Ingeniería Informática Superior

Curso académico 2014/2015

Proyecto Fin de Carrera

Autolocalización de un robot industrial en interiores con balizas visuales

Autor:

José Manuel Villarán Núñez

Tutor:

José María Cañas Plaza

Resumen

El auge de la robótica ha traído consigo el planteamiento de nuevos problemas para los desarrolladores, como la localización de robots en un determinado entorno. En este proyecto de fin de carrera se aborda el problema de la autolocalización de un sistema robótico en una nave industrial, utilizando sus cámaras.

Debido al reciente desarrollo de nuevas interfaces en la industria de los videojuegos, los sensores RGB-D, como Kinect, han demostrado un gran potencial. Por su versatilidad y su relativo bajo coste actualmente se utilizan en múltiples aplicaciones de otras áreas como la visión artificial y la robótica. Estos serán los principales sensores de nuestro sistema.

Se ha desarrollado un componente que proporciona estimaciones de localización robustas utilizando balizas visuales (AprilTags) que se distribuyen por el entorno. Estas son detectadas por las cámaras y se calcula, utilizando geometría y sabiendo la configuración del sistema, la posición y orientación estimadas del robot en el sistema de referencia absoluto. Posteriormente, todas las estimaciones instantáneas son fusionadas añadiendo también información odométrica del robot. De esta forma se consiguen estimaciones finales a lo largo del tiempo que son robustas ante oclusiones o falsos positivos, lo cual se ha comprobado tanto en el simulador Gazebo como en real.

Para la programación del sistema se ha utilizado el lenguaje C++ sobre la plataforma JdeRobot en Linux. Puesto que el componente desarrollado forma parte de un proyecto industrial, esto ha facilitado la comunicación con otros componentes de navegación, control y de interacción con el usuario.

Índice general

Capítulo 1

Introducción

1.1. Visión en robótica

Este proyecto de fin de carrera se sitúa en el marco de la visión artificial y la robótica puesto que es parte del desarrollo completo de un sistema robótico real para una empresa industrial determinada. Este sistema debe ser totalmente autónomo y capaz de navegar, sin colisionar con objetos fijos ni móviles, por naves industriales llevando cargas y siguiendo rutas establecidas previamente por operarios desde un PC o desde una aplicación Android sobre una tablet. Además el robot cuenta con mecanismos de parada de emergencia y teleoperación por operadores en caso de cualquier tipo problema.

Para ello se desarrolla un componente de autolocalización basado en balizas visuales, montado sobre el robot anteriormente descrito, utilizando como sensores dos dispositivos RGB-D y encoders como fuentes de datos para conseguir estimaciónes precisas de posición en todo momento.

1.1.1. Robótica

Se denomina robótica a la rama de la tecnología que se dedica al diseño, construcción y aplicación de máquinas robóticas para solucionar problemas de forma automática. Gen-

eralmente, los robots sirven para realizar tareas concretas en las que se necesita rapidez y precisión al mismo tiempo, tareas pesadas o peligrosas para el ser humano, etc. La robótica hace uso de múltiples campos de conocimiento para lograr sus objetivos, como la mecánica, informática, electrónica, visión artificial o física.

El origen de la robótica se establece en el siglo XIII, cuando aparecen los primeros autómatas capaces de realizar actividades de forma mecánica en entornos muy simplificados y controlados. Con el avance de la técnica y la industrialización se fue consiguiendo la creación de robots más potentes. El término 'robot' deriva de la palabra checa 'robota' (significa 'trabajo forzado'), utilizada en 1921 en la obra dramática 'Rossum's Universal Robots' de Karel Capek.

Un robot consta siempre de tres partes diferenciadas:

- Sensores: son la parte encargada de percibir el mundo que rodea al robot. Reciben datos usando diferentes mecanismos y los traducen a señales eléctricas que un ordenador pueda procesar. Generalmente se distinguen entre pasivos (sólo reciben información del mundo, sin modificarlo) y activos (para recibir información emiten alguna señal al entorno). Existen actualmente múltiples sensores: utrasonidos, cámaras, láser, infrarrojo, micrófonos, etc.
- Actuadores: realizan las acciones indicadas por la unidad de procesamiento en el mundo. Suelen ser sistemas basados en motores.
- Unidad de procesamiento: se encarga de interpretar la información de los sensores, calcular una acción a realizar como respuesta y enviar las órdenes adecuadas a los actuadores para que la realizen. Generalmente son ordenadores de mayor o menor potencia.

Dentro de la robótica se distinguen dos grandes grupos de clasificación:

- Robots de manipulación, generalmente brazos mecánicos que realizan tareas muy limitadas de forma precisa y veloz
- Robots móviles, que se desplazan por un entorno determinado interactuando con él de una u otra manera.

Este proyecto de fin de carrera se enmarca dentro del segundo grupo, donde podemos encontrar ejemplos de actualidad como el coche autónomo de Google o los *drones* que navegan sin teleoperación (figura 1.1).

(a) Google car (b) Dron

Figura 1.1: (a)Google car - (b) Dron teleoperado desde un móvil

1.1.2. Visión artificial

La visión artificial, o visión computacional, es un campo de la inteligencia artificial que intenta extraer la máxima información posible procesando de imágenes con diferentes algoritmos. Dicha información puede servir para detectar objetos en la escena, reconstruir virtualmente entornos en 3D, contruir trayectorias, etc. Puesto que los datos visuales que procesamos los seres humanos con nuestro cerebro son en definitiva imágenes de color en 2D, se presupone un enorme potencial a los sensores de tipo cámara (pueden aportar mucha información), aunque la dificultad de los algoritmos para extraerla y la capacidad computacional necesaria hacen que sea una tarea complicada.

Se considera que el comienzo de la visión artificial fue en el año 1961, cuando Larry Roberts realizó experimentos de reconocimiento de bloques en entornos simples, capturando imágenes con una cámara y procesándolas posteriormente con un ordenador. En esa década, muchos científicos se sientieron atraídos por la idea de poder reconocer y analizar el mundo real a través de un ordenador, por lo que aparecieron múltiples grupos de investigación.

Más tarde, en la década de los 90, cuando los ordenadores adquirieron suficiente ca-

pacidad de cómputo, empezaron a surgir aplicaciónes reales específicas para realizar tareas concretas como seguimiento de objetos, reconocimiento de bordes o clasificación de patrones en entornos controlados. Ya tras el cambio de milenio, debido al gran auge de la tecnología móvil, los sensores de cámara redujeron considerablemente su coste, lo que propició un uso más generalizado del mismo y surgieron más aplicaciones en otros ámbitos.

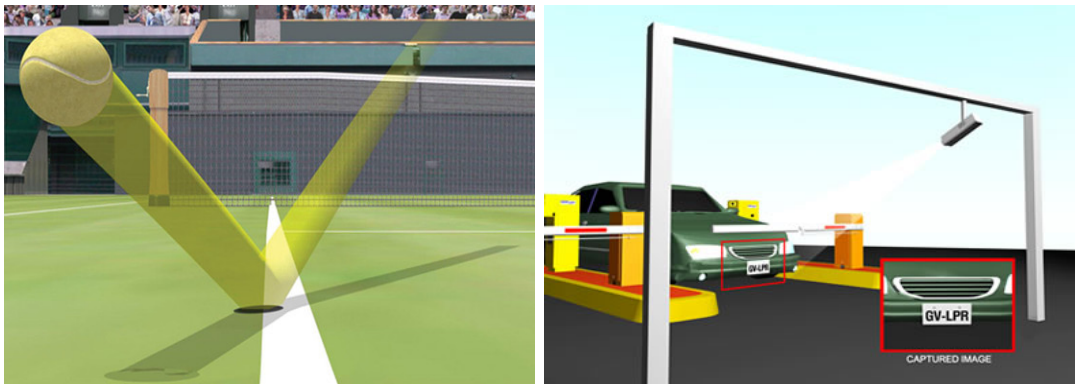
Los problemas que intenta resolver la visión artificial y algunas de las aplicaciones que actualmente podemos encontrar haciendo uso de esta tecnología y algoritmos son:

- La detección, segmentación, localización y reconocimiento de ciertos objetos en imágenes, como caras humanas en sistemas de control de acceso o la detección de defectos en productos de cadenas de producción industriales. Otro ejemplo concreto es el sistema de detección automática de matrículas en el parking del aeropuerto de Barajas, que permite la entrada y salida de vehículos reconociendo números y letras de sus matrículas y comparándolos con una base de datos (figura 1.2).

- El registro de diferentes imágenes de una misma escena u objeto, es decir, hacer concordar un mismo objeto en diversas imágenes para realizar reconstrucciones en 3D de entornos. Por ejemplo en medicina se realizan diagnósticos por imagen que consiguen mayor precisión que los métodos tradicionales. También se utiliza para la construcción de mapas en los que podría navegar un robot.

- El seguimiento de un objeto en una secuencia de imágenes. Existen sistemas de cámaras de tráfico que pueden seguir vehículos y extraer información del estado de la circulación. El sistema del 'ojo de halcón' de tenis también hace uso de esta tecnología para seguir en todo momento la pelota y poder recrear su trayectoria. En el ámbito de los videojuegos, sensores como 'Kinect' proporcionan una nueva forma de interactuar con la consola gracias a la detección y seguimiento de los movimientos de los jugadores.

- El procesamiento de imágenes consigue la reducción de ruido y la mejora del contraste, lo que resulta útil en campos como la fotografía y los videojuegos. Google Street View también utiliza procesado de imágenes para el emborronado automático de caras y matrículas.



(a) Sistema Ojo de Halcón (b) Reconocimiento visual de matrículas

Figura 1.2: (a) Sistema Ojo de Halcón - (b) Reconocimiento visual de matrículas

1.2. Autolocalización de robots

Uno de los problemas dentro de la robótica es conocer la posición donde se encuentra nuestro robot en el entorno, para poder navegar correctamente por él. Por localización se entiende 'averiguar el lugar en que se halla alguien o algo' y, por tanto, autolocalizarse es conocer por sí mismo la posición donde se encuentra un individuo. En la actualidad se han desarrollado diversas técnicas para resolver el problema de localizar dispositivos en un determinado entorno. Dependiendo del dispositivo, el entorno y la precisión necesaria para su función se emplea generalmente uno u otro método.

En la naturaleza podemos encontrar diversos sistemas de autolocalización que utilizan características propias del entorno para obtener información y, consecuentemente, una estimación de posición. Por ejemplo, animales como los cetáceos o los murciélagos se basan en mecanismos de ecolocalización para conocer el entorno y saber su lugar en el mismo. Los murciélagos cuentan con dos oídos, situados a cierta distancia uno del otro, en los que reciben el ultrasonido que emiten, rebotado de los obstáculos del entorno con diferencias de intensidad, tiempo y frecuencia dependiendo de la posición espacial del obstáculo. Con esta información calculan la distancia, tamaño y características de los objetos del entorno, por lo que también pueden autolocalizarse dentro de él. Durante los viajes migratorios de aves, estas consiguen orientarse y calcular así la dirección en la que deben desplazarse en cada momento a través de información geomagnética y celeste para estimar su latitud. Gracias al efecto "Zeno Cuántico", el campo magnético terrestre genera una respuesta química en las retinas

de estos animales que activa la parte visual de su cerebro, facilitándoles la autolocalización.

Para muchos seres vivos, incluido el ser humano, el principal sentido para autolocalizarse es la vista. Nuestro cerebro está preparado para recibir simultáneamente una imagen de cada ojo y procesarlas adecuadamente. Puesto que nuestros ojos están separados una determinada distancia el uno de el otro, podemos inferir distancias a objetos, reconocerlos y compararlos con conocimientos previamente adquiridos para conocer nuestra posición en un escenario.

1.2.1. Localización GPS

El sistema más utilizado para la autolocalización en exteriores es el denominado "Sistema de Posicionamiento Global"(GPS). Este sistema permite a un dispositivo situado en cualquier parte del mundo conocer su posición en el mismo con una precisión de unos pocos metros. En 1964 fue desarrollado e instalado inicialmente por el departamento de defensa de los Estados Unidos para proveer a sus flotas de observaciones de posición actualizadas y precisas (sistema TRANSIT). El sistema fue mejorando y utilizándose también en aplicaciones comerciales.

El GPS funciona mediante una red de 24 satélites en órbita sobre el planeta tierra, a 20.200 km de altura, con trayectorias sincronizadas para cubrir toda la superficie de la Tierra. Cuando se desea determinar la posición, el receptor que se utiliza para ello localiza el máximo número posible de satélites de la red, de los que recibe unas señales indicando la identificación y la hora del reloj de cada uno de ellos. Con base a tiempo que tardan en llegar las señales al equipo, se calcula la distancia al satélite y determina fácilmente su posición relativa mediante triangulación. Conociendo además las coordenadas de cada uno de ellos por la señal que emiten, se obtiene la posición absoluta del dispositivo.

El sistema GPS tiene múltiples aplicaciones, tanto civiles como militares, en diferentes áreas:

- Civiles
  - Navegación terrestre, tanto en vehículos como de forma peatonal.
  - Teléfonos móviles
  - Topografía y geodesia, para análisis y desarrollo de mapas.

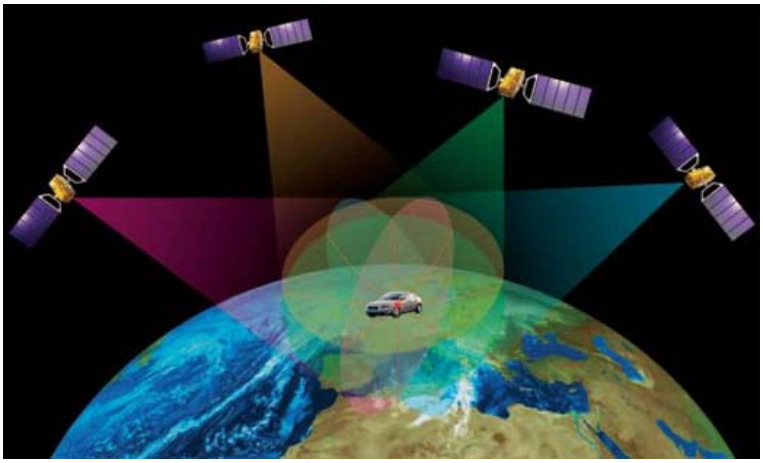


Figura 1.3: Localización GPS

- Aeronavegación, para la localización de aviones.
- Construcción de infraestructuras, para la nivelación de terrenos, cortes de talud, tendido de tuberías, etc.
- Localización en el campo para agricultura y ganadería, seguimiento de animales en peligro de extinción, trabajos de campo, etc.
- Navegación marítima de embarcaciones.
- Deportes aéreos como el parapente, ala delta, planeadores, etc.

▪ Militares

- Navegación terrestre, aérea y marítima.
- Guiado de misiles y proyectiles de diverso tipo.
- Búsqueda y rescate.
- Reconocimiento y cartografía.

Actualmente la Unión Europea está desarrollando su propio sistema de posicionamiento por satélite ("Galileo") y la República Popular China su propio sistema de navegación ("Beidou") para el año 2020.

1.2.2. Localización DGPS

El GPS diferencial, o DGPS, es un sistema que proporciona a los receptores de GPS correcciones de los datos recibidos de los satélites GPS, con el fin de proporcionar una mayor precisión en la posición calculada. Consiste en la utilización de un receptor móvil y una estación (o estaciones) de referencia situadas en coordenadas conocidas con gran exactitud. La estación de referencia comprueba todas las medidas a los satélites y obtiene en tiempo real las coordenadas de ese punto, cuya posición ya se conocían con exactitud previamente por otra fuente (p.e. topografía). Comparando resultados, calcula los errores del sistema en tiempo real y transmite dichas correcciones a los receptores móviles dentro su área para que recalculen su posición, compensando el error y consiguiendo así estimaciones mucho más precisas.

Las correcciones DGPS pueden recibirse por radio (FM), por internet o por algún sistema de satélites diseñado para tal efecto (en Estados Unidos existe el WAAS, en Europa el EGNOS y en Japón el MSAS, todos compatibles entre sí).

Para que las correcciones DGPS sean suficientemente válidas, el dispositivo receptor tiene que estar relativamente cerca de alguna estación DGPS (por lo general a menos de 1000 km). Las precisiones que se consiguen con este tipo de sistemas son de unos pocos centímetros, por lo que tienen utilidades en el ámbito de la ingeniería, transporte, industria, etc.

1.2.3. Localización GSM

La localización GSM es un método propio de la telefonía móvil que permite determinar, con una cierta precisión, dónde se encuentra físicamente un terminal móvil determinado. Para ello, se utilizan diferentes técnicas utilizando Identificadores de célula, diferencia de tiempos y ángulos de llegada de la señal, etc. consiguiendo delimitar la zona donde se encuentra el terminal con una precisión bastante baja comparada con otros sistemas (200 m en áreas urbanas, 2 km en áreas suburbanas y entre 3 - 4 km en entornos rurales). Debido a esto, sus aplicaciones son muy limitadas, como por ejemplo conocer la zona aproximada donde se encuentra un teléfono movil perdido.

1.3. Autolocalización visual

Unas de las técnicas más precisas de autolocalización son las basadas en visión. Generalmente intentan utilizar los datos que proporciona uno o varios sensores de tipo cámara para captar información del entorno, procesarla y dar una estimación tanto de la posición como de la orientación del dispositivo. Un ejemplo actual de uso de autolocalización visual es el de las aspiradoras de Dyson, que constan de una cámara con la que procesa su entorno en 360 grados de forma dinámica, busca puntos de interés en el entorno (tal y como se observa en la figura 1.4) y consigue así localizarse en la estancia y avanzar de forma más inteligente.

Figura 1.4: Aspiradora Dyson utilizando autolocalización visual

Los sistemas de posicionamiento en interiores (IPS, del inglés 'Indoor Positioning System) se encargan de localizar dispositivos o personas dentro de un entorno cerrado, como edificios, almacenes industriales, centros comerciales, museos, etc. Debido a que otros sistemas como el GPS no funcionan adecuadamente en entornos cerrados y sus márgenes de error son demasiado grandes para entornos pequeños, se hacen necesarios otros tipos de tecnologías como la óptica, la radio, señales acústicas o visuales, etc. Por lo general se utilizan varias medidas independientes para compensar errores y ambigüedades, como en el caso de localización visual a través de la triangulación de balizas en el entorno.

Según la Sociedad Española para la Investigación y Desarrollo en Robótica (SEIDROB), el problema de la 'Localización 3D en interiores y tiempo real' es una cuestión de actualidad en el mundo de la robótica que todavía se encuentra en desarrollo para llegar a una solución suficientemente robusta. De hecho, esta sociedad entregaba un premio durante el curso 2014-

2015 para desarrolladores que consiguieran resolver este problema.

1.3.1. Técnicas de autolocalización visual

El problema de localizarse utilizando visión artificial ha sido denominado históricamete de forma distinta por diferentes comunidades: *Structure for Motion* en visión artificial (procesando imágenes en lotes) y *Visual SLAM* en robótica (tiempo real). A continuación se explican algunas de las técnicas más importantes para abordar este problema:

- **Odometría visual**: Esta técnica intenta estimar una matriz de rotación y translación óptima entre dos conjuntos de puntos, extraídos de dos imágenes, para estimar el movimiento relativo 3D de la cámara entre ambos fotogramas. Para ello se utilizan métodos de extracción de puntos de interés y cálculo de descriptores y cálculo de emparejamientos para asociar las características encontradas entre lecturas consecutivas de las imágenes RGB. También se suele realizar una estimación del movimiento relativo entre fotogramas consecutivos mediante los algoritmos SVD y RANSAC como en el proyecto de fin de carrera de Daniel Martín Organista [Martín Organista, 2014]. Aunque normalmente se intenta reducir el ruido y las observaciones erróneas para evitar trayectorias inconsistentes, esta técnica es incremental y arrastra el error de una iteración a la siguiente.

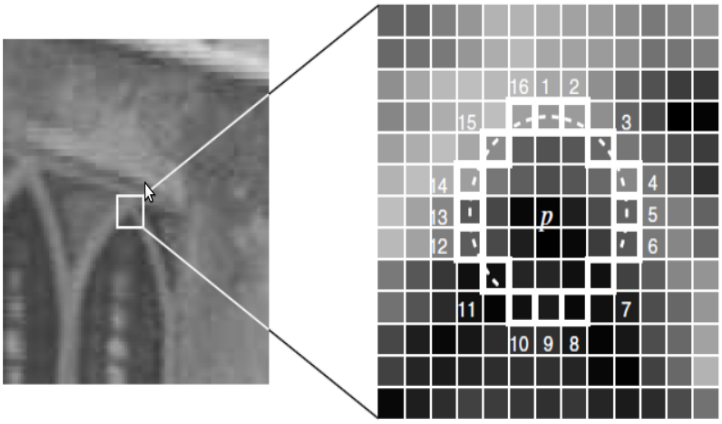


Figura 1.5: Detección de esquinas

- **Localización probabilística**: Este tipo de localización consiste en determinar la probabilidad de que el robot se encuentre en una determinada posición del entorno a través de sus lecturas sensoriales y movimientos a lo largo del tiempo. A cada posible posición se le asocia una probabilidad de que sea la posición actual real del robot. Esta probabilidad se va realimentando con la incorporación de nuevas lecturas sensoriales (imágenes) y movimientos del robot, por lo que se van eliminando poco a poco las ambigüedades hasta que las estimaciones convergen alrededor del punto donde se encuentra el robot. Por tanto, aunque no conozcamos la posición inicial, finalmente se da una estimación de posición conociendo el mapa del entorno. El mayor problema de esta técnica es la localización en espacios altamente simétricos, donde la verosimilitud de muchos de los puntos es prácticamente la misma. Ejemplos de esta técnica de localización podemos encontrar en el proyecto de fin de carrera de Alberto López [López Fernández, 2005] o en los trabajos de Frank Dellaert [Dellaert, 1999], que desarrolló un robot llamado *Minerva* capaz de leer marcas en el techo a través de una cámara para autolocalizarse y actuar de guía en un museo.

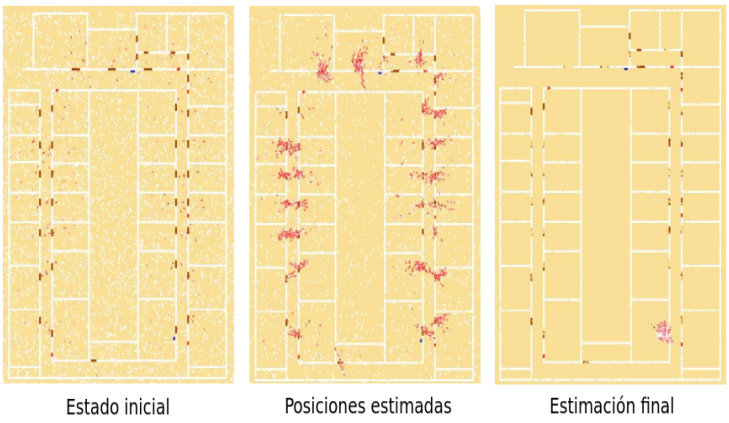


Figura 1.6: Localización probabilística

- **MonoSLAM**: son las siglas de siglas de *Monocular Simultaneous Location And Mapping*. Esta técnica intenta reconstruir un mapa y localizarse en el partiendo de imágenes de una sola cámara. Para ello se extraen características de la imagen por medio de reconocimiento de bordes y esquinas y, posteriormente, se utilizan filtros de seguimiento y filtros de ruido para conseguir una continuidad temporal de los puntos de la imagen. Uno de los filtros más usados es el filtro de Kalman, en el que las observaciones son los píxeles donde proyectan los puntos de interés 3D y el estado es la posición y orientación

de la cámara (6 valores) junto con la posición de los puntos del mapa que se van construyendo. Ejemplos representativos del uso de esta técnica se pueden encontrar en las publicaciones de Andrew Davison [Davison, 2003] y en su página web[1].

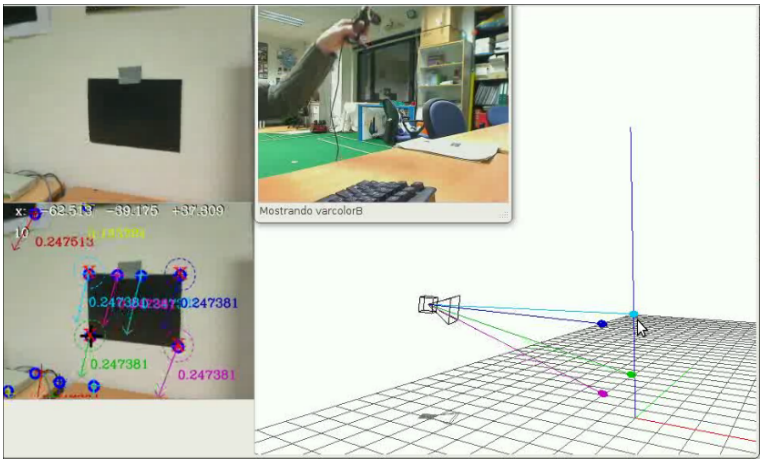


Figura 1.7: MonoSLAM

- **PTAM**: sus siglas vienen de *Parallel Tracking and Mapping*, es un algoritmo que tiene como objetivo resolver el mismo problema que MonoSLAM pero desde otro punto de vista. Esta técnica se centra en que la parte de *tracking* (posición de la cámara) pueda funcionar en tiempo real, por lo que evita realizar la parte de *mapping* en cada iteración. De esta forma funcionan tracking y mapping en hilos separados, aumentando la velocidad de procesamiento del algoritmo. En vez de usar los métodos de filtrado de MonoSLAM, PTAM emplea una optimización por ajuste de haces (*bundle adjustment*) y detecciones de puntos de interés y emparejamiento. Además, este algoritmo hace uso de *keyframes* (fotogramas claves) que se utilizan tanto para localizarse como para crear el mapa de puntos. El investigador Georg Klein [Klein, 2009] tiene en su página web[2] numerosos ejemplos del uso de esta forma de autolocalización.

- Existen otros sistemas que se basan en el uso de balizas visuales situadas en el entorno para conseguir estimaciones precisas de autolocalización.

[1] http://www.doc.ic.ac.uk/ ajd/
[2] http://www.robots.ox.ac.uk/ gk/

1.3.2. Autolocalización visual en la URJC

Dentro del departamento de robótica de la Universidad Rey Juan Carlos existen precedentes de estudios sobre la localización en interiores, utilizando generalmente visión artificial. Estos trabajos han servido como fuente de información para la elaboración de este proyecto de fin de carrera y son los antecedentes directos del mismo:

- Proyecto de fin de carrera de José Alberto López Fernández en 2005, con el título *Localización de un robot con visión local* [López Fernández, 2005].
- Proyecto de fin de carrera de Luis Miguel López Ramos en 2010, con el título *Autolocalización en tiempo real mediante seguimiento visual monocular* [López Ramos, 2010].
- Proyecto de fin de carrera de Daniel Martín Organista en 2014, que trata de la *Odometría visual con sensores RGBD* [Martín Organista, 2014].
- Trabajo de fin de máster de Alejandro Hernández Cordero en 2014 sobre *Autolocalización visual aplicada a la Realidad Aumentada* [Hernández, 2014].
- Proyecto fin de carrera de Luis Roberto Morales en el año 2014 sobre un *Algoritmo de autolocalización evolutivo multimodal para robots autónomos* [Morales, 2014].

Capítulo 2

Objetivos

Tras haber expuesto las motivaciones y el contexto en el que se engloba este proyecto de fin de carrera, en este capítulo daremos una explicación más detallada del problema que se intenta resolver y los pasos dados para conseguirlo.

2.1. Descripción del problema

Como se ha esbozado anteriormente, este proyecto se enmarca dentro del departamento de robótica de la URJC para el desarrollo completo de un sistema robótico real, para una empresa del área de la robótica, que sea totalmente autónomo y capaz de navegar (sin colisionar con objetos fijos ni móviles) por naves industriales llevando cargas y siguiendo rutas establecidas previamente por operarios sobre un mapa del entorno.

Aunque parte del software se encontraba desarrollado, se demuestra empíricamente que el algoritmo encargado de la autolocalización del robot no es lo suficientemente robusto para el tipo de problema que se desea resolver. Por tanto, nuestro objetivo principal es el desarrollo completo de un componente de autolocalización que sea capaz de dar estimaciones precisas de posición y orientación para nuestro robot.

Este objetivo global se ha divido en otros tres subobjetivos:

1. Familiarización y apredizaje de diversas herramientas, librerías y entornos de programación para el desarrollo de aplicaciones robóticas, como JdeRobot u OpenCV.

2. Diseño y desarrollo de un componente para la autolocalización visual de nuestro robot. Se deben elaborar algoritmos de prueba para comprobar la robusted y precisión de diferentes librerías de detección y paso a 3D instantáneo de observaciones de balizas.

3. Pruebas exhaustivas tanto sobre el entorno simulado como sobre el real para comprobar el correcto funcionamiento de nuestro componente. También se comprobará en estas pruebas la robustez y precisión de la estimación final conseguida.

2.2. Requisitos

Para cumplir los objetivos marcados de forma satisfactoria, debemos además satisfacer los siguientes requisitos:

- El desarrollo se realizará utilizando la herramienta JdeRobot, creada, mantenida y utilizada activamente por el Grupo de Robótica de la URJC, lo que facilitará la creación de software modular en forma de componentes. La programación se realizará en el lenguaje C++.

- El software desarrollado debe ser portable entre diferentes arquitecturas y sistemas, puesto que será necesario en el momento de las pruebas comprobar distintas configuraciones para optimizar el rendimiento del sistema. También debe poder utilizarse tanto en simulación como en el robot real.

- La robusted y precisión del algoritmo desarrollado serán unos de los requisitos más importantes puesto que este tendrá que funcionar sobre un robot real y poder aportar siempre estimaciones lo más precisas posibles pese a cambios en el entorno, oclusiones de balizas, visualizaciones escoradas de las mismas, etc.

- El componente de autolocalización visual debe funcionar en tiempo real, por lo que tendrá que estar lo suficientemente optimizado para poder ejecutarse en un pc junto con otros componentes del sistema robótico.

- Puesto que el robot contará con dos dispositivos RGB-D, nuestro algoritmo deberá poder integrar en sus cálculos imágenes obtenidas de cualquiera de las dos cámaras, situadas en distintas posiciones sobre el robot.

- El componente desarrollado debe seguir los convenios e interfaces utilizados en el resto de componentes del sistema del que formará parte.

2.3. Metodología

Durante este proyecto de fin de carrera se ha utilizado el modelo de ciclo de vida en espiral, que facilita el desarrollo incremental del software. De esta forma, se consigue en cada una de las iteraciones un componente funcional que va aumentando su complejidad y sus funciones conforme avance el tiempo de desarrollo.

Cada una de las fases del proyecto se corresponde con un ciclo del modelo en espiral, que a su vez se dividira en 4 partes diferenciadas:

- Determinar objetivos: En este momento se establecen las necesidades que debe satisfacer el componente en la iteración actual, teniendo en cuenta los objetivos finales, por los que según avancen los ciclos aumentará el en coste y complejidad.

- Evaluar alternativas: Se determina las diferentes formas de alcanzar los objetivos que se han establecido en la fase anterior, empleando diferentes enfoques como el rendimiento, gestión del sistema, organización, etc. También se evaluan y reducen los riesgos tomados en esta iteración.

- Diseñar, desarrollar y verificar: En esta parte se diseña el producto siguiendo la alternativa más idónea de la sección anterior y se desarrollará siguiendo las especificaciones fijadas. Después se pasará a la parte de pruebas para comprobar el correcto funcionamiento del componente.

- Planificar las fases siguientes: Teniendo en cuenta los resultados de la fase anterior, se planifica la siguiente iteración revisando posibles errores cometidos a lo largo del ciclo y, finalmente, se inicia un nuevo ciclo de la espiral.

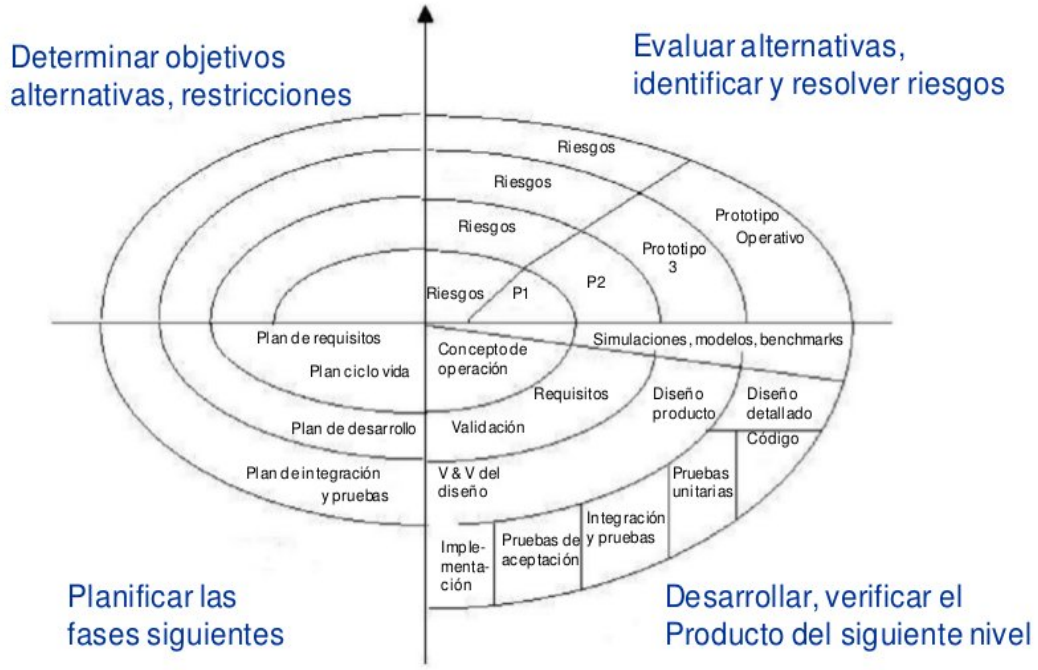


Figura 2.1: Modelo en espiral

Durante el transcurso de proyecto, se han realizado reuniones semanales con el tutor del Proyecto Fin de Carrera para acordar, establecer y revisar los objetivos que se pretenden alcanzar en cada ciclo y las mejores alternativas en cada caso.

Desde el comienzo de este proyecto en 2013 se utiliza una herramienta web para realizar un seguimiento temporal del plan de trabajo, la mediawiki. Ahí se reflejan los hitos principales alcanzados, a modo de bitácora, con imágenes y vídeos de los resultados obtenidos.

2.4. Plan de trabajo

A lo largo de este proyecto se han ido desarrollando una serie de etapas diferenciadas que han permitido llegar a nuestros objetivos finales:

- Familiarización con el entorno software: JdeRobot es la plataforma de desarrollo prin-

cipal utilizada en la mayoría de proyectos realizados en el departamento de robótica de la URJC. El objetivo principal de esta fase es aprender a utilizar este software, sus componentes y sus drivers para más adelante utilizarlos como parte de nuestro proyecto. Como parte del aprendizaje, se desarrolla un driver para el robot *Kobuki* adaptado a la arquitectura JdeRobot y también se adapta un modelo del mismo en simulación.

- Desarrollo de prototipo de autolocalización: Utilizando la librería Apriltags, se desarrolla un componente básico que devuelve estimaciones de posición utilizando una sola cámara. Se crea, por tanto, una primera base software para contruir nuestro componente.

- Desarrollo de una interfaz gráfica: A modo de depuración y visualización de resultados, se crea un mundo 3D utilizando la librería OpenGL. Será de gran ayuda durante el proyecto para la corrección de errores y refinamiento del algoritmo.

- Mejoras en el algoritmo: Se añade una segunda cámara al algoritmo, por lo que se deberán integrar estimaciones individuales de cada una de las cámaras en una final. Además se reconcilian las estimaciones de las cámaras con la estimación dada por los encoders, lo que nos permitirá tener una única estimación más robusta en todo momento.

- Integración con el resto de componentes: Sobre el robot industrial real, se adaptan interfaces y convenios con el resto de software desarrollado para él, para así conseguir un sistema funcional. Se realizan numerosas pruebas para el refinamiento del algoritmo y del sistema completo. También se adapta el componente para su funcionamiento en el simulador.

Capítulo 3

Infraestructura

En este capítulo describiremos por un lado las librerías y entornos de programación que se han utilizado para la construcción de nuestro componente de autolocalización visual. Por otro lado, se muestra la estructura hardware y software del sistema completo del proyecto industrial en que se enmarca, dentro del cual se ha realizado este proyecto de fin de carrera.

3.1. Librerías y entornos

Para la realización de este proyecto de fin de carrera se han utilizado, principalmente, las siguientes librerías software y entornos:

- **ICE**: son las siglas de 'Internet Communications Engine' y se trata de un middleware desarrollado por la empresa ZeroC[1] para realizar la comunicación entre aplicaciones distribuidas. Sigue el paradigma de la orientación a objetos para realizar llamadas remotas a procedimientos de forma sencilla, como si se trataran de llamadas locales (fue muy influenciado por CORBA), y sin la necesidad de usar el protocolo HTTP. Ofrece también otras funciones como balanceo de carga, capacidad de atravesar cortafuegos y servicios de publicación y subscripción. Para utilizar estos servicios, las aplicaciones se asocian a unos esquemas definidos en el lenguaje 'slice', que sirven para determinar

[1] https://zeroc.com

las interfaces a utilizar.

Soporta varios lenguajes como C++, Java, Visual Basic, C#, Objective-C, Python, PHP y Ruby y funciona sobre la mayoría de sistemas operativos (Windows, Linux, Mac OS X y Solaris).

En el presente proyecto se ha empleado la version 3.4.2 para la comunicación de nuestro componente con otros, como por ejemplo la recepción de imágenes desde las cámaras.

- **JdeRobot**: es una herramienta de software libre para el desarrollo de aplicaciones de robótica, domótica y visión artificial que nació en el año 2003 a partir de la tesis doctoral de José María Cañas [Plaza, 2003] y fue desde entonces evolucionada y mantenida por el Grupo de Robótica de la Universidad Rey Juan Carlos[2].

  JdeRobot se basa en la creación y uso de componentes que se comunican entre sí a través de interfaces estandarizadas utilizando la tecnología ICE anteriormente descrita. Estos componentes están, por tanto, en un nivel superior de abstracción puesto que para el programador es indiferente la forma en la que se realice la transmisión de datos a otros componentes o como estén estos construidos internamente, siempre y cuando se mantengan las mismas interfaces de comunicación TCP/IP . Esto simplifica y permite el desarrollo de sistemas distribuidos, proporcionando gran flexibilidad a la hora de elegir donde ejecutar cada componente; por ejemplo, se podrán tener componentes interactuando en diferentes lenguajes, entre computadores con diferentes sistemas operativos, en dispositivos móviles, tabletas, etc. Además, la sustitución de alguno de estos componentes se puede realizar de forma sencilla, sin impacto negativo en la arquitectura del sistema.

  JdeRobot esta fundamentalmente programado en C++, aunque también existen modulos para otros lenguajes como Python y Java. Cada aplicación desarrollada hace uso de uno o más componentes ejecutados en procesos independientes, por lo que se resuelven problemas relacionados con la sincronización de procesos. Ofrece además un amplio repertorio de *drivers* con dispositivos hardware que facilita el envío y recepción de datos de sensores y actuadores, utilizando las mismas interfaces ICE que las usadas entre componentes.

  Se ha empleado la version 5.2.4 para encapsular el algoritmo de autolocalización visual dentro de un componente, de forma que pueda funcionar de forma distribuida y comunicarse con otros componente de manera más sencilla.

[2] http://jderobot.org

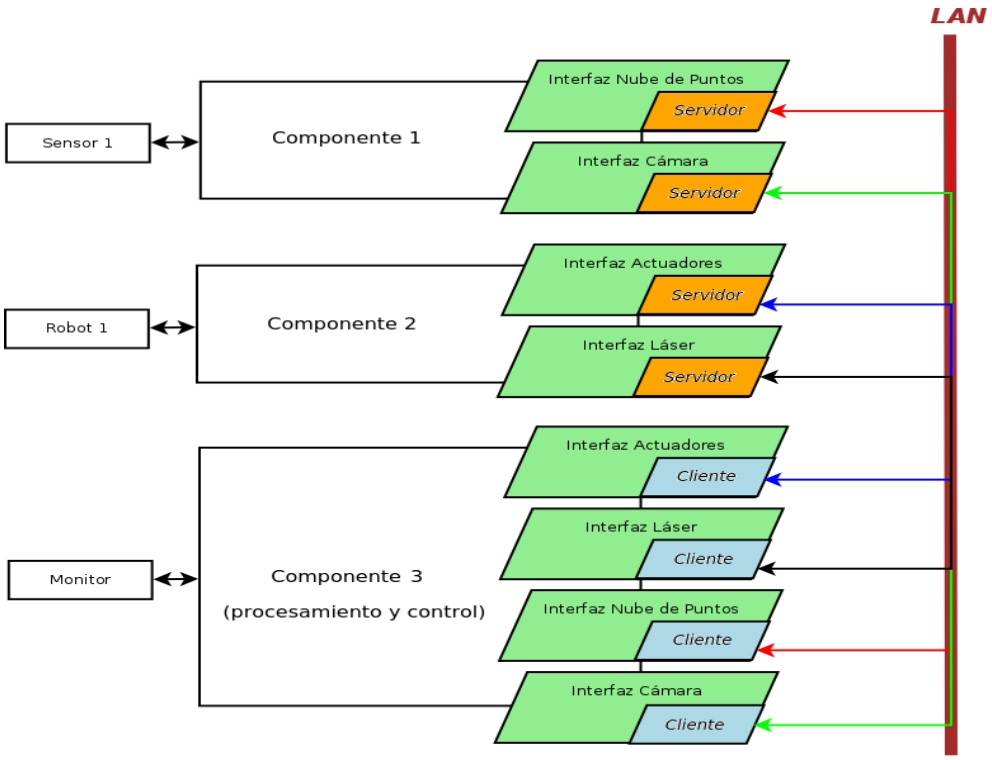


Figura 3.1: Ejemplo de arquitectura con JdeRobot

- **Eigen**: es una librería para cálculos de álgebra lineal, escrita en el lenguaje C++, que facilita el uso de matrices, vectores y algoritmos relacionados como métodos de resolución y reducción de sistemas lineales.

  Esta librería[3], de código abierto, ha sido utilizada frecuentemente en este proyecto en su versión 3.1.2 para realizar cálculos matriciales y vectoriales debido a su API sencilla y el buen rendimiento de sus operaciones.

- **OpenCV**: es una biblioteca de código abierto cuyo principal campo de aplicación es el de la visión artificial. Fue originariamente desarrollada por Intel en enero de 1999 y actualmente mantenida por Willow Garage e Itseez . Debido a que se encuentra bajo la licencia BSD, es ampliamente utilizada en aplicaciones comerciales y de investigación.

  OpenCV[4] se encuentra programado en C y C++ optimizados, centrándose en proporcionar un entorno de desarrollo fácil y altamente eficiente. Además, aprovecha las capacidades que proveen los procesadores multinúcleo y puede hacer uso de rutinas de bajo nivel específicas para procesadores Intel.

[3] http://eigen.tuxfamily.org
[4] http://opencv.org

Esta librería se encuentra disponible tanto para GNU/Linux como para Mac OS X y Windows. Cuenta con mas de 500 funciones relacionadas con el procesamiento básico de imágenes, reconocimiento de objetos, calibración de cámaras, visión en estéreo y visión en robótica, entre otras. Debido a su uso genalizado, tiene amplia documentación, foro propio y una gran comunidad de desarrolladores detrás.

Para este proyecto, se ha utilizado esta librería no tanto por su parte gráfica sino por sus estructuras de datos para manejo de imágenes y matrices, conversiones entre espacios de colores y algoritmos básicos (versión 2.4.9).

- **AprilTags**: es una librería[5] para la detección y cálculos de posición espacial de códigos de barras en dos dimensiones. Fue desarrollada por Ed Olson para aplicaciones dentro del campo de la robótica.

  Las etiquetas empleadas son similares a códigos QR pero simplificados, permitiendo la codificación de menos estados lo que aumenta su robustez. Cada etiqueta tiene un identificador único y pueden ser imprimidas en papel en una impresora corriente.

  La librería detecta cualquier etiqueta en la imágen proporcionada, proporciona un ID único a la misma y calcula su posición en la imagen. Si la cámara está calibrada adecuadamente y se conoce el tamaño físico de la etiqueta , calcula la transformación relativa entre dicha etiqueta y la cámara (orientación y posición). El software de detección es robusto a condiciones de luz y ángulo de visión.

  Actualmente encuentra disponible en los lenguajes C++, Java y C y ha sido empleada ampliamente en este proyecto de fin de carrera su versión 2.1 para la detección de balizas visuales situadas en el entorno.

- **Aruco**: se trata de una librería[6] desarrollada en la universidad de Córdoba para aplicacione de realidad aumentada basada exclusívamente en OpenCV. Al igual que AprilTags, utiliza marcadores o balizas con códigos bidimensionales que son detectados y tratados adecuadamente.

  Como resultado de las pruebas, se comprobó que esta librería daba mejores resultados en el paso a 3D desde las detecciones obtenidas en las imágenes, por lo que ha sido utilizada para tal efecto en su versión 1.2.

- **OpenGL**: 'Open Graphics Library'[7] es una especificación estándar que define una

[5] http://april.eecs.umich.edu/wiki/index.php/AprilTags
[6] http://www.uco.es/investiga/grupos/ava/node/1
[7] www.opengl.org

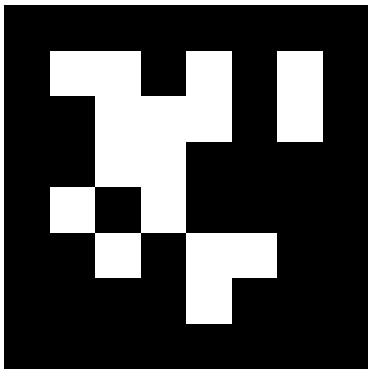


Figura 3.2: Baliza de AprilTags

API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos en 2D y 3D. A partir de esta especificación, los fabricantes de hardware crean implementaciones (librerías de funciones) que aprovechas la arquitectura física desarrollada de forma eficiente. Fue desarrollada en 1002 por Silicon Graphics Inc.

Actualmente se utiliza ampliamente en aplicaciones de diseño asistido por ordenador (CAD), realidad virtual, representación científica, visualización de información, simulaciones de vuelo, desarrollo de videojuegos, etc. Dos propósitos esenciales dentro de la filosofía de OpenGL son:

- Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, ofreciendo así una API única y uniforme.
- Ocultar las diferentes capacidades de las distintas plataformas hardware.

Durante este proyecto de fin de carrera, OpenGL ha resultado muy útil para la visualización de los resultados en 2D y 3D, permitiendo una mejor depuración de los algoritmos y representación de los mismos (versión 4.1).

- **QT creator**: es un entorno de desarrollo integrado multiplataforma que integra múltiples herramientas para facilitar la edición y depuración de código, principalmente en C++. Ha sido empleado continuamente durante este proyecto de fin de carrera por muchas de sus funcionalidades como resaltado de sintaxis y autocompletado de código, herramientas para la rápida navegación, plegado de código, depurador visual, control estático de código y estilo etc. También contiene una herramienta de creación de inter-

faces gráficas de usuario para diseñar de forma rápida e intuitiva formularios, *widgets* y diálogos que luego se dotarán de funcionalidad.

Además del entorno, se ha empleado la versión 4 de la librería QT[8] para el uso del mecanismo de señales y *slots* que proporciona. Esto sirve para comunicar hilos dentro del programa declarando que método o métodos se ejecutarán (slots) cuando se lance una señal determinada.

3.2. Proyecto industrial

Este proyecto de fin de carrera se enmarca dentro del un grupo de Robótica de la Universidad Rey Juan Carlos, encargado de desarrollar un sistema completo para una empresa real determinada en el ámbito industrial. Nuestro robot debe ser capaz de navegar de forma autónoma por una nave industrial, portando cargas pesadas, sabiendo en todo momento su posición en el espacio e interpretando el entorno para evitar cualquier tipo de colisión con obstáculos fijos o dinámicos.

El robot, denominado Autonav, deberá seguir con precisión rutas previamente establecidas por operarios a través de una aplicación de escritorio o un programa desarrollado en Android para tablets. Además se proporcionan mecanismos de parada de emergencia y teleoperación en caso de que se produzca cualquier tipo de problema. A continuación se describe la arquitectura software y hardware de nuestro sistema.

3.2.1. Hardware

El robot Autonav cuenta con las siguientes partes diferenciadas para su funcionamiento:

- Tres ruedas, dos atrás y una delante. Esta última sirve para dirigir el movimiento del robot.
- USB CAN Driver, encargado de hacer la transformación de los datos recibidos de los encoders del robot a format digital y servirlos por USB. Además, sirve como interfaz

[8] www.qt.io

de acceso para mandar órdenes a los motores de nuestro robot.

- Dos sensors tipo RGB-D, de la marca ASUS (Xtion), que se conectarán a los ordenadores por USB para ofrecer la información tanto de imagen RGB como de distancia. Se sitúan en la parte delantera del robot, mirando hacia delante pero inclinados hacia los lados en un determinado ángulo.

- Uno o varios computadores (dependiendo de la configuración) que se conectarán a los sensores y actuadores por USB y ejecutarán los componentes adecuados. Entre ellos se realizará la comunicación usando redes Wi-Fi.

- Una tablet con el sistema operativo Android para la teleoperación y establecimiento de rutas de forma más cómoda.

- Baterias, conexiones cableadas y carcasa metálica propias de un vehículo industrial.

- Un sistema de arranque con llave para inicial el robot, situado en el lateral del mismo. También consta de un botón para armar y una pantalla LCD que indica el estado del robot.

- Una seta de parada de emergencia, que podrá ser pulsada de forma manual en caso de peligro.

Además contaremos con un mapa del entorno (nave industrial) y balizas visuales, necesarios para el correcto funcionamiento de los algoritmos.

3.2.2. Software

La arquitectura software del proyecto Autonav está divida en componentes, utilizando siempre JdeRobot para la comunicación entre ellos. A continuación se describe brevemente a cada uno de ellos.

- **Componente CAN driver**: es un intermediario de comunicación entre el conector CAN y los computadores. Ofrece las interfaces 'Encoders' y 'Motors' para leer los datos de los odometría y enviar órdenes a los motores del robot, respectivamente.

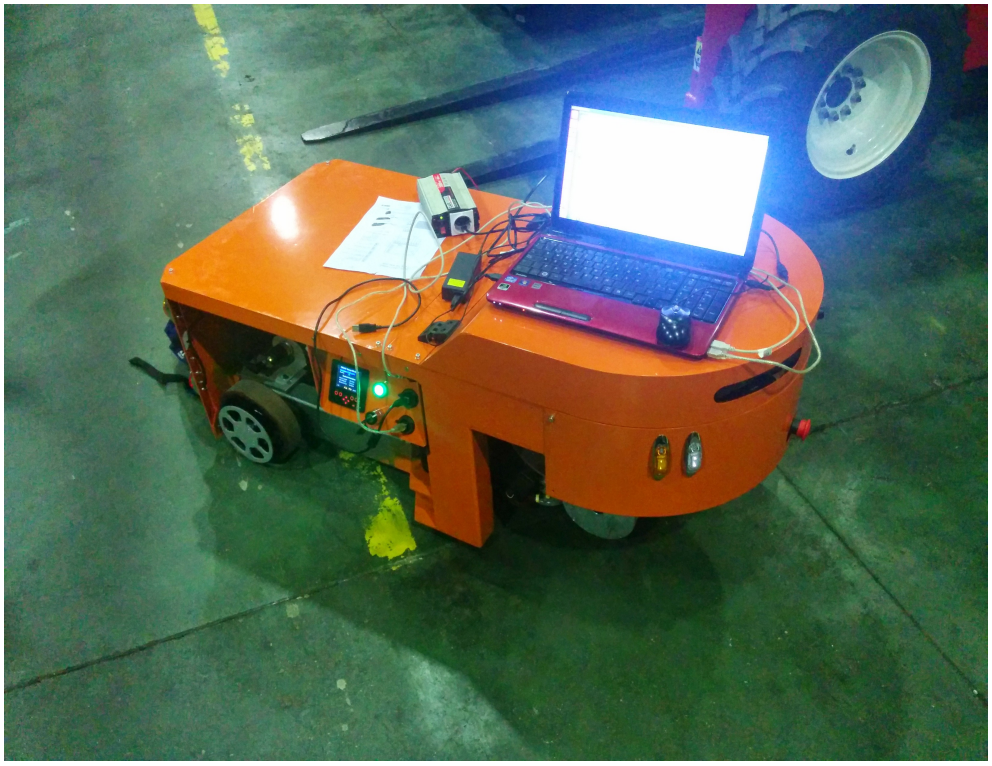


Figura 3.3: Robot Autonav en la nave industrial

- **Componente OpenniServer**: se encarga de transformar la información proviniente de cada uno de los dispositivos RGB-D y ofrecerla de forma adecuada para otros componentes vía ICE. La información útil que procesada por el sistema es:

  - Imágenes a color para la detección de balizas colocadas por el entorno.
  - Imágenes de profundidad utilizadas para calcular las nubes de puntos asociadas a cada fotograma, obteniéndose como resultado final información del entorno en tres dimensiones a partir de imágenes en 2 dimensiones. Estas nubes de puntos se utilizarán para dotar de seguridad al robot, incluyendo la funcionalidad de localizar obstáculos en tiempo real a partir de la información proveniente de los sensores visuales.

  Se lanzarán, por tanto, dos instancias de este componente (una por dispositivo).

- **Componente VisualLoc**: es el encargado de dar una estimación precisa de posición del robot en el entorno de forma continua. Utilizará información proviniente de los sensores RGB-D (imágenes) y datos odométricos de los encoders para los cálculos,

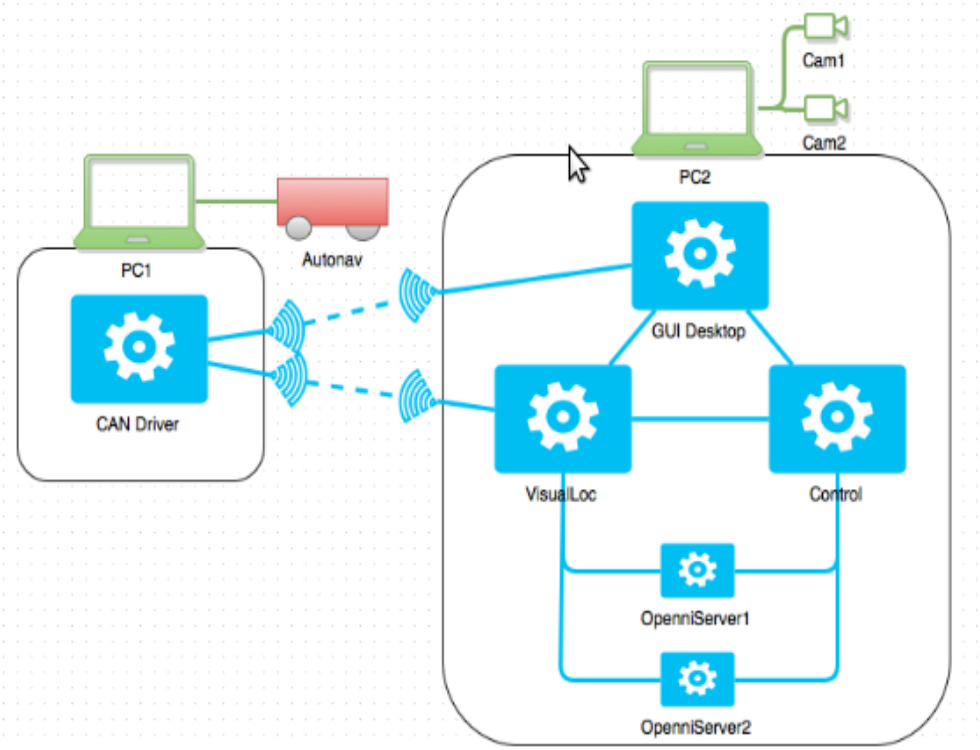


Figura 3.4: Arquitectura software de Autonav

a parte de la información conocida de la posición y orientación de las balizas en el entorno. Este es el componente desarrollado en este Proyecto de Fin de Carrera.

- **Componente de Control**: la función principal de este componente es comandar todas las acciones a través del envío de órdenes a los motores del robot a través de interfaces ICE entre este componente y CAN Driver. Además se encarga de la detección de obstáculos (seguridad) y de la ejecución del algoritmo de navegación.

  En cuanto a la parte de seguridad, este componente hace uso de la información procedente de OpenniServer, en particular de la nube de puntos. El algoritmo de seguridad calcula cuántos de los puntos de esa nube están dentro de una zona definida llamada "zona de seguridad" que comprende 2,00m hacia el frente del robot desde el punto de origen del mismo (centro del eje trasero), y 1,30m hacia derecha e izquierda respectivamente. Si el número de puntos dentro de la zona de seguridad supera un umbral definido y configurable, se considera que hay un obstáculo en la ruta del robot, por lo que Control comandará a los motores que se detengan para evitar una colisión con dicho obstáculo.

  Por su parte, el algoritmo de navegación que ejecuta este componente, está implemen-

tado mediante un controlador PID, donde se tienen en cuenta las constantes derivada y proporcional. Además, la navegación está basada en casos, de tal modo que en función del estado del robot y de la ruta en un momento concreto, el algoritmo de navegación tomará decisiones pertinentes en función de los casos programados.

- **Componente GUI Desktop**: este componente consta de una interfaz gráfica que muestra el mapa del entorno para el establecimiento manual de la ruta que deberá seguir el robot de forma autónoma. Además representará en 2D la posición estimada del robot en cada momento. También permite la teleoperación del mismo por un operador.

  Este componente es el encargado de servir como interfaz entre el sistema y el usuario humano. Lo hace a través de un GUI que hace las veces de panel de control, desde el cual se puede controlar el movimiento del robot, tanto manual, como autónomo. Muestra en todo momento el mapa del entorno en 2D y representa en él la posición estimada del robot en cada instante. Este GUI, se compone de varios elementos:

  - Control manual del robot, mediante un slider para el control de velocidad y un volante para el control de la rueda directriz.
  - Control de inicio/parada/reanude del robot.
  - Planificador de rutas y puntos de control para la navegación autónoma.
  - Visor del mapa de entorno y de la posición y orientación del robot en tiempo real.

  Además, existe una aplicación análoga desarrollada para Android, ejecutable desde tabletas y dispositivos móviles.

Para la intercomunicación entre los componentes descritos, se utilzan distintos interfaces ICE en función de la información que se desea intercambiar entre ellos. A continuación se detallan todas la interfaces ICE existentes en el sistema:

- **Encoders**: el sistema tiene varias interfaces ICE que manejan información de encoders. Los encoders son la representación de la información odométrica del robot que se maneja en el sistema. Los componentes que utilizan este tipo de interfaz son: VisualLoc, CANDriver, Control y GUI Desktop.
- **PointCloud**: este tipo de interfaz es el utilizado para transmitir la información en forma de nube de puntos proveniente de los sensores RGBD. Utilizan esta interfaz los componentes OpenniServer, Control y GUI Desktop.

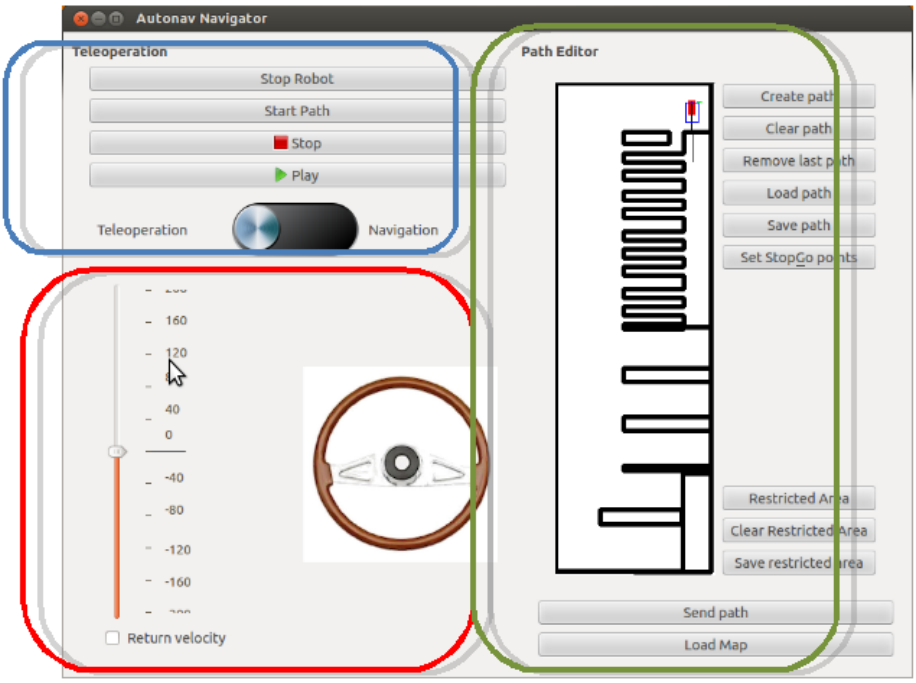


Figura 3.5: Componente GUI de Autonav

- **RGBD**: representación de las imágenes a color provenientes de los sensores RGBD. Ofrecen las imágenes captadas por los mismos para que sean procesadas de forma adecuada. Los componentes que utilizan esta interfaz son OpenniServer y VisualLoc.
- **Motors**: esta interfaz se utiliza para comandar acciones a los motores del robot. Los componentes que las utilizan son CANDriver y Control.
- **Messages**: esta interfaz creada al uso para transmitir información de la ruta en tiempo real, se utiliza para representar el objetivo más inmediato del robot mientras sigue una ruta determinada. Los componentes que utilizan esta interfaz son Control y GUI Desktop.

Capítulo 4

Componente Visualloc

Una vez presentados los requisitos que la aplicación debe cumplir, el contexto y las herramientas utilizadas en este Proyecto Fin de Carrera, se describen de forma detallada en este capítulo cada una de las partes que conforman el componente final desarrollado.

4.1. Descripción general

El principal objetivo de este componente es proporcionar una estimación de posición 3D precisa de dónde se encuentra el robot dentro de un escenario determinado. Puesto que forma parte de un sistema en tiempo real, deberá ser capaz de dar esta estimación en todo momento.

En la figura 4.1 se puede observar un diagrama de caja negra del componente Visualloc. Como entrada recibe la información procedente de dos sensores RGBD, situados en la parte delantera del robot, a través de interfaces estándar de JdeRobot de tipo *image*. Aunque estos sensores también son capaces de ofrecer una nube de puntos en tres dimensiones, nuestro algoritmo sólo usa la parte puramente de imagen RGB para sus cálculos. Además, el componente también recibe información de la odometría del robot (interfaz *encoders* de JdeRobot) para medir pequeños incrementos en la posición y orientación del robot. Respecto a la salida, Visualloc ofrece las estimaciones de posición resultantes a través de dos interfaces JdeRobot al resto de componentes del sistema: *encoders* y *pose3D*.

Figura 4.1: Diagrama de caja negra del componente Visualloc

Respecto al funcionamiento interno del componente Visualloc, se puede apreciar en la figura 4.2 que se conforma de varias partes que se alimentan unas a otras. Primero, las imágenes obtenidas de los dispositivos RGBD son procesadas y en cada iteración se extraen de ellas las balizas en 2D que estén presentes. Posteriormente, realizando cálculos geométricos, se consigue una estimación por cada baliza de la posición en 3D del robot en el mundo. Finalmente, se fusionan todas las estimaciones instantáneas, considerando los encoders de forma incremental como otra estimación individual. Por lo tanto, se consigue como resultado una única estimación precisa por instante de tiempo. El algoritmo es y debe ser robusto puesto que las decisiones de movimiento del componente de navegación se basarán en la estimación de posición y orientación proporcionada por Visualloc.

Además, el componente cuenta con una interfaz de usuario formada por ventanas donde se muestran las imágenes de cada cámara y se resaltan las balizas localizadas y un mundo en 3D interactivo para seguir la estimación del robot en cada momento y facilitar la depuración de los algoritmos.

En las siguientes secciones se detalla el funcionamiento de cada uno de los tres bloques de procesamiento del componente, además de la interfaz de usuario creada.

4.2. Detección de balizas en 2D

Como se ha indicado anteriormente, el componente Visualloc hace uso de balizas bidimensionales colocadas por todo el entorno para calcular estimaciones de posición del robot. El primer paso es detectar las balizas en las imágenes ofrecidas por ambas cámaras. Para ello se

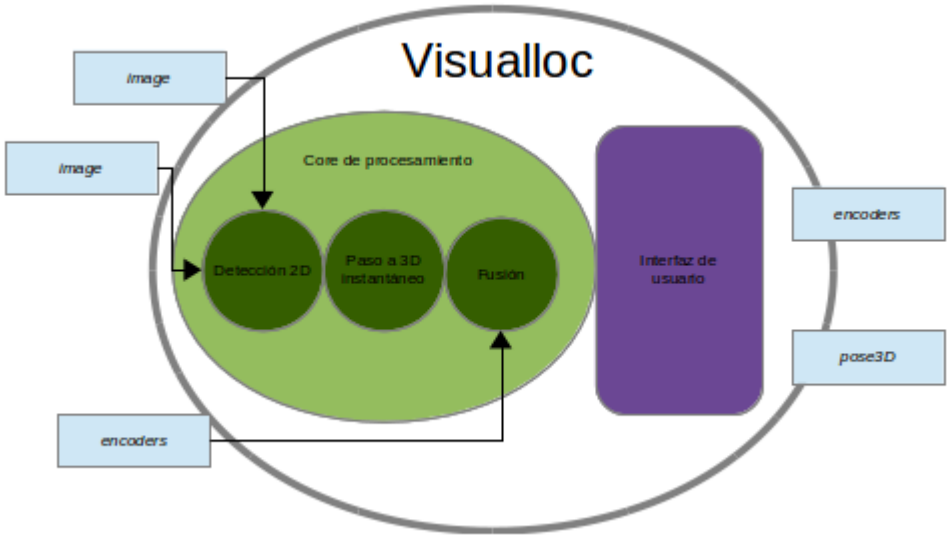


Figura 4.2: Diagrama de caja blanca del componente Visualloc

valoró el uso de diversas librerías de visión artificial, como Aruco y AprilTags, decantándonos finalmente por la segunda.

AprilTags, como se ha explicado en el capítulo de infraestructura, es una librería que permite la detección de códigos de barras bidimensionales de forma robusta a través de técnicas como la detección de bordes, filtrado, etc. Cada una de estas balizas tiene asignado un identificador individual, lo que permite su procesamiento. Como se muestra en la figura 4.3, AprilTags cuenta con diferentes conjuntos o familias de etiquetas que aportan mayor o menor robustez a cambio de un menor o mayor número de etiquetas disponibles. Finalmente se seleccionó la familia *36h11*, cuyas balizas tienen un formato 6x6, porque daban un menor número de falsos positivos durante las pruebas.

El algoritmo empleado en esta parte es el siguiente:

- En un bucle, se solicitan imágenes a las dos cámaras del sistema, a través de las interfaces JdeRobot anteriormente descritas, con una frecuencia establecida como constante. El valor de esta constante se estableció finalmente a 50 milisegundos.
- Para cada una de estas imágenes, se realiza una conversión de formato de color a BGR (utilizando OpenCV) para facilitar su mostrado posterior en ventana. En este momento

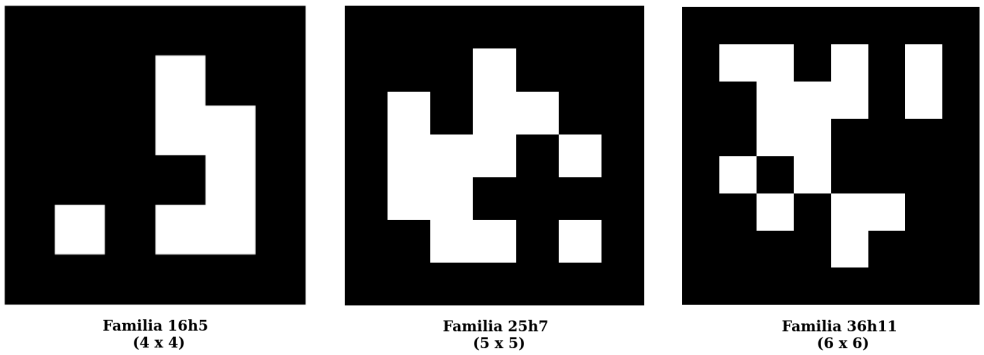


Figura 4.3: Balizas con identificador 0 de tres familias de etiquetas en AprilTags

se llama al método *processImage*, pasándole como parámetro estas dos imágenes (una por cámara).

- El método *processImage* realiza otra conversión de las dos imágenes desde BGR a escala de grises para que puedan ser procesas finalmente por el extractor de etiquetas de AprilTags. Este genera una lista por imagen con las detecciones de etiquetas extraídas de cada una de esta imágenes. Las detecciones cuentan básicamente con un número identificador y con la posición de cada uno de sus vértices en la imagen (x, y).

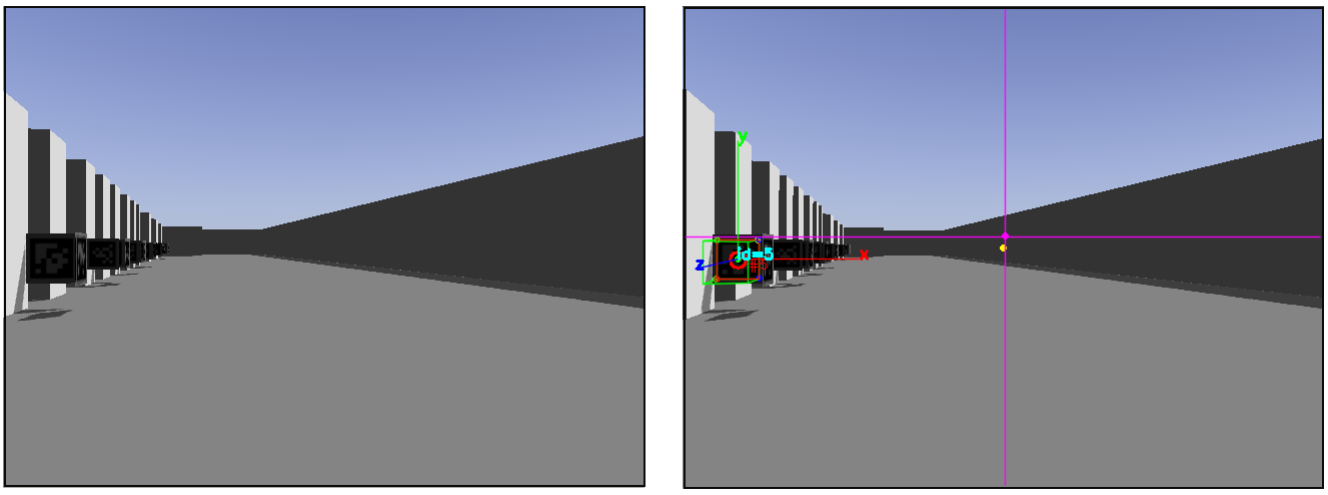


Figura 4.4: Detección de balizas en imágenes

En la figura 4.4 se puede observar que la primera baliza, con número identificador 5, está siendo detectada. Sin embargo, balizas más lejanas o con demasiado ángulo de inclinación

respecto a nuestra posición no serán detectadas por AprilTags. Además, la capacidad de detección de la librería está fuertemente determinada por las condiciones de luminosidad que tengamos en nuestro escenario.

Tras este proceso tendremos etiquetas detectadas con la suficiente información para ser procesadas por la siguiente sección del algoritmo, el paso a 3D de cada una de las detecciones.

4.3. Paso a 3D instantáneo

Para cada una de las detecciones que hemos obtenido tendremos que calcular un punto orientado en el espacio, que se corresponde con la posición y orientación del robot en el mundo absoluto para tal y cómo se ve la baliza en cuestión en la imágen. Para ello tendremos la siguiente información:

- Posición y orientación absoluta de la baliza en el espacio. Se obtienen a partir del fichero de configuración del componente.
- Tamaño real de la baliza en el mundo. Como son cuadradas, bastará saber la longitud de uno de sus lados. También obtenido por medio del fichero de configuración.
- Calibración de los parámetros intrínsecos de la cámara que realiza la detección, que influirán en gran medida en el algoritmo de cálculo de la posición y orientación de la cámara respecto a la baliza detectada. Serán introducidos a partir de un fichero de configuración en el componente.
- Detección conseguida en la sección anterior del algoritmo, (*Detección de balizas en 2D*), que nos proporciona los cuatro vértices de la baliza en 2D y un número identificador asignado.
- Posición y orientación exactas de las cámaras respecto al sistema de referencia solidario con el robot, partiendo desde su centro de gravedad (parámetros extrínsecos).

Para poder calcular la posición y orientación de nuestro robot en el mundo a partir de esta información, necesitamos conocer los distintos sistemas de referencia con los que trabajaremos. Son los cuatro siguientes:

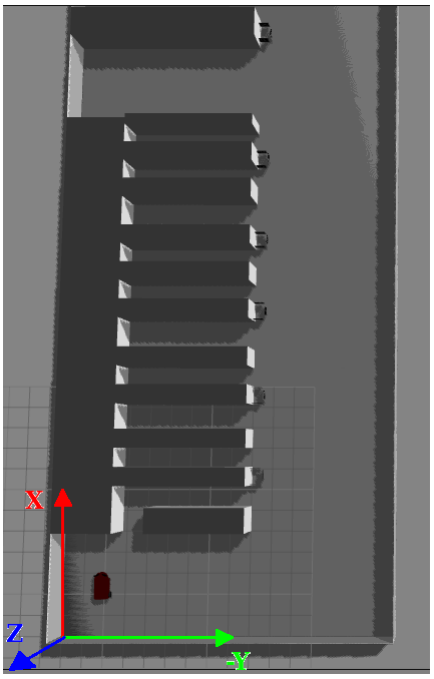


Figura 4.5: Eje de referencia absoluto en el escenario

- Sistema de referencia del escenario, o lo que es lo mismo, el sistema de referencia absoluto. Véase figura 4.5.

- Sistema de referencia solidario con la baliza. Véase figura 4.6.

- Sistema de referencia de cada una de las cámaras. Véase figura 4.7.

- Sistema de referencia del robot, que es el centro del eje entre las ruedas traseras. Véase figura 4.7.

El algoritmo de cálculo tendrá que calcular las correspondientes matrices RT (rotación-traslación) que realizarán las transformaciones entre los distintos sistemas de referencia de nuestro sistema. De esta forma, podremos hallar la matriz RT final para transformar cualquier punto en el sistema solidario con el robot a un punto en el sistema de referencia absoluto. Por tanto, la estimación final que necesitamos se puede representar como una multiplicación de matrices RT de la siguiente forma:

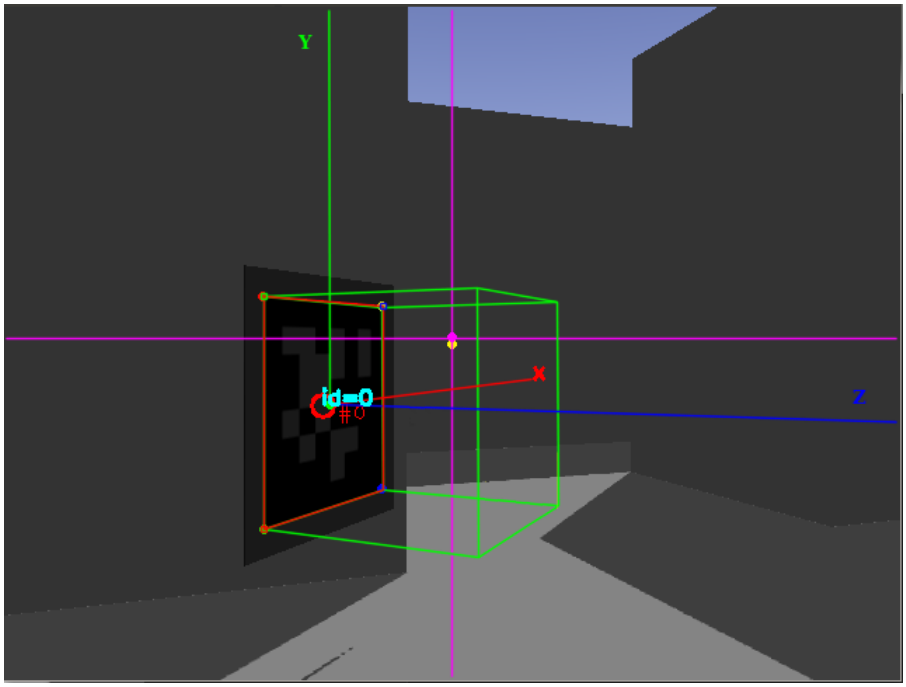


Figura 4.6: Sistema de ejes solidario con la baliza

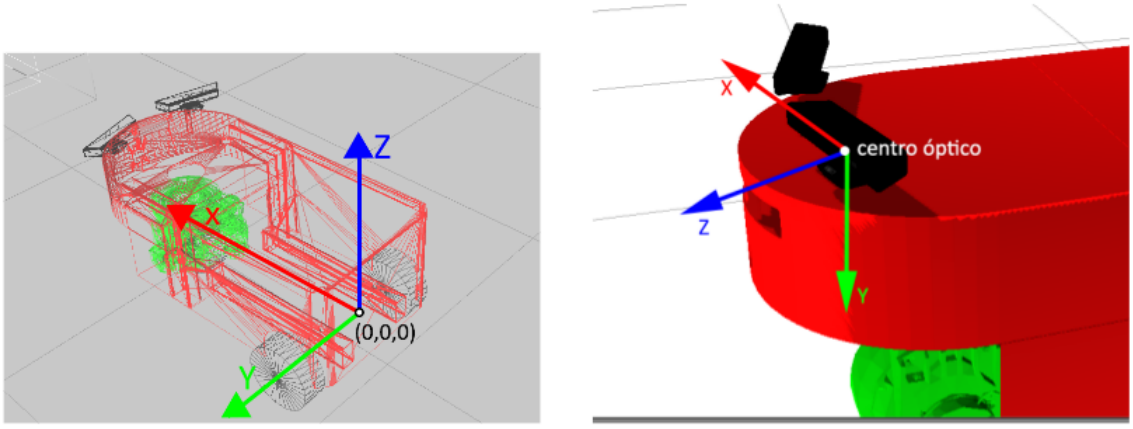


Figura 4.7: Sistemas de referencia del robot (izquierda) y de una cámara (derecha)

$$[P]^{robot}_{mundo} = [RT]^{baliza}_{mundo} * [RT]^{cámara}_{baliza} * [RT]^{robot}_{cámara} * [P]^{robot}_{robot}$$

El último elemento de la fórmula es un punto cualquiera del robot en el sistema de referencia del mismo. Cuando tengamos todas las partes de esta fórmula, este punto será para nosotros el (0,0,0), o sea, el centro del eje trasero del robot que, después de aplicarle todas las rotaciones y traslaciones, tendremos este mismo punto pero en el sistema de referencia del mundo (absoluto).

Primero, calcularemos una matriz RT que nos permite transformar puntos en el sistema de referencia de la baliza a puntos solidarios con la cámara. Para ello, utilizamos los vértices 2D calculados en la parte de detección (ordenados de forma correcta), el tamaño real de la baliza detectada y los parámetros intrínsecos de la cámara (su matriz y sus coeficientes de distorsión). La librería encargada de procesar toda esta información es Aruco, que nos proporcionará dos vectores, uno de rotación y otro de traslación, que se corresponden con la transformación de la baliza respecto a la cámara que la detectó. Para realizar este cálculo, Aruco se sirve internamente de la funcion *solvePnP* de OpenCV, que calcula la homografía correspondiente a partir de los parámetros intrínsecos de la cámara (la matriz de la cámara y los coeficientes de distorsión) y los puntos detectados en 2D. En el siguiente pseudo-código se observa cómo se conforma la matriz *RT* usando esos dos vectores. Sobre el vector de rotación se utiliza la función de *Rodrigues*, que permite construir una matriz RT a partir de 3 ángulos dados, facilitando así los cálculos ya que se pueden realizar multiplicaciones de matrices directamente:

```
//Creamos el vector de puntos con las 4 esquinas detectadas de la baliza.
Vector puntos;

puntos.agregar(deteccion[0].x, deteccion[0].y);
puntos.agregar(deteccion[1].x, deteccion[1].y);
puntos.agregar(deteccion[2].x, deteccion[2].y);
puntos.agregar(deteccion[3].x, deteccion[3].y);

//Construimos un marcador de Aruco y llamamos al calculo de los vectores.
```

```
Marker m(puntos);
m.calculateExtrinsics(lado_baliza, matriz_camara, coeficientes_distorsion);

Vector vector_translacion = m.Tvec;
Vector vector_rotacion = Rodrigues(m.Rvec);

RT_baliza_cam = construirRT(vector_translacion, vector_rotacion);
```

Puesto que la matriz RT transforma puntos de la baliza respecto a la cámara, tendremos que realizar una inversión de dicha matriz para obtener la que necesitamos para nuestra fórmula de cálculo anteriormente descrita (RT que transforma puntos de la cámara respecto a la baliza).

Como segundo paso, calculamos la matriz de rotación y traslación de la baliza (o también marcador) respecto al mundo. Para ello, si tenemos en cuenta los ejes del mundo y de la baliza, habrá que realizar un primer giro de 90º sobre su eje X para levantar dicha baliza y colocarla de forma perpendicular al suelo del escenario. Además se aplican las rotaciones correspondientes de la baliza determinada, especificadas en el fichero de configuración, a través de multiplicaciones de matrices. Igualmente, se introducirán en la matriz a calcular la posición en 3D de esta baliza en el escenario.

$$[RT]^{baliza}_{mundo} = [RT]^{90^{o}}_{ejeX} * [RT]^{r\&t}_{fich\_config}$$

Teniendo en este punto dos matrices, la primera que relaciona puntos de la baliza respecto al mundo y la segunda que transforma puntos de la baliza a puntos en el sistema de referencia de la cámara, podemos calcular la matriz RT de la cámara respecto al mundo, o sea, su posición y orientación absolutas estimadas. Para ello, debemos invertir esta última matriz:

$$[RT]^{cámara}_{mundo} = [RT]^{baliza}_{mundo} * inv\left([RT]^{baliza}_{cámara}\right)$$

Como siguiente paso, calculamos la matriz RT de la cámara respecto al robot. Para ello, teniendo en cuenta los sistemas de referencia del robot y de la cámara, realizamos dos rotaciones: una alrededor del eje X de -90º y otra alrededor del Y de 90º. A partir de la información del fichero de configuración se obtiene la posición de las cámaras relativas al centro del robot y su orientación, que se añaden al calculo matricial de forma análoga al resto de partes del algoritmo (multiplicación de matrices RT):

$$[RT]_{robot}^{cámara} = [RT]_{ejeX}^{-90^{\circ}} * [RT]_{ejeY}^{90^{\circ}} * [RT]_{fich\_config}^{r\&t\_cam}$$

En este punto ya estamos en disposición de calcular la matriz RT final que permite transformar puntos en el sistema de referencia del robot a puntos en el mundo (sistema de referencia absoluto). Habrá que invertir la matriz de cámara respecto al robot recién calculada para obtener la matriz de robot respecto a cámara y poder realizar el cálculo matricial correctamente:

$$[RT]_{mundo}^{robot} = [RT]_{mundo}^{cámara} * inv\left([RT]_{robot}^{cámara}\right)$$

Por tanto, para obtener la posición final estimada de nuestro robot en el sistema de referencia absoluto (siempre para la baliza y cámara tenidas en cuenta en el cálculo), bastará con realizar el siguiente cálculo, utilizando como punto de referencia del robot el (0,0,0):

$$[P]_{mundo}^{robot} = [RT]_{mundo}^{robot} * [P]_{robot}^{robot}$$

Para calcular la orientación estimada de nuestro robot en el mundo, tendremos que descomponer la matriz RT del robot respecto al mundo, quedándonos con la parte de rotación para aplicar sobre ella la función inversa de *Rodríges* y obtener así los ángulos de rotación en los tres ejes. En el siguiente pseudo-código se muestra la descomposición de la matriz para obtener la rotación aplicada y otra forma de obtener la posición final estimada del robot en el mundo.

```
Matriz RT_RM = obtenerRT_robotEnMundo();
Matriz R_RM; //solo rotacion del robot en el mundo
R_RM(0,0)=RT_RM(0,0);
R_RM(1,0)=RT_RM(1,0);
R_RM(2,0)=RT_RM(2,0);
R_RM(0,1)=RT_RM(0,1);
R_RM(1,1)=RT_RM(1,1);
R_RM(2,1)=RT_RM(2,1);
R_RM(0,2)=RT_RM(0,2);
R_RM(1,2)=RT_RM(1,2);
R_RM(2,2)=RT_RM(2,2);

Vector T_RM; //solo traslacion del robot en el mundo
T_RM(0)=RT_RM(0,3);
T_RM(1)=RT_RM(1,3);
T_RM(2)=RT_RM(2,3);

Vector R_RM_vec;
Rodrigues(R_RM,R_RM_vec);

//Posicion y orientacion finales de la estimacion
establecerEstimacion(T_RM, R_RM_vec);
```

De esta forma conseguimos una estimación instantánea, por baliza detectada en alguna de las imágenes, de dónde se encuentra el robot dentro del mundo y con que orientación.

4.4. Fusión de estimaciones del robot en el mundo

Después de haber conseguido una posición y orientación estimada del robot en el mundo por cada baliza dectectada en cada una de las dos cámaras en un instante de tiempo, se plantea el problema de cómo obtener una única estimación, enriquecida por todas las disponibles para aumentar la precisión y robustez de la misma.

Para ello, se realizará una primera fusión de todas las estimaciones provinientes de las cámaras a través de medias ponderadas, en las que el peso de cada estimación en el cómputo estará relacionado con la distancia a la que se encontraba la baliza de la cámara en el momento de la medición. Después se utilizará información de los encoders del robot para mejorar aún más la estimación, tirándo de ella incluso cuando no se aprecia ninguna baliza en las cámaras (po oclusiones o porque en ese momento no haya ninguna en los alrededores).

4.4.1. Fusión de observaciones visuales

El primer paso es, por tanto, ajustar el peso que tendrá cada estimación en el cálculo de la media ponderada. El algoritmo calcula la distancia entre la baliza y la cámara y se establecen tres rangos, en los cuáles el peso será mayor si la distancia entre la baliza y la cámara es menor. Estos rangos y pesos fueron ajustados empíricamente hasta obtener combinaciones razonables. Pueden variar dependiendo del entorno para el que esté diseñado el robot. Por ejemplo, en el entorno simulado en Gazebo podremos dar mayor peso a estimaciones más lejanas que en un entorno real.

Ahora que tenemos todas las estimaciones procedentes de la visión con sus pesos, procedemos a realizar el cálculo de la suma ponderada entre todas ellas. La media se calculará de la siguiente forma:

- Para el cálculo de la posición, por cada componente (x, y, z) se realiza una multiplicación del valor de la misma por el peso correspondiente para cada estimación. Además se suman todos estos valores parciales y, finalmente, se divide esta suma por la suma de los pesos de todas las estimaciones tenidas en cuenta. Por ejemplo, para la componente x el cálculo podría expresarse como:

$$\frac{\sum_{i=0}^{n-1}(x_i * w_i)}{\sum_{i=0}^{n-1}(w_i)}$$

siendo 'n' el número de estimaciones tenidas en cuenta y 'w' el peso de la estimación.

- Para el cálculo de la orientación, tendremos tantos ángulos de inclinación en el plano 'xy' como estimaciones. Este es el único plano relevante para nuestro robot puesto que es sobre el que se desplaza. Descompondremos cada ángulo en su seno y coseno, que irán cada uno multiplicado por el peso correspondiente de su estimación. Se realiza una suma de todos los senos ponderados por un lado y de todos los cosenos ponderados por otro, para finalmente calcular la tangente entre estos dos valores y su arcotangente para obtener el ángulo de orientación final. Matemáticamente, se puede expresar como:

$$\operatorname{arc\,tg}\left(\frac{\sum_{i=0}^{n-1}(\operatorname{sen}(\alpha_i) * w_i)}{\sum_{i=0}^{n-1}(\cos(\alpha_i) * w_i)}\right)$$

  siendo 'n' el número de estimaciones tenidas en cuenta, 'α' el ángulo de orientación en el plano 'xy' del sistema de referencia absoluto y 'w' el peso de la estimación.

  Esta forma de sumar ángulos se ha llevado a cabo así debido a la discontinuidad de los ángulos al pasar una vuelta completa ($359^o + 1^o = 0^o$). Por ello no se realiza una suma poderada común, sumando todos los ángulos (multiplicados por sus pesos) y dividiendo por la suma de los pesos, como sí se ha hecho para calcular la posición. Si se hiciera de esa manera, como bien se representa en la figura 4.8, la suma de, por ejemplo, 359^o con 1^o sería $(359+1)/2 = 180^o$ (flecha roja) en vez del resultado correcto de la suma, que son 0^o (flecha verde).

- El peso de la estimación final consideraremos que es el máximo entre las estimaciones tenidas en cuenta, puesto que es la máxima calidad obtenida en esa fusión.

4.4.2. Filtro de espúreos

Durante la fusión de estimaciones visuales anterior, se utiliza la función *outsidersFilter* sobre dichas estimaciones antes de ser fusionadas. Esta función actúa como un filtro de

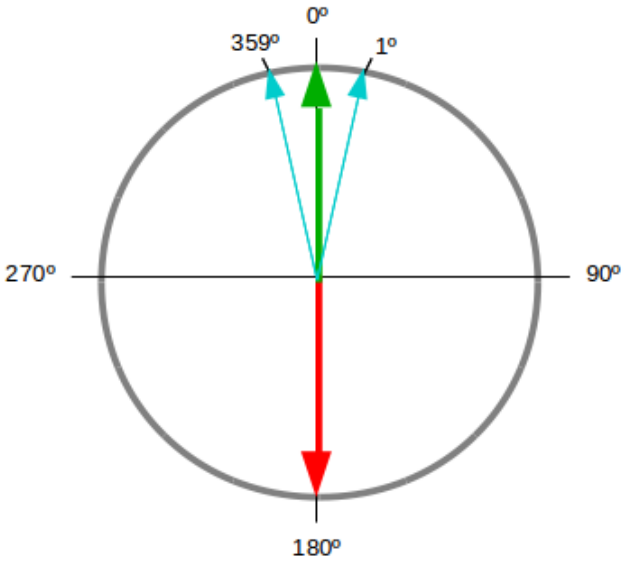


Figura 4.8: Discontinuidad de ángulos (suma de 359º con 1º)

estimaciones que se alejan demasiado de la última posición y orientación que consideramos correcta. Básicamente, se establece una zona o disco alrededor de la estimación previa y se considera que la estimación actual es buena si se encuetra dentro. Es decir, si obtenemos una estimación que se aleja demasiado de donde creemos que se encuentra el robot, esta será descartada automáticamente para evitar saltos y ruido en la fusión de estimaciones.

Estas estimaciones erróneas, que son descartadas por nuestro filtro, pueden aparecer por diversas razones como las siguientes:

- La función *solvePnP* puede entregar malas soluciones esporádicamente.
- Las balizas lejanas son más difíciles de tratar puesto que, cuando son detectadas, ocupan muchos menos píxeles en la imagen RGB que las balizas cercanas, lo que hace que empeore considerablemente la estimación, llegando al punto de determinar posiciones y orientaciones muy alejadas de la realidad.
- En ocasiones, el algorimo de detección puede confundir partes del entorno con balizas, lo que muy probablemente entregaría una estimación alejada de la buena.

En la figura 4.9 se representa la estimación anteriormente obtenida de la posición y orientación de nuestro robot en el mundo (flecha azul). Alrededor de este se establece el disco

de filtrado, también en azul, dentro del cuál se podrán situar estimaciones que consideraremos correctas. El robot con la flecha verde se considera, por tanto, una estimación válida para el siguiente instante de tiempo y se tendrá en cuenta en la fusión con un peso determinado. Sin embargo, el robot con una flecha roja se considera una estimación no válida puesto que se encuentra demasiado lejos de la anterior (fuera del disco).

Para la posición del robot, se ha establecido un disco de un radio de 3 metros para realizar los descartes y para la orientación se establece un máximo de 30 grados de desviación en la estimación. En el siguiente pseudo-código se puede seguir el proceso detallado de filtrado de estimaciones, pasadas como parámetro a la función:

```
funcion outsidersFilter(Estimacion e){

  Double diff = (e->orientacion) - orientacionActual;

  if (diff < 0) {
    diff += 2*PI;
  }
  if (diff > M_PI) {
    diff = 2*PI - diff;
  }

  Double dist = sqrt( pow(e->x - xActual, 2) + pow(e->y - yActual, 2) );

  if (diff > anguloDisco) {
    return 0; //angulo lejano descartado
  }
  else if (abs(dist) > radioDisco){
    return 0; //distancia lejana descartada
  }
  else{
     return e;
  }
}
```

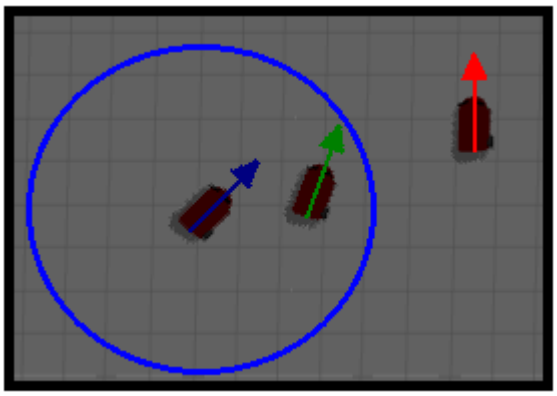


Figura 4.9: Filtro de estimaciones espúreas

4.4.3. Incorporación de la odometría

En este punto hemos conseguido una única estimación procedente de la fusión entre estimaciones correspondientes a ambas cámaras. Ahora incluiremos la información procedente de los encoders para enriquecer nuestra estimación.

Primero, procedemos a calcular la estimación procedente de los encoders. Para ello, calculamos el incremento producido entre la última posición y orientación dadas por la odometría y las que da esta en el instante actual. Se realiza, por tanto, la diferencia tanto para posición como para orientación (en la fórmula siguiente se agrupan ambos términos bajo la palabra *pose* para mayor claridad). El valor de la estimación en el eje z no se tendrá en cuenta puesto que el robot se mueve en el plano xy.

$$\left.\begin{array}{l} Pose_{odom}(t) \\ Pose_{odom}(t-1) \end{array}\right\} = \Delta_{odom}(t)$$

Después de calcular el incremento dado por la odometría, se añade este incremento a la estimación final obtenida en el instante de tiempo anterior. En alto nivel podría representarse como:

$$Estimación_{odom}(t) = Estimación_{final}(t-1) + \Delta_{odom}(t)$$

Puesto que los incrementos se dan respecto al eje del robot, hay que realizar cálculos trigonométricos para poder añadirlos a las componentes x e y de la estimación final anterior, en el sistema de referencia absoluto.

Como segundo paso, daremos un peso adecuado a la estimación que hemos obtenido de los encoders. En nuestro caso, si el peso de la estimación visual final anteriormente obtenida se corresponde con el de la distancia más cercana entre baliza y cámara (entre 0 y 1 metros), el peso de la estimación de los encoders será cero; es decir, no se tomará en cuenta. De esta forma conseguimos corregir la posición y estimación del robot más rápido cuando este detecta balizas de las que estamos bastante seguros de su precisión. En el resto de los casos, utilizaremos una constante para establecer el peso de la estimación, ajustado empíricamente durante las pruebas (para nuestro robot es de 1,5), que depende de la calidad de los encoders.

Como tercer y último paso, procedemos a fusionar la estimación visual final y la estimación procedente de la odometría. Simplemente, utilizamos la misma fusión a través de medias ponderadas anteriormente usada para agregar todas las estimaciónes procedentes de las cámaras. Sin embargo, esta vez sólo habrá que fusionar dos estimaciones: la procedente de la parte visual y la procedente de los encoders, cada una con sus pesos previamente calculados.

De esta forma, si no hay ninguna estimación visual disponible en la iteración actual (debido a oclusiones o balizas demasiado lejanas o mal colocadas), la estimación final obedecerá a la proporcionada por los enconders puesto que en ese caso sería la única fuente de información de autolocalización. Por tanto conseguimos, así, continuidad en las estimaciones dadas por el algoritmo obtenemos una estimación final adecuada que responde al movimiento del robot.

Como resultado tendremos siempre una estimación de posición y orientación de nuestro robot en el mundo, cuya precisión dependerá de factores como la densidad de balizas en el mundo, la calidad de los encoders del robot, etc.

4.5. Interfaz gráfica de usuario

Con el propósito de facilitar el uso del componente Visualloc a posibles usuarios se ha desarrollado una interfaz de usuario que permite interpretar mejor los datos y resultados obtenidos de autolocalización. Además, esta interfaz ha resultado de gran ayuda durante las fases de desarrollo para poder depurar errores y afinar el algoritmo.

Principalmente, se puede dividir la interfaz en dos partes. La primera son dos ventanas, una correspondiente a cada cámara abordo del robot, en las que se visualizan los fotogramas que estan procesando estas cámaras. Además, cuando una baliza es detectada en alguna de ellas, esta es resaltada en la imagen, recuadrándola y mostrando los ejes de la misma. Tal y como se muestra en la figura 4.10, la baliza es recuadrada con un cubo en verde que sirve también para darnos una idea de la orientación de la misma. Los vértices son resaltados con puntos y los ejes de la baliza se dibujan siguiendo el siguiente criterio de color: rojo para el eje x, verde para el eje y y azul para el eje z. Además, se incluye en la imágen el número identificador de la baliza, lo que facilita la depuración de posibles errores.

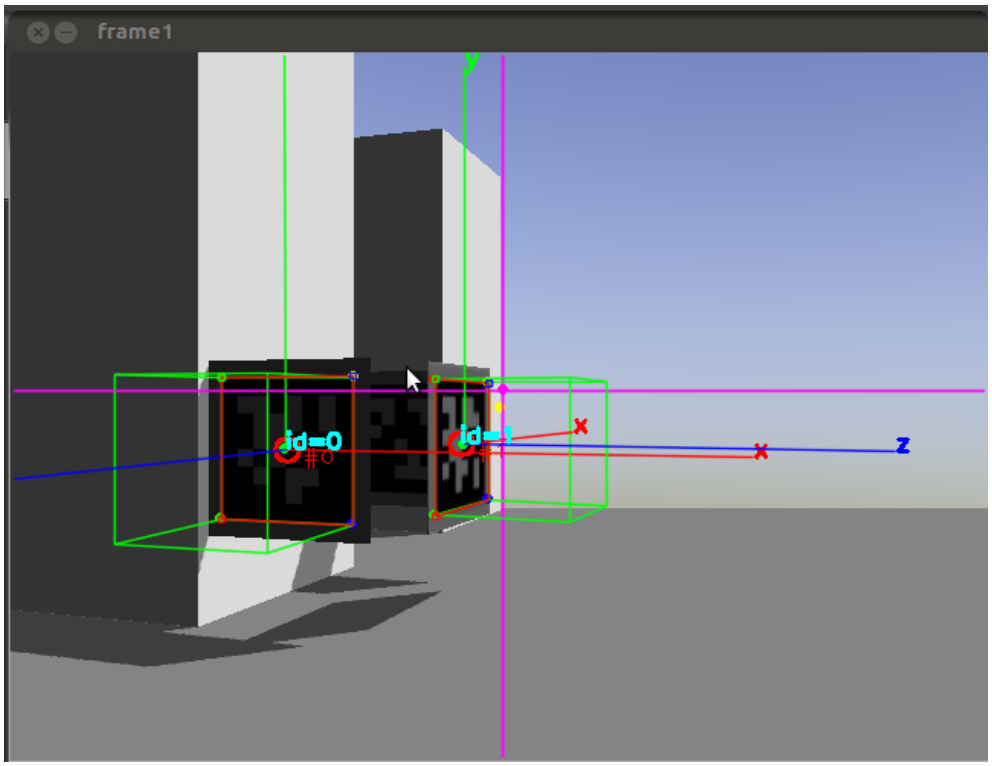
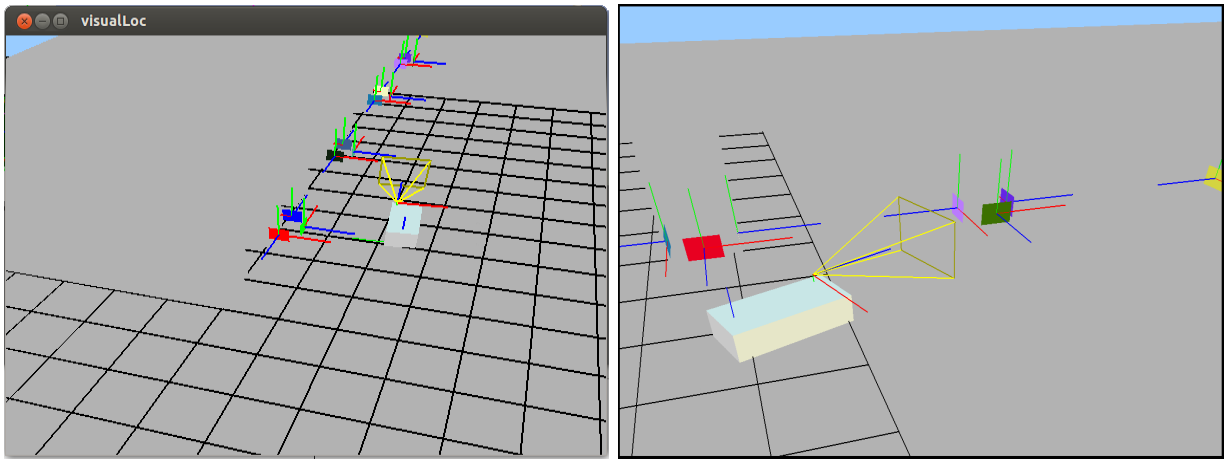


Figura 4.10: Cámara capturando dos balizas en el simulador

La segunda parte de la interfaz es un mundo tridimensional desarrollado en openGL (figura 4.11), en el que se visualiza una simplificacion del robot, las balizas tal y como están posicionadas en el fichero de configuración y las paredes del entorno. Conforme avanza el robot real, el algoritmo calculará una estimación de la posición y orientación del mismo y lo representará en el mundo 3D.

Además, cuando una baliza se detecte en alguna de las cámaras, aparecerá en el mundo openGL una pirámide de visión representando la posición y orientación de la cámara que la detectó. En la figura 4.11 se puede apreciar en color amarillo la pirámide de visión al detectar una de las balizas (representadas como parches de colores variados). Esta pirámide tiene su punta colocada sobre el punto en el robot donde está colocada la cámara que la detectó, y además en la orientación en la que se encuentra dicha cámara. El robot (caja de color blanco-grisáceo) ha corregido su posición en el mundo a consecuencia de la visualización.

Como se puede observar, se ha mantenido el mismo criterio de color indicado anteriormente para representar los ejes de cada uno de los sistemas de referencia (rojo para el eje x, verde para el eje y y azul para el eje z). De esta forma se facilita la comprobación de la orientación correcta de las balizas, el robot y la cámara.



(a) Captura de baliza (visto desde atrás)

(b) Captura de baliza (visto desde la derecha del robot)

Figura 4.11: Mundo 3D en openGL

La comunicación entre el algoritmo de autolocalización y este mundo en openGL se realiza a través del mecanismo de *signals* y *slots* propio de QT, también utilizado para comunicar

otras clases dentro del componente. Básicamente, se define en la clase de procesamiento la señal que se emitirá con, en este caso, la posición final estimada del robot. En la parte gráfica, se define el *slot* que recibirá dicha señal, que será el encargado de procesar la información recibida. Ambas partes se relaccionan con la siguente llamada en el programa principal:

```
QObject::connect(
        tag_processor, SIGNAL(estimatedCamPose3d(Pose3D*)),
        mundo3D, SLOT(paintEstimatedCamPos(Pose3D*))
                );
```

Esta ventana con el mundo en tres dimensiones puede ser ocultada a través de un flag en el fichero de configuración, mejorando así el rendimiento del componente.

Capítulo 5

Pruebas

Después de haber descrito tanto el contexto general como el funcionamiento interno del componente Visualloc, pasamos a describir el entorno de pruebas y depuración de los algoritmos además de mostrar algunas de las pruebas realizadas.

5.1. Escenario de pruebas

Durante la realización de este proyecto de fin de carrera se han realizado numerosas pruebas de funcionamiento y precisión en la nave industrial real para la que está diseñada el robot. Sin embargo, para poder desarrollar y refinar correctamente el algoritmo, se creó un mundo virtual tridimensional dentro del simulador de Gazebo que imita la configuración del escenario real.

Este mundo se creó generando un fichero *.world* procesable por Gazebo, que contiene diferentes modelos de elementos definidos a través de ficheros *.sdf*. Dentro de estos ficheros se introducen enlaces a ficheros *.dae* de *Collada*, que contienen modelados de partes del robot por programas externos, para proporcionar gráficos más completos a nuestros modelos en Gazebo. En la figura 5.1 se describe de forma general la configuración de nuestro mundo simulado como modelos anidados. Además, se incluyen los nombres de los plugins que aportan funcionalidad a cada una de las partes:

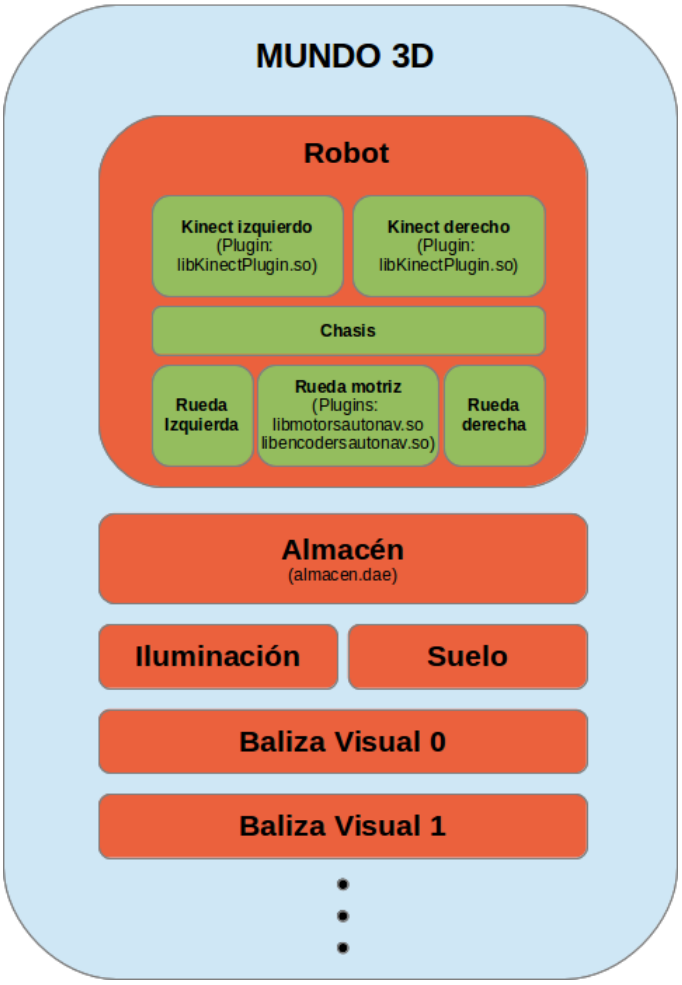


Figura 5.1: Esquema del mundo simulado en Gazebo

Como se puede observar en la figura 5.2, el mundo en Gazebo tiene los muros correspondientes al escenario real y un suelo de cuadrícula, donde cada cuadrado representa un metro cuadrado en la realidad. Se han introducido varias familias de balizas de AprilTags como modelos de Gazebo. Finalmente se eligió la familia de balizas 6x6 que pueden verse distribuidas por las paredes en este mundo virtual, alrededor de las columnas en el lado izquierdo y sobre la pared derecha de la imagen.

El robot encargado de desplazarse por el entorno también se encuentra en el escenario 3D. Su modelo consta de varios otros modelos que lo conforman. Principalmente, el robot cuenta con dos modelos de kinect situados y orientados correctamente sobre el robot. Estos modelos tienen un *plugin* enlazado que les da funcionalidad real, es decir, permite capturar imágenes y datos RGBD del mundo virtual y servirlas a través de interfaces ICE de JdeRobot que permitirán interactuar con nuestro componente. El robot simulado también está conformado por un chasis, dos ruedas traseras y una rueda motriz delantera. El modelo de esta última rueda está enlazado con dos plugins: uno que permite generar y acceder desde el exterior a la información proviniente de la odometría del robot y otro que hace posible dar movimiento a este robot simulado desde otros componentes.

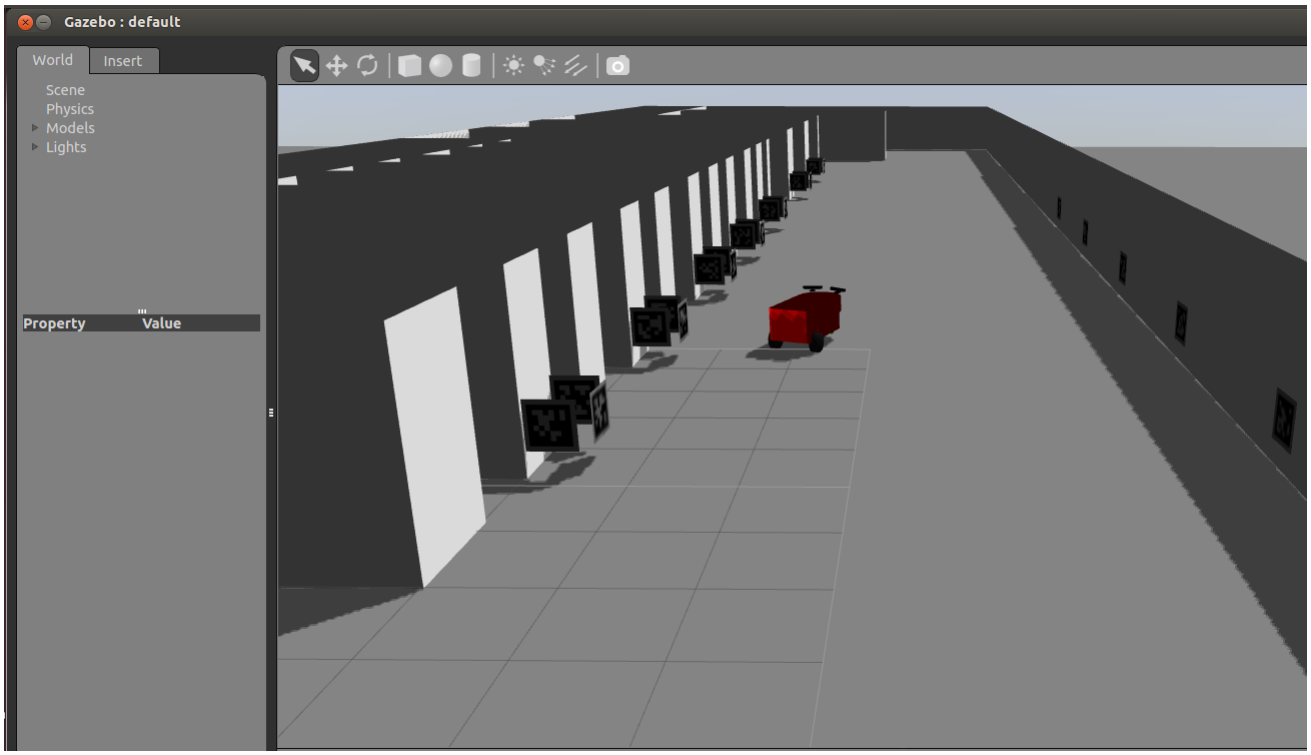


Figura 5.2: Mundo simulado en Gazebo

5.2. Recorridos con autolocalización

En esta sección presentaremos varios ejemplos de funcionamiento del componente Visualloc. Además se explicarán el desarrollo y resultados obtenidos durante la ejecución de estos casos de prueba. Primero comentaremos experimentos realizados en el simulador y, posteriormente, mostraremos pruebas sobre el robot real en el entorno industrial.

5.2.1. Recorridos en simulador

En estas pruebas realizados en el simulador Gazebo con nuestro componente, moveremos el robot desde la posición inicial del mismo en una esquina del almacén, realizando un recorrido por el mismo. Puesto que tenemos balizas a ambos lados de su camino, el robot irá detectándolas con las dos cámaras y realizando los cálculos descritos en el capítulo anterior para calcular su posición en el entorno.

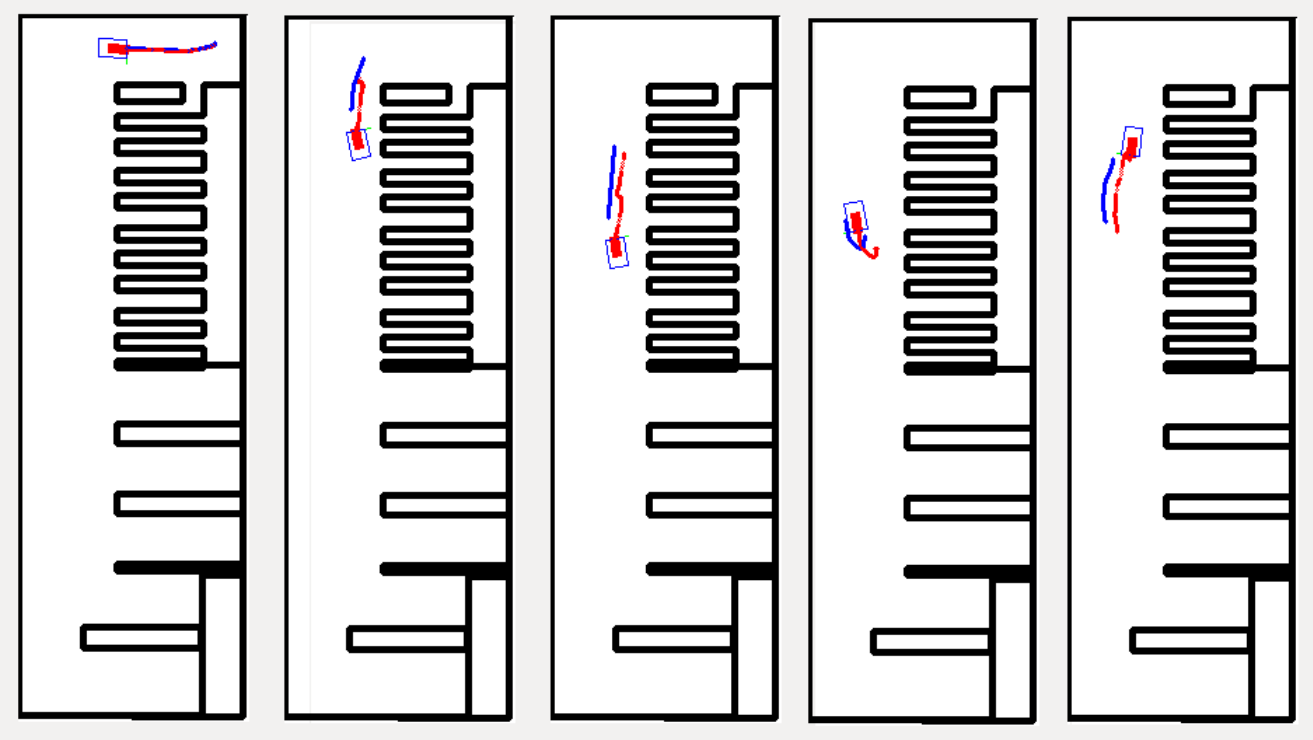


Figura 5.3: Pruebas sobre el algoritmo Visualloc

En la figura 5.3 se puede observar una comparativa entre la estimación de posición dada

solamente por los encoders (con ruido) y la ofrecida finalmente por nuestro componente Visualloc, que tiene en cuenta tanto la información visual como la odométrica. El camino azul es el que pertenece a estos encoders y el rojo el resultado del algoritmo de Visualloc.

En la primera imagen de la figura, hemos arrancado el algoritmo y no hemos detectado aún ninguna baliza. Debido a esto, la estimación odométrica y la proporcionada por nuestro componente coinciden totalmente. Por lo tanto, se demuestra que aunque no tengamos balizas en nuestro alrededor temporalmente, tendremos una estimación de posición y orientación.

En la segunda y tercera imagen pasamos por una zona del almacén con balizas que el robot detecta. Como resultado, la estimación de Visualloc se desdobla de la de los encoders, reajustándose a la posición real del robot en el mundo 3D y corrigiendo el error arrastrado por la odometría.

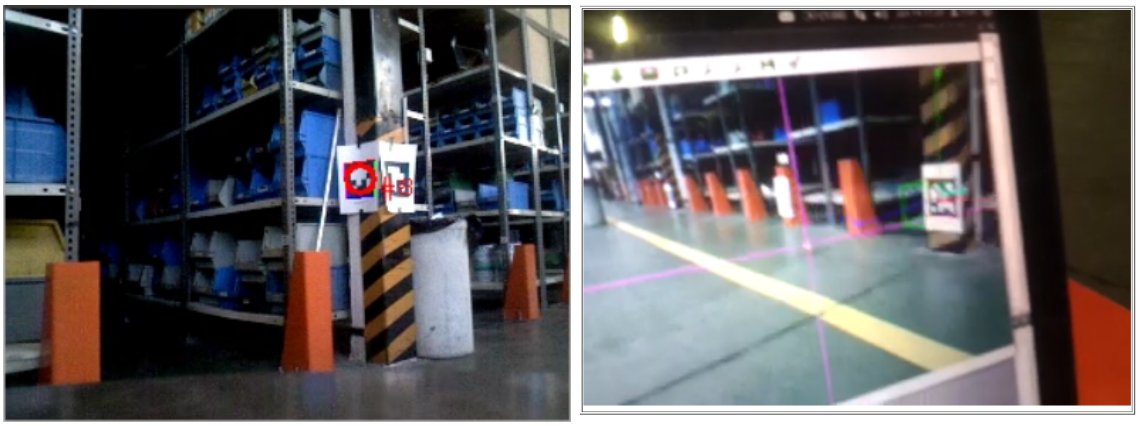
En la cuarta imagen de la figura, el robot realiza una media vuelta. Se puede observar que, debido a que los encoders son incrementales, estos siguen arrastrando su error e incluso aumentándolo debido al cambio de dirección realizado. Sin embargo, nuestro componente es capaz de corregirlo gracias al balizado del entorno que es procesado por su algoritmo.

La última imagen muestra el desfase entre la estimación puramente odométrica, con el ruido propio de este tipo de sensores, y la proporcionada por nuestro componente, que es muy próxima a la correcta, después de una vuelta completa por el almacén. Por lo tanto, se deduce que si el robot tuviera que realizar varios recorridos por el entorno sólo con los encoders como fuente de información de autolocalización, esto provocaría tal error que nuestro robot se perdería y empezaría a colisionar con el entorno.

5.2.2. Recorridos en robot real

Además de las pruebas realizadas en el simulador, se han llevado a cabo numerosos experimentos sobre el robot real en el entorno industrial, lo que ha permitido calibrar el algoritmo para una aplicación real.

En la figura 5.4 se puede apreciar la detección de balizas colocadas en una columna de la nave industrial por la que se desplaza nuestro robot, tanto en una imagen en parado (izquierda) como otra en movimiento (derecha).



(a) Detección de baliza en entorno real (b) Pantalla del ordenador ejecutando Visualloc

Figura 5.4: Detección de balizas por Visualloc en el entorno real

Una de las múltiples pruebas que se realizaron fue comparar la estimación devuelta por el algoritmo Visualloc, representada en el mundo 3D generado en OpenGL, con la posición o orientación reales del robot en el escenario. En la figura 5.5 se observa, en la parte derecha, la detección de una de las balizas con la cámara izquierda del robot. La baliza aparece resaltada en la imagen y se coloca el número identificador de la baliza. En la parte izquierda de la misma figura tenemos el mundo 3D para depuración, en el que se representa la estimación calculada a través de una caja de color grisáceo (robot) que nos da una buena visión de la posición y la orientación. Además en amarillo tenemos la pirámide de visión de la cámara que está detectando la baliza, por lo que se puede comprobar que la estimación devuelta es correcta.

Una de las pruebas más exhaustivas realizas fue realizar con el robot varias vueltas dentro de la nave siguiendo el mismo recorrido. Esto se realizó a través del componente de interación con el usuario, que permite tanto teleoperar el robot como establecer rutas que éste deberá recorrer de forma autónoma. Por tanto, para que esto sea posible una parte crítica es que la estimación de posición calculada por nuestro componente sea precisa y robusta. En esta prueba se estableció una ruta circular, por lo que el robot debería pasar, en cada una de las vueltas, por el mismo sitio exacto.

Como se observa en la figura 5.6, el robot pasó durante esta prueba por la misma posición y con la misma orientación aproximadas en cada una de las vueltas, salvo por pequeños márgenes de error. Por lo tanto, esta fue una buena demostración de la precisión del algoritmo.

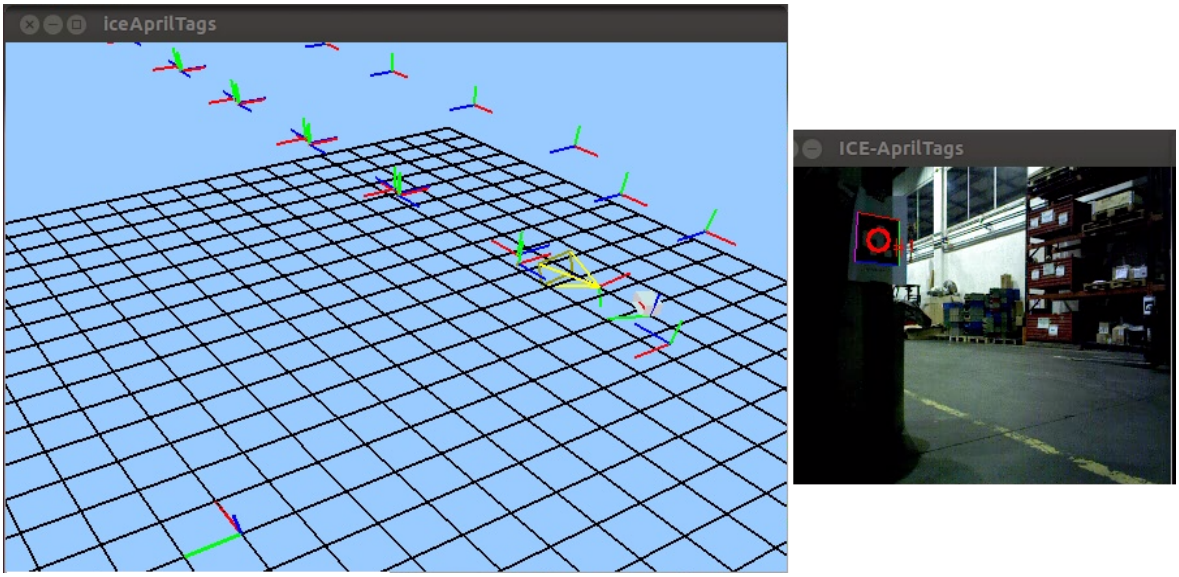
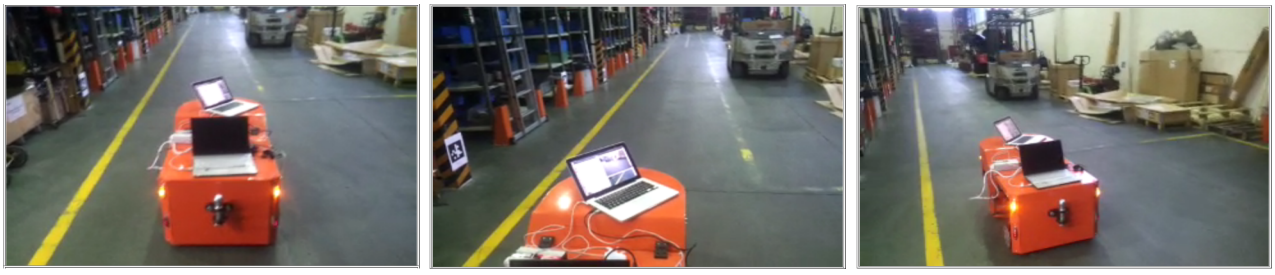


Figura 5.5: Estimación de posición obtenida por Visualloc a partir de una de las balizas



(a) Vuelta nº 1 (b) Vuelta nº 2 (c) Vuelta nº 3

Figura 5.6: Misma parte del recorrido del robot en diferente vuelta

Como resultado, nuestro componente de autolocalización tiene unos márgenes de error de unos 10 cm respecto al camino real seguido por el robot, lo que lo hace suficientemente robusto para aplicaciones reales.

5.3. Análisis de coste computacional

En el capítulo 4 se ha explicado en detalle el funcionamiento interno del componente de autolocalización visual desarrollado en este proyecto de fin de carrera y, tras haber descrito el escenario de pruebas y algunos ejemplos en secciones anteriores, pasamos a realizar un breve análisis de coste computacional del algoritmo. Esto nos permitirá conocer, además del coste general, que partes son las más costosas y en qué momentos determinados.

Como se explicó en el capítulo de descripción del componente Visualloc, éste se conforma de varias partes diferenciadas. Los costes en tiempo de cada una de estas partes están influenciados por la cantidad de balizas presentes en cada una de las imágenes de las cámaras. Teniendo una baliza detectada en una de las imágenes, el coste total del algoritmo de autolocalización oscila entre 124 y 127 milisegundos. Los tiempos consumidos en las mismas condiciones, desglosados por partes, son los siguientes:

- **Detección de balizas en 2D en las imágenes**: entre 107 y 108 ms
- **Paso instantáneo a 3D de las detecciones**: entre 0.3 y 0.4 ms
- **Fusión de todas las estimaciones (visuales y encoders)**: entre 3.8 y 4.4 ms

Por supuesto estos tiempos pueden variar dependiendo del equipo donde lo ejecutemos y la carga presente en el mismo en ese momento. Sin embargo, estos datos nos sirven para hacernos una idea global del coste en tiempo en un ordenador portátil convencional y de cuáles son las partes de más peso en el algoritmo.

Como se puede observar, la parte encargada de la detección de balizas en las imágenes es, con creces, la parte más costosa del algoritmo. Esto resulta bastante lógico puesto que el tratamiento y procesamiento de imágenes siempre suele ser una parte bastante pesada en coste computacional debido a la gran cantidad de datos que se manejan. Sin embargo,

detectar más o menos balizas en las imágenes apenas influye en el tiempo de procesamiento puesto que la mayor parte del algoritmo se encarga de la búsqueda de balizas y no de de procesar las detecciones encontradas.

La parte del algoritmo que hace los cálculos para obtener estimaciones absolutas en 3D de cada una de las detecciones es una parte poco costosa, puesto que la mayoría de las operaciones son configuraciones y multiplicaciones de matrices, las cuáles están debidamente optimizadas por el uso de la librería matemática *Eigen*. Esta parte se ve claramente influenciada por el número de detecciones encontradas, ya que esto hará aumentar en gran medida el número de operaciones. Por cada baliza detectada, el tiempo de procesamiento en esta sección aumenta en unos 0.2 - 0.3 milisegundos.

La última parte del algoritmo es la encargada de realizar la fusión entre todas las estimaciones obtenidas. Aunque no es una sección muy costosa comparada con la detección de balizas en 2D, los cálculos trigonométricos que se realizan hacen que sea la segunda parte más costosa. La parte de cálculo de la estimación de los encoders y su fusión es fija, independientemente del número de estimaciones visuales a procesar. Por cada baliza detectada y, por tanto, por cada estimación visual a fusionar, el coste computacional de esta parte del algoritmo aumenta entre 1 y 1.2 milisegundos.

Como resultado de este análisis, se puede determinar que el algoritmo de autolocalización desarrollado tiene un coste computacional aceptable independientemente del número de balizas detectadas y procesadas, lo que lo hace idóneo para aplicaciones robóticas en tiempo real.

Capítulo 6

Conclusiones

En los capítulos anteriores se han mostrados las herramientas utilizadas, la solución desarrollada para resolver el problema planteado y experimentos que la validan. En el presente capítulo se resume en qué medida se han cumplido los objetivos y satisfecho los requisitos planteados al principio de este trabajo. También se repasarán los aportes y los conocimientos que se han adquirido a lo largo de la realización de este proyecto fin de carrera. Como cierre de la memoria, se indican posibles trabajos futuros partiendo de la base aquí desarrollada.

6.1. Conclusiones

En este proyecto de fin de carrera se ha desarrollado el componente Visualloc para conseguir estimaciones precisas y robustas de autolocalización en todo momento a través de balizas visuales detectadas con las cámaras a bordo del robot y enriquecidas con la información odométrica para poder dar estimaciones incluso cuando no hay ninguna baliza en el entorno (por oclusiones temporales o baja densidad de balizado).

Primero, pasamos a detallar el grado de realización de cada uno de los objetivos propuestos en el capítulo 2 de esta memoria:

1. Durante la transcurso de este proyecto de fin de carrera he tenido la oportunidad de aprender numerosas cosas y de reforzar otras de las que sólo tenía unos conocimientos

básicos.

Respecto a sistemas operativos, puesto que todo el desarrollo se ha realizado sobre sistemas basados en Unix, he podido aprender bastante sobre su manejo, scripting e instalación y tratamiento de librerías. Además, he profundizado en el uso de bastantes componentes de la herramienta JdeRobot que me ha proporcionado una base para poder introducirme en el desarrollo de aplicaciones para robótica. Como parte de la depuración del componente, he aprendido el manejo y configuración del simulador Gazebo, ampliamente utilizado para la emulación de entornos y robots.

Respecto a la parte de visión artificial, he adquirido conocimientos en el uso de librerías como OpenCV, AprilTags, Aruco y Eigen que facilitan enormemente el desarrollo de aplicaciones en este ámbito. Además, he tenido que repasar y ampliar mis conocimientos en geometría para poder realizar y comprender todas las transformaciones entre los numerosos sistemas de referencia presentes en nuestro problema.

2. Para la consecución de este objetivo se ha desarrollado un componente, llamado Visualloc, encargado de calcular estimaciones de posición y orientación de nuestro robot en el mundo. Como se ha indicado en el capítulo 4, el componente se ha dividido en tres partes:

   - La primera se encarga de detectar balizas en las imágenes obtenidas por las cámaras, usando finalmente la librería AprilTags (sección 4.2).
   - La segunda parte realiza el cálculo de posición y orientación en el mundo dado por cada una de las detecciones, usando Aruco y numerosos cálculos geométricos. (sección 4.3).
   - Partiendo de estas estimaciones instantáneas obtenidas, se realiza una fusión entre ellas a través de medias ponderadas, cuyo peso en el cálculo dependerá de la distancia entre la baliza y la cámara que la detectó. Posteriormente se fusiona también esta estimación con el incremento recibido de la odometría del robot, lo que proporciona una estimación aceptable en todo momento, incluso cuando no se estén detectando balizas en las imágenes (sección 4.4).

   Además de esto, se realizan filtros de espúreos para evitar estimaciones erróneas y falsos positivos (sección 4.4.2) y se ha creado una interfaz de usuario que consta de un mundo tridimensional en OpenGL para facilitar la visualización de los resultados y la depuración del algoritmo (sección 4.5).

3. Para validar experimentalmente la robustez y precisión de nuestro algoritmo, se han realizado numerosas pruebas tanto en la nave industrial real como en el simulador Gazebo, para lo que se ha diseñado un escenario virtual con todos los elementos necesarios (capítulo 5). Puesto que este componente desarrollado forma parte de sistema robótico real para una empresa en el ámbito de la robótica industrial, las pruebas realizadas en ambos escenarios demuestran que es un sistema lo suficientemente fiable, robusto y preciso.

A continuación se revisan los requisitos planteados en el capítulo 2 y se indica en qué medida se han cumplido cada uno de ellos:

- El componente que se ha desarrollado en C++ y ha seguido la arquitectura y herramientas propuestas por JdeRobot.

- Todo el software desarrollado se ha comprobado que funciona sobre distintas arquitecturas y sistemas, puesto que se ha ejecutado en distintos ordenadores y se ha conseguido que funcione tanto sobre el robot real como en el simulador, como se puede observar en el apartado de pruebas (capítulo 5).

- Tras la realización de numerosas pruebas se concluye que el algoritmo de autolocalización es capaz de dar estimaciones con un márgen de error de 10 cm, por lo que es bastante preciso.

- Como se ha indicado en el capítulo de pruebas, el algoritmo de cálculo de estimaciones de autolocalización es lo suficientemente rápido para poder trabajar en tiempo real. De hecho, el componente funciona efectívamente sobre un robot real en una nave industrial.

- En el la sección 4.4 se describen todos los cálculos geométricos que posibilitan la integración entre observaciones obtenidas por distintas cámaras. Además, la posición de las cámaras sobre el robot se encuentra parametrizada a través del fichero de configuración de nuestro componente, lo que facilita su ejecución en distintas configuraciones del robot.

- El componente Visualloc se ha integrado con éxito dentro de la arquitectura del robot industrial, por lo que cumple con las interfaces concretas previamente establecidas.

6.2. Trabajos futuros

Después de haber logrado un componente robusto de autolocalización con balizas visuales, algunas de las mejoras que se podrían realizar están relacionadas con la versatilidad del algoritmo. Una posibidad sería adaptar el funcionamiento del algoritmo para que adaptara distintas familias de etiquetas de forma simultánea, o incluso etiquetas de distintas librerías a la vez (AprilTags y Aruco, por ejemplo). De esta forma conseguiríamos un mayor número de etiquetas disponibles y mayor adaptabilidad del algoritmo. En esta línea de mejora, también podría incluirse la incorporación de un número parametrizable de cámaras sobre el robot, de forma que el componente pueda funcionar sobre sistemas robóticos mucho más heterogéneos.

Además de esto, algunas líneas de investigación que se podrían seguir son las siguientes:

- Realizar un estudio de la precisión de la detección y paso a 3D instantáneo de balizas situadas en distintos ángulos y a distintas distancias de la cámara. Esto podría proporcionar mejoras en el algoritmo al adaptar de forma más adecuada los pesos de cada una de las estimaciones en la parte de fusión.
- Investigar distintas formas de conseguir detectar balizas que se encuentren más alejadas de la posición del robot, por ejemplo aumentando el tamaño de las balizas o utilizando cámaras de mayor resolución. De esta forma conseguiríamos tener mayor número de balizas detectadas incluso en espacios grandes, donde éstas son colocadas en paredes distantes.
- Introducir en el algoritmo la posibilidad de trabajar con balizas acopladas, por ejemplo, colocar las balizas de 4 en 4 formando un cuadrado, con la intención de aumentar la precisión del algorimo de autolocalización.

Bibliografía

[Martín Organista, 2014] Daniel Martín Organista. Odometría visual con sensores RGBD. *Proyecto de fin de carrera, Universidad Rey Juan Carlos*, 2014.

[López Fernández, 2005] José Alberto López Fernández. Localización de un robot con visión local. *Proyecto de fin de carrera, Universidad Rey Juan Carlos*, 2005.

[Dellaert, 1999] Frank Dellaert. Robotics and Automation. *Universidad Carnegie Mellon, Pittsburgh (Pensilvania)*, 1999.

[Davison, 2003] Andrew J. Davison. Real-Time Simultaneous Localisation and Mapping with a Single Camera. *Robotics Research Group, Dept. of Engineering Science, University of Oxford*, 2003.

[Klein, 2009] Georg Klein. Parallel Tracking and Mapping on a Camera Phone. *Active Vision Laboratory Department of Engineering Science University of Oxford*, 2009.

[Hernández, 2014] Alejandro Hernández Cordero. Autolocalización visual aplicada a la Realidad Aumentada. *Proyecto de fin de máster, Universidad Rey Juan Carlos*, 2014.

[Morales, 2014] Luis Roberto Morales Iglesias. Algoritmo de Autolocalización Evolutiva Multimodal para Robots Autónomos. *Proyecto de fin de carrera, Universidad Rey Juan Carlos*, 2014.

[López Ramos, 2010] Luis Miguel López Ramos. Autolocalización en tiempo real mediante seguimiento visual monocular. *Proyecto de fin de carrera, Universidad Rey Juan Carlos*, 2010.

[Plaza, 2003] José María Cañas Plaza. Jerarquía dinámica de esquemas para la generación de comportamiento autónomo. *Tesis doctoral, Universidad Politécnica de Madrid*, 2003.