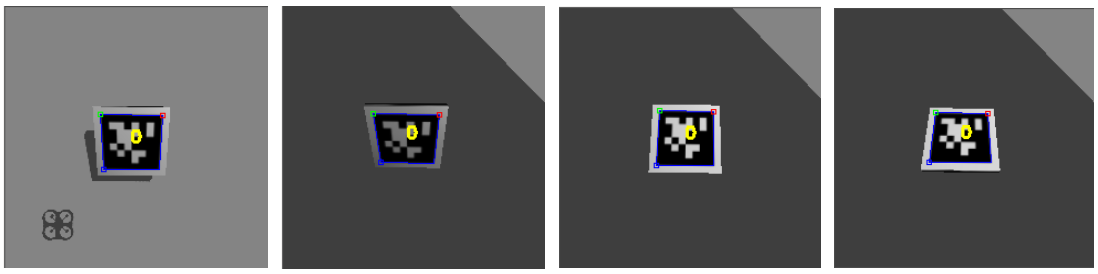

Figura 54 - Baliza a diferentes ángulos *roll*

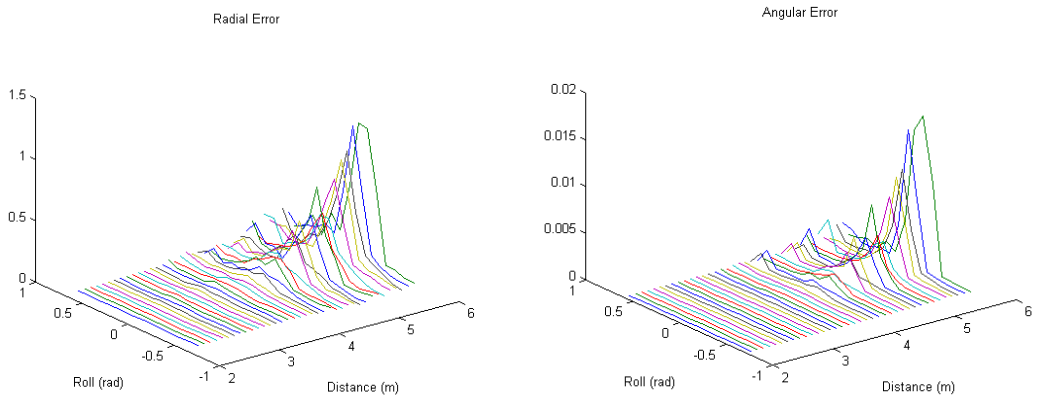

Figura 55 - Errores frente a distancia y *roll*

En las gráficas bidimensionales se observa algo parecido a lo que ocurría en el caso anterior y es que la distancia no afecta tanto a la estimación si la baliza se observa con un cierto *roll*.

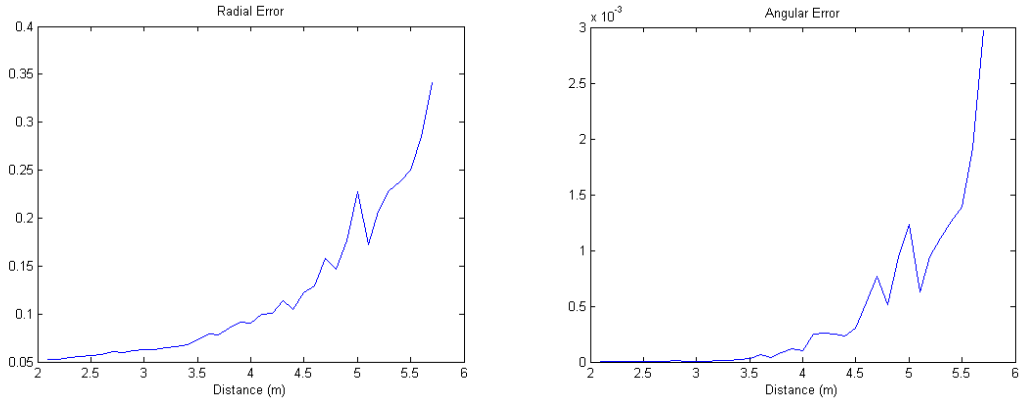

Figura 56 - Errores frente a distancia

Efectivamente el error aumenta con la distancia, como indican las gráficas anteriores, pero en ellas está acumulado el error en todo el rango de *roll*, por lo que no se aprecia la incidencia menor de la distancia a mayores valores de éste. Es en las siguientes gráficas donde se observa perfectamente que el error disminuye según aumenta el valor absoluto del *roll* existente entre la cámara y la baliza.

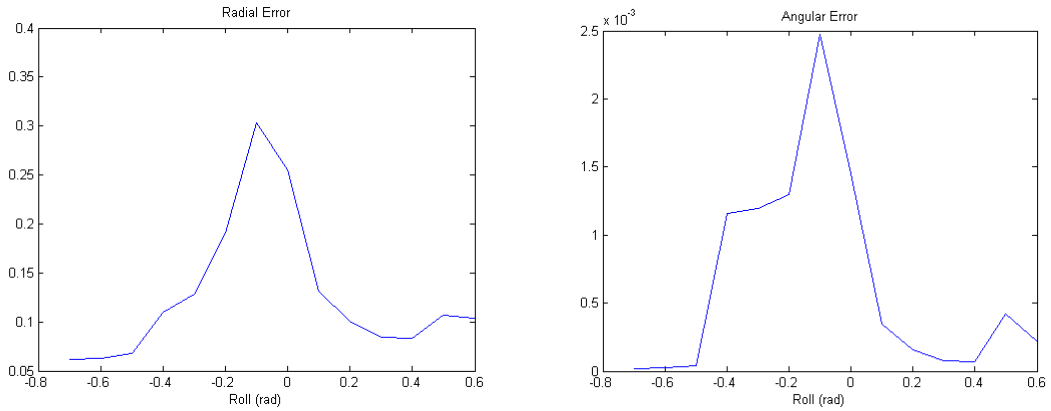

Figura 57 - Errores frente a *roll*

Análogamente a lo que se indicaba en el caso del *pitch*, en este caso se puede concluir que siempre es mejor para la calidad de la estimación que las balizas presenten una cierta inclinación en vez de presentarse paralelas al plano de imagen de la cámara.

6.1.4 Pruebas de desplazamiento en imagen

Las pruebas de desplazamiento pretenden estudiar si la distancia a la que se encuentra la baliza del centro de la imagen tiene influencia en la precisión de la estimación. Por ello se ha dividido la imagen en nueve zonas y se ha ido colocando el cuadricóptero de forma que la baliza cayera en cada una de ellas, registrando veinte medidas de error cada vez.

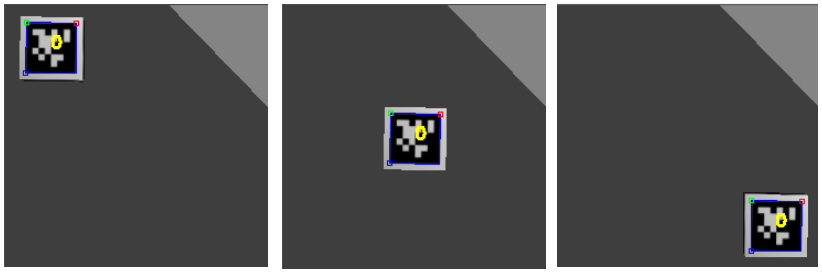

Figura 58 - Baliza en 3 de las 9 posibles posiciones de desplazamiento

La siguiente tabla muestra los valores de error radial en cm obtenidos en cada caso:

Error radial (cm)		
10,93	10,63	10,65
10,45	10,53	10,64
10,25	10,62	10,96

Como se puede observar las diferencias son mínimas, siendo la mayor de 7,1 mm. Tampoco se observan tendencias en los incrementos o decrementos del error que hagan pensar que la estimación es mejor cuando la baliza está centrada o en un rincón de la

imagen. De todas formas, en este caso puede resultar interesante el estudiar el error XY y el Z por separado:

Error XY (mm)		
2,95	11,28	4,98
9,37	6,28	7,40
4,09	5,09	2,13
Error Z (cm)		
10,92	10,56	10,63
10,41	10,51	10,61
10,24	10,60	10,96

En la primera parte de la tabla anterior sí que se aprecia una tendencia significativa: las zonas de la imagen donde el error XY es menor son los cuatro rincones, lo que enlaza con las conclusiones anteriores que indicaban que un cierto *roll* y un cierto *pitch* (el no estar la baliza totalmente centrada) aumenta la calidad de la estimación. Sin embargo, al ser este error mucho menor que el error en Z y ser éste independiente del desplazamiento de la baliza, la tendencia queda enmascarada dentro del error radial.

6.2 Experimentos en el entorno real

Además de en el entorno virtual, se ha llevado a cabo un estudio de precisión con una cámara real, en el que se han realizado experimentos con el fin de corroborar lo observado con respecto a la influencia de la distancia y de los ángulos en la validez de las estimaciones. Para ello se ha fabricado una baliza imprimiendo en un folio uno de los marcadores y se ha utilizado una cámara web.

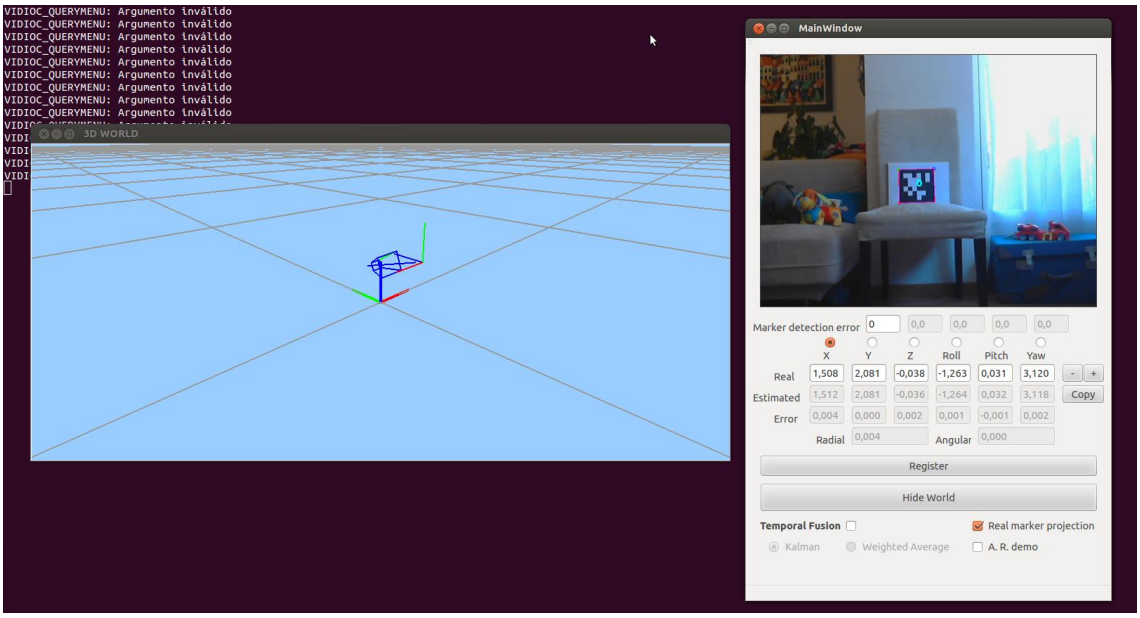

Figura 59 - Escenario de pruebas en entorno real

6.2.1 Estimación de pose verdadera

En el caso real no se dispone a priori de la información verdadera de posición y orientación, por lo que se ha realizado lo comentado en el capítulo anterior para poder contar con ella: se ha añadido a la aplicación la posibilidad de introducir esta información a mano y con ella pintar sobre la imagen la proyección resultante de un marcador, con lo que se toma como verdadera la ubicación que hace que la proyección y el marcador coincidan perfectamente.

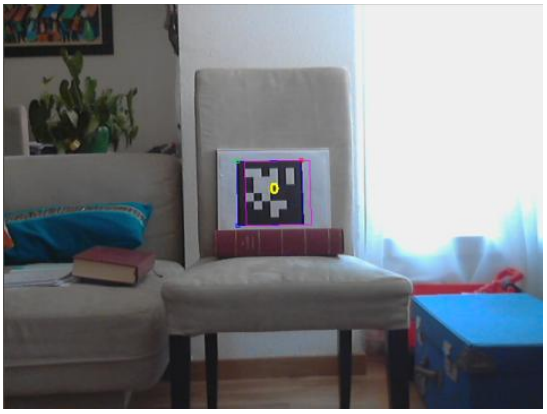

Figura 60 - Proyección según la ubicación verdadera de la cámara introducida manualmente

Esta manera de obtener la posición y orientación verdadera de la cámara no es exacta, por lo que las conclusiones que se puedan sacar hay que tomarlas con ciertas prevenciones. De todas maneras y a pesar de la inexactitud del método, pensamos que pueden servir para ver las tendencias.

6.2.2 Prueba de distancia

El primer experimento ha sido el de distancia, colocando la baliza en posiciones cada vez más cercanas.

Figura 61 - Baliza a dos distancias distintas

El proceso seguido ha sido el siguiente:

1) Se deja la cámara fija frente a la baliza.
2) Se busca la información verdadera de la manera ya explicada, esto es, introduciendo a mano los valores de posición y orientación hasta que la supuesta proyección verdadera coincide con la baliza en la imagen.
3) Se registra varias veces la información de error en ese momento (hasta quince veces por cada posición).
4) Se repite el proceso acercando la baliza visual progresivamente (el rango total de distancias ha sido de 0,4 m hasta 2,2 m, tomando valores en 36 ocasiones).

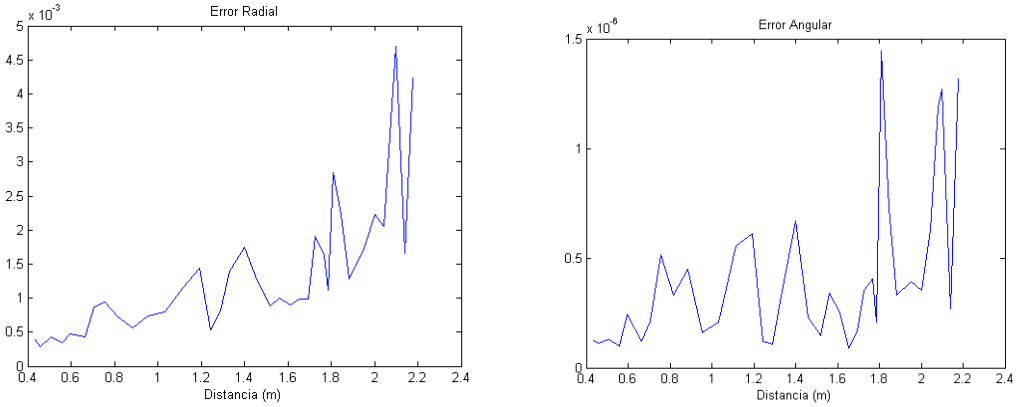

Figura 62 - Errores frente a distancia

En las gráficas de la Figura 62 se puede apreciar, en primer lugar, que el error cometido es pequeño (a la distancia de estudio más alta el radial no supera los 5 mm y el angular los 0,00015º), y en segundo lugar, que se vuelve a observar la tendencia al alza del error con la distancia.

Además de las pruebas a diferente distancia, se ha querido observar la distancia máxima a la que la baliza se sigue detectando, siendo el valor obtenido unos 9 metros.

6.2.3 Prueba de ángulo *yaw*

En este caso el objetivo es determinar cómo influye que la baliza presente un cierto ángulo alrededor de su eje Z, es decir, un cierto ángulo *yaw*, en la precisión de las estimaciones.

Figura 63 - Baliza a dos ángulos *yaw* distintos

El proceso seguido es análogo al explicado en el apartado anterior, habiéndose colocado la baliza a una distancia para ir variando en este caso su orientación correspondientemente, a la vez que se busca manualmente la posición verdadera.

El rango de medición ha sido de -90º a 90º ya que, a raíz de lo observado en los experimentos en el entorno virtual, que señalaban una dependencia baja con el ángulo de estudio, se ha considerado que no era necesario capturar datos dando una vuelta completa a la baliza. En este rango se han realizado 15 mediciones para cada una de las 40 orientaciones en que se han registrado medidas.

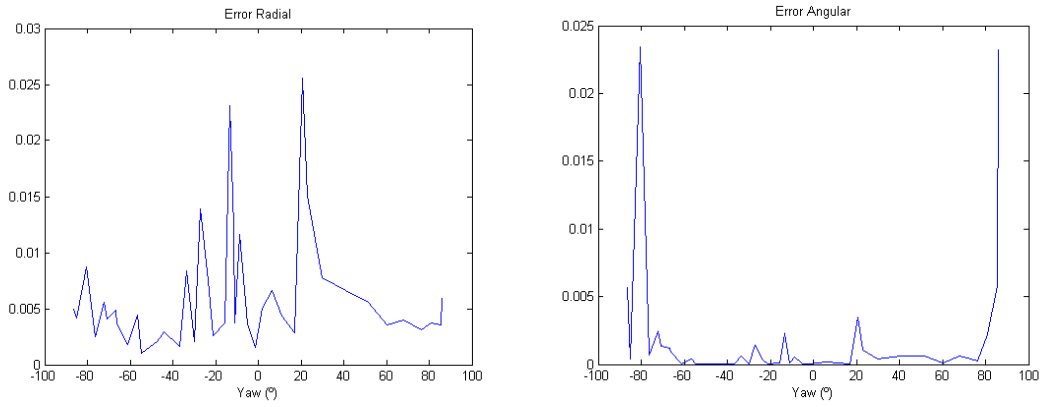

Figura 64 - Errores frente a *yaw*

En las gráficas anteriores se aprecia, como se esperaba, una cierta independencia del error con el ángulo *yaw,* sobre todo del error angular en un rango de ±80º en el que se mantiene por debajo de 0,005º. En el error radial se aprecian una serie de picos pero que no parecen indicar una fuerte dependencia con determinadas orientaciones, siendo el error bastante bajo en cualquier caso (el pico más alto no supera los 2,5 cm).

6.2.4 Prueba de ángulo *pitch*

En el presente experimento se analiza la influencia en las estimaciones del ángulo *pitch* entre la cámara y la baliza. La figura siguiente muestra la baliza en dos valores del ángulo distintos.

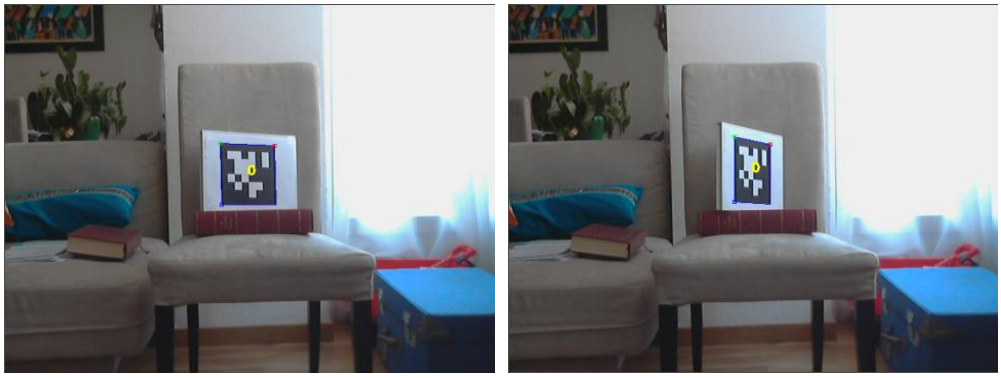

Figura 65 - Baliza a dos ángulos pitch distintos

En este caso el rango de posibles valores es de $\pm 80^{o}$ ya que con valores absolutos mayores se deja de detectar la baliza. Al igual que en los casos anteriores, se han realizado 15 mediciones para 20 valores de ángulo distintos.

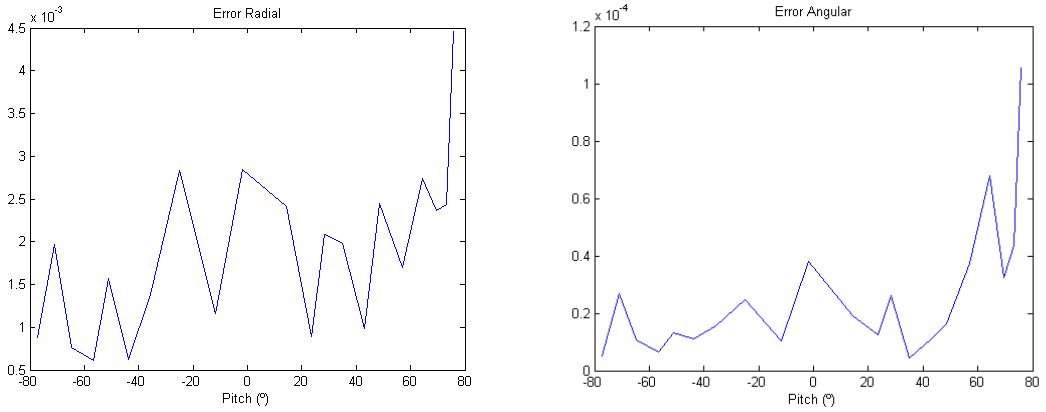

Figura 66 - Errores frente a *pitch*

En el experimento del caso virtual en que se analizaba el ángulo *pitch* se observaba cómo el error disminuía cuando el valor absoluto del ángulo aumentaba. En este caso, en las gráficas de la Figura 66 se observa una tendencia parecida, sobre todo en el rango negativo, aunque en la parte positiva la tendencia parece invertirse a partir de 40^{o}, lo que podría tener su explicación en el método de estimación de pose verdadera utilizado, que como se ha comentado, no es exacto. Tanto el error radial como el angular se mantienen en cotas considerablemente pequeñas (lo que también se puede achacar al método de estimación de pose verdadera utilizado), no superando los 4,5 mm el primero y los $0{,}00012^{o}$ el segundo.

6.2.5 Prueba de ángulo *roll*

El último experimento pone el foco sobre el ángulo *roll* existente entre la cámara y la baliza. En la figura siguiente se pueden ver dos momentos del mismo, con la baliza girada alrededor de su eje X en dos valores distintos.

Figura 67 - Baliza a dos ángulos roll distintos

A lo largo del experimento la baliza comienza prácticamente paralela al plano imagen de la cámara (-15º) y termina prácticamente perpendicular (-80º ya que a partir de este valor se deja de detectar). No se han registrado valores para el rango positivo porque colocar la baliza con inclinación positiva sin que se caiga resulta complicado. El número total de valores para los que se han registrado medidas ha sido 10, tomándose como siempre 15 mediciones en cada caso.

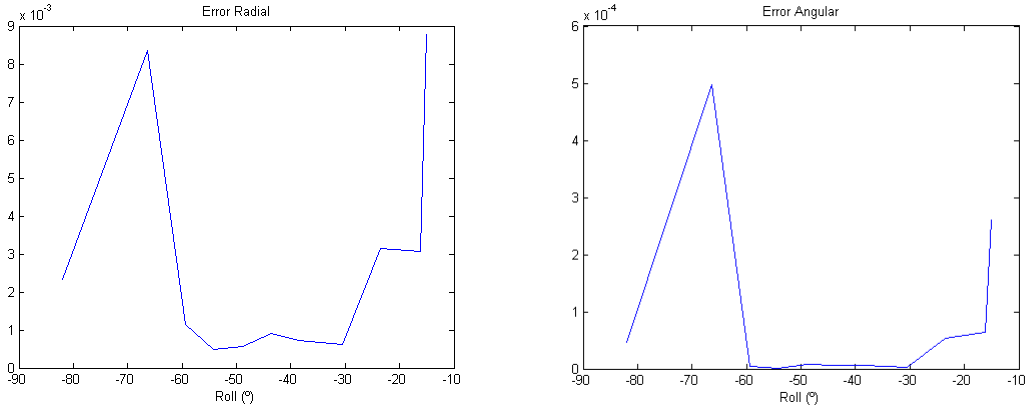

Figura 68 - Errores frente a roll

A partir de lo observado en el experimento en el entorno virtual se podría esperar que el error disminuyera al aumentar el valor absoluto de la inclinación, lo que en las gráficas de la Figura 68 se cumple en el rango (-60, -10). Sin embargo, curiosamente aparece un pico cerca de -70º tanto en el error radial como en el angular que podría estar debido a una detección errónea puntual de la baliza. En cualquier caso el error es significativamente pequeño en todo el rango de estudio, estando el radial por debajo de 9 mm en todo momento y de 0,0006º el angular.

6.3 Conclusiones de los estudios de precisión

Las conclusiones principales que se pueden extraer de los estudios de precisión realizados se enumeran a continuación:

- El algoritmo desarrollado obtiene unos resultados de precisión aceptables en unas condiciones de distancia y de configuración de escena determinadas, por lo que la validación del mismo es satisfactoria.
- Existe una clara dependencia del error (tanto radial como angular) con la distancia a la que se encuentran las balizas. Cuanto más lejos se encuentre un marcador de la cámara mayor es el error que se comete en la estimación de su posición y orientación.
- La dependencia comentada en el punto anterior se ve atenuada con la inclusión de más balizas en la escena, consiguiendo aumentar el rango de distancias en las que el algoritmo se comporta de manera razonable.
- Utilizar balizas en planos perpendiculares también mejora la precisión del algoritmo, sobre todo disminuyendo el error en Z cometido.
- La influencia en la calidad de la estimación del ángulo *yaw* entre la baliza y la cámara es pequeña.
- Utilizar balizas con una cierta inclinación en *roll* o en *pitch* favorece la precisión de las estimaciones obtenidas.
- Se han observado en el entorno real tendencias similares a las apreciadas en el entorno virtual, si bien no se han correspondido completamente debido seguramente a la inexactitud del método utilizado para la estimación de pose verdadera.

Capítulo 7

Conclusiones

Los capítulos presentados hasta el momento han expuesto el contexto de trabajo, los objetivos que se han marcado en el mismo y el desarrollo que se ha llevado a cabo para cumplirlos, junto con la validación correspondiente mediante la realización de varios experimentos. En este último capítulo se recapitulan las principales conclusiones que se extraen de la realización de este Trabajo Fin de Máster y las posibles líneas de trabajo que pueden derivarse de él.

7.1 Conclusiones

El objetivo principal de este trabajo era programar y caracterizar un algoritmo de autolocalización visual basado en marcadores, que se ha cumplido con la implementación del componente *cam_autoloc*. Esta aplicación toma un flujo de imágenes como entrada para proporcionar a su salida una estimación de la posición y orientación de la cámara que las capta. Además se ha validado experimentalmente en dos entornos distintos: en primer lugar en un entorno virtual utilizando las cámaras de un modelo robótico de simulación, y en segundo lugar en un entorno real haciendo uso de una cámara de videoconferencia.

A continuación se repasan los subobjetivos planteados en el capítulo 2 y en qué medida se ha conseguido cada uno de ellos.

- **Desarrollo de un algoritmo de autolocalización visual basado en marcadores**: la respuesta a este subobjetivo es el componente *cam_autoloc*, en el que se ha diseñado y programado en C++ un algoritmo de autolocalización basado en balizas visuales que hace uso solamente de las imágenes proporcionadas por una cámara, sin utilizar sensores adicionales. El proceso se

ha dividido en cuatro grandes pasos, siendo el primero la detección de los marcadores presentes en la imagen, para lo que se ha utilizado balizas *AprilTags* de la familia 36h11. El segundo consiste en la estimación de la posición 3D a partir de cada detección, para a continuación realizar la fusión de todas las estimaciones. Por último, la aplicación permite de dos maneras distintas tener en cuenta las estimaciones pasadas para evitar cambios bruscos: un filtro de Kalman y un filtro de medias ponderadas. Para todo ello se ha hecho uso de diferentes librerías, entre las que se encuentran AprilTags-C++ para la detección de las balizas, OpenCV principalmente para el cálculo 3D y Eigen para los cálculos de álgebra lineal.

- **Caracterización de la solución adoptada**: para cumplir con este subobjetivo se ha realizado un estudio de precisión en el que se ha hecho una comparativa de las estimaciones realizadas frente a las ubicaciones verdaderas en diferentes escenarios y en función de distintas variables. Por un lado se ha trabajado en el entorno virtual de Gazebo y con el cuadricóptero virtual Ardrone, realizando experimentos en los que se desplazaba el drone o se modificaba la escena en función de las variables que se pretendían estudiar en cada caso. Concretamente se ha estudiado la dependencia de la calidad de la estimación con la distancia, con los ángulos *roll*, *pitch* y *yaw* y con la posición del marcador dentro de la imagen. Por otro lado se ha trabajado en un entorno real llevando a cabo experimentos equivalentes a los recién comentados utilizando una cámara de videoconferencia. Para el análisis de los datos recogidos en los experimentos se han desarrollado un conjunto de funciones en Matlab que construyen las gráficas donde se presentan de manera visual los resultados. Además de la validación cuantitativa recién comentada, se ha realizado una validación cualitativa a través de una demostración de realidad aumentada, donde se ha comprobado que el objeto introducido en la escena se observa coherentemente a lo largo de todas las posiciones y orientaciones tomadas por la cámara.

Los requisitos que debe satisfacer la solución implementada, establecidos en la sección 2.2, se analizan a continuación.

- La aplicación se ha desarrollado en el lenguaje de programación C++ sobre la plataforma JdeRobot, siguiendo su filosofía de componentes modulares que se intercomunican a través de interfaces ICE.
- La solución implementada se ejecuta sobre un sistema operativo GNU/Linux Ubuntu 12.04, haciendo uso de diversas librerías operativas en este entorno, como las ya comentadas AprilTags-C++, OpenCV y Eigen, además de

OpenGL para la representación 3D y Qt para la creación del interfaz gráfico de usuario.

- El algoritmo desarrollado consigue su objetivo utilizando como información sensorial únicamente la proporcionada por una cámara, sin hacer uso de otros dispositivos como puedan ser sensores inerciales o de posicionamiento. La única información a priori que necesita es el mapa de balizas (con la posición y orientación de cada una de las que se encuentren en la escena) y los parámetros intrínsecos de calibración de la cámara con la que se trabaje.
- La ejecución de la aplicación resulta fluida, no apreciándose retardos ni parpadeos en la imagen a pesar de estar detectándose marcadores y realizando los cálculos asociados en todo momento. Además, el sistema soporta el movimiento suave de la cámara sin perder la detección de las balizas y consecuentemente la localización estimada.

Entre los conocimientos aportados por el presente Trabajo Fin de Máster se puede destacar los adquiridos con respecto a técnicas de autolocalización (principalmente basadas en marcadores), ámbito considerablemente importante dentro de la visión artificial y de la robótica. Además, para el completo entendimiento de la técnica implementada se ha tenido que profundizar en conceptos de álgebra lineal y geometría 3D.

Este Trabajo Fin de Máster tiene asociado una bitácora digital[1] donde se ha ido recogiendo los progresos según avanzaba el proyecto y en el que se pueden visualizar vídeos con los resultados alcanzados. También se puede acceder al código fuente a través de un repositorio SVN[2].

7.2 Trabajos futuros

A continuación se enumera una serie de líneas de trabajo que pueden partir del presente trabajo tanto para mejorar los resultados obtenidos como para alcanzar nuevos objetivos.

- **Utilización de realidad aumentada**: una primera línea de trabajo puede ser utilizar el algoritmo implementado como base para la investigación en técnicas relacionadas con la realidad aumentada.
- **Optimización del algoritmo de autolocalización visual**: investigar en técnicas de fusión temporal diferentes a las implementadas en este trabajo, que filtren y suavicen aún más los posibles errores de estimación, puede ser una posible mejora. Incluso una línea de trabajo puede ser la implementación de otros algoritmos de autolocalización que no necesiten de balizas, pero que

[1] http://jderobot.org/Alopezceron-tfm

[2] https://svn.jderobot.org/users/alopezceron/tfm/

consigan igual o mejor precisión. En la introducción de esta memoria se mencionaban un par de técnicas que cumplen estos requisitos: MonoSLAM y PTAM.

- **Adaptación del código para móviles y tabletas**: el ordenador utilizado en este proyecto es lo suficientemente potente como para no haberse tenido que preocupar de optimizar el código para que el algoritmo funcionase con fluidez. Un trabajo interesante sería la adaptación del mismo para hacerlo funcionar razonablemente bien en plataformas hardware más limitadas, como pueden ser los móviles y las tabletas.
- **Aplicación en robótica**: incluir el algoritmo implementado como base del sistema de autolocalización de un robot real es otra interesante línea de trabajo.
- **Fusión sensorial**: para mejorar la precisión de la localización, otra posible mejora es fusionar las estimaciones provenientes del algoritmo visual con la información proveniente de otro tipo de sensores, como pueden ser los inerciales.
- **Estudio de la robustez frente a oclusiones y cambios de iluminación**: en el presente proyecto siempre se ha trabajado en condiciones óptimas de iluminación y con las balizas visibles completamente. Un estudio interesante podría ser observar cómo se comporta la detección de los marcadores visuales en diferentes condiciones de iluminación y frente a oclusiones parciales de las balizas.

Referencias

Azuara, D. (2015). Realidad aumentada con interacción física desde una cámara móvil usando JdeRobot. *Proyecto Fin de Carrera, ETSIT, Universidad Rey Juan Carlos* .

Bradski, G., & Kaehler, A. (2008). *Learning OpenCV.* O'Reilly Media, Inc.

Cumberbirch, Y. (2015). 3D augmented reality system using JdeRobot. *Proyecto Fin de Carrera, ETSIT, Universidad Rey Juan Carlos* .

Davison, A. J. (2003). Real-time simultaneous localization and mapping with a single camera. *In Proc. International Conference on Computer Vision* , 1403-1410.

Davison, A. J. (2002). Slam with a single camera. *SLAM/CML Workshop at ICRA 2002* .

Gao, X.-S., Hou, X.-R., Tang, J., & Cheng, H.-F. (2003). Complete solution classification for the perspective-three-point problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence* , 930 - 943.

Hernández, A. (2014). Autolocalización visual aplicada a la realidad aumentada. *Trabajo de Fin de Master, Master Oficial en Visión Artificial, Universidad Rey Juan Carlos* .

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering* .

Klein, G., & Murray, D. (2007). Parallel tracking and mapping for small AR workspaces. *In Proceedings of the Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality* , 225-234.

Lepetit, V., Moreno-Noguer, F., & Fua, P. (2008). EPnP: An Accurate O(n) Solution to the PnP Problem.

López, A. (2010). Localización de un robot con visión local. *Proyecto de Fin de Carrera, ESECT, Universidad Rey Juan Carlos* .

López, L. M. (2010). Autolocalización en tiempo real mediante seguimiento visual monocular. *Proyecto Fin de Carrera, ETSIT, Universidad Rey Juan Carlos* .

Martín, A. (2014). Navegación visual en un cuadricóptero para el seguimiento de objetos. *Trabajo Fin de Grado, Grado en Ingeniería de Computadores, Universidad Rey Juan Carlos* .

Olson, E. (2011). AprilTag: A robust and flexible visual fiducial system. *IEEE International Conference on Robotics and Automation (ICRA),* , 3400 - 3407.

Perdices, E. (2010). Autolocalización visual en la RoboCup con algoritmos basados en muestras. *Trabajo de Fin de Master, ETSIT, Universidad Rey Juan Carlos* .

Ramos, R. (2011). Transformaciones lineales en 3D. En *Apuntes de la asignatura Informática Gráfica.* Universidad de Oviedo.

Rodríguez, D. (2010). Algoritmo evolutivo para la autolocalización de un robot móvil con láser y visión. *Proyecto Fin de Carrera, ETSII, Universidad Rey Juan Carlos* .

Szeliski, R. (2010). *Computer Vision: Algorithms and Applications.*

Yagüe, D. (2015). Cuadricóptero AR.Drone en Gazebo y JdeRobot. *Proyecto Fin de Carrera, ETSIT, Universidad Rey Juan Carlos* .

Zhang, Z. (2000). A Flexible New Technique for Camera Calibration. *IEEE Trans. Pattern Anal. Mach. Intell* , 1330-1334.