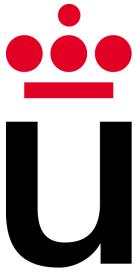


Universidad
Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Grado en Ingeniería en
Tecnologías de la Telecomunicación

Trabajo Fin de Grado

Mapas 3D de Parches Planos en Entornos Reales utilizando Sensores RGB-D

Autor: Samuel Martín Martínez

Tutor: Prof. Dr. José María Cañas Plaza

Curso académico 2015/2016

Stay hungry. Stay foolish.

Agradecimientos

En primer lugar, quiero agradecer a mis padres que siempre me hayan guiado en una buena dirección, motivado a estudiar desde que era pequeño y animado a conseguir mis objetivos. Sin duda, no habría llegado aquí si no fuera por ellos. A mi hermana Amanda por hacerme mejor persona. También a mis abuelos, tíos y primos por su apoyo. En especial a mi tía Pili, que siempre se preocupaba por mis inquietudes y por cómo me iba en la universidad y no ha podido conocer la conclusión de esta etapa.

En segundo lugar me gustaría agradecer a mi tutor Jośe María su dedicación, paciencia, apoyo y motivación a lo largo de este trabajo. Recuerdo un sinfín de reuniones con él a las que llegaba con la cabeza hecha un lío, pero siempre era capaz de despejar todas las nubes de dudas y de motivarme para continuar con muchas ganas.

También quiero dar las gracias a los compañeros del laboratorio de robótica de la universidad. A Juan Navarro, por tenderme una mano incluso antes de que lo pidiera. A Alberto López-Cerón, por encontrar huecos en su agenda para resolverme dudas. Y a Francisco Pérez por ayudarme a través de la lista de correo.

Tampoco puedo olvidarme de dar las gracias a todos los profesores que he tenido a lo largo de la carrera.

Por supuesto, quiero agradecer a mis amigos de siempre y a los que he conocido durante esta etapa, tanto en clase como en asociaciones, por hacer de ella algo maravilloso. Han sido una parte fundamental de mi día a día y ha sido, y será, un placer compartir el tiempo con ellos. También quiero agradecerle a mi novia, Esther, su cariño, apoyo, comprensión y paciencia aguantando innumerables conversaciones sobre los problemas que se me presentaban en este trabajo.

Por último, pero no menos importante, me gustaría dar las gracias a mis antiguos compañeros de trabajo, con los que empecé las prácticas y me quedé a trabajar durante más de un año y medio. Siempre que lo necesité encontraban el tiempo que hiciese falta para echarme una mano y enseñarme lo mejor posible.

¡Muchísimas gracias a todos!

Resumen

Los rápidos avances en la industria de los videojuegos han generado dispositivos capaces de mejorar la experiencia del usuario, como el sensor RGB-D Kinect para consola Xbox. Este dispositivo incorpora, además de una cámara a color, una cámara de profundidad, con lo que el usuario puede interaccionar con los juegos mediante movimientos y gestos sin necesidad de mandos. La llegada de este tipo de dispositivos bajó el precio de accesibilidad a sensores de profundidad en un orden de magnitud, por lo que, desde su lanzamiento, han sido utilizados en multitud de proyectos de robótica y visión artificial.

El objetivo fundamental de este Trabajo Fin de Grado ha sido diseñar un sistema que permita realizar mapas tridimensionales de entornos reales a partir de sensores RGB-D. Los mapas se generan con el sensor en movimiento, en tiempo real, de forma fluida y mediante parches planos grandes. Un sensor que genera mapas tridimensionales del entorno mientras está en movimiento, además de un mecanismo para conocer la posición de los objetos en el mundo, como son las imágenes de profundidad en este caso, precisa de un modo para autolocalizarse y saber su posición en el mapa. En este trabajo para ello se han utilizado balizas visuales cuya localización se conoce en la escena. El sistema diseñado ha sido validado experimentalmente tanto en entornos simulados como reales.

Este trabajo se apoya en un Proyecto Fin de Carrera previo que generaba mapas de parches planos en simulación y de modo discontinuo en el tiempo. Se ha mejorado el software para que funcione correctamente con sensores reales y de modo fluido en el tiempo. Para ello ha habido que corregir errores de sincronismo e incorporar un sistema de autolocalización 3D visual. Para el desarrollo del sistema se han utilizado los lenguajes de programación Python y C++ sobre la plataforma de software libre JdeRobot, así como otras librerías libres, en el sistema operativo Ubuntu 14.04 de 64 bits.

Índice general

Índice de figuras

Capítulo 1

Introducción

Cuando entramos en una habitación y echamos un vistazo alrededor, reconocemos inmediatamente el suelo, las paredes, el techo, las mesas y el resto de objetos que hay en ella. Sin embargo, cuando un robot explora una habitación con una cámara, no ve otra cosa que una colección de líneas y curvas que convierte en píxeles. Se necesita una gran cantidad de tiempo de computación para que el robot finalmente pueda reconocer los límites del entorno, u objetos, y ser capaz de navegar por ese espacio sin colisionar.

Gracias a la llegada de los sensores RGB-D al mercado, además de tener su imagen a color, podemos tener información sobre la distancia a la que está todo de un modo sencillo y accesible. En este Trabajo Fin de Grado abordaremos el problema de la construcción virtual en 3D de mapas reales mediante parches planos grandes utilizando un sensor RGB-D en movimiento, dando por conocida la localización de ciertas balizas visuales que situaremos en la escena. A partir de estas balizas, seremos capaces de determinar la posición y orientación del sensor y, junto con la información de la cámara de profundidad, construir mapas 3D del entorno.

En este capítulo se propone dar una visión general del contexto de este Trabajo Fin de Grado que sirva de base para el resto de capítulos. Primero, se hablará sobre la visión por computador y sensores RGB-D y después, se irá profundizando más, pasando a la construcción de mapas 3D y la autolocalización de un sensor en su entorno explorando diversas técnicas. También se hablará sobre los proyectos realizados sobre estos temas en el laboratorio de robótica de la URJC y que son precursores directos de este trabajo.

1.1. Visión por computador

La visión por computador, o visión artificial, es un campo de la inteligencia artificial, figura 1.1, que tiene por objetivo modelar matemáticamente los procesos de percepción visual en los seres vivos y generar programas que permitan simular estas capacidades visuales en un computador. Pretende capturar la información visual del entorno físico para extraer características relevantes siguiendo procedimientos automáticos.

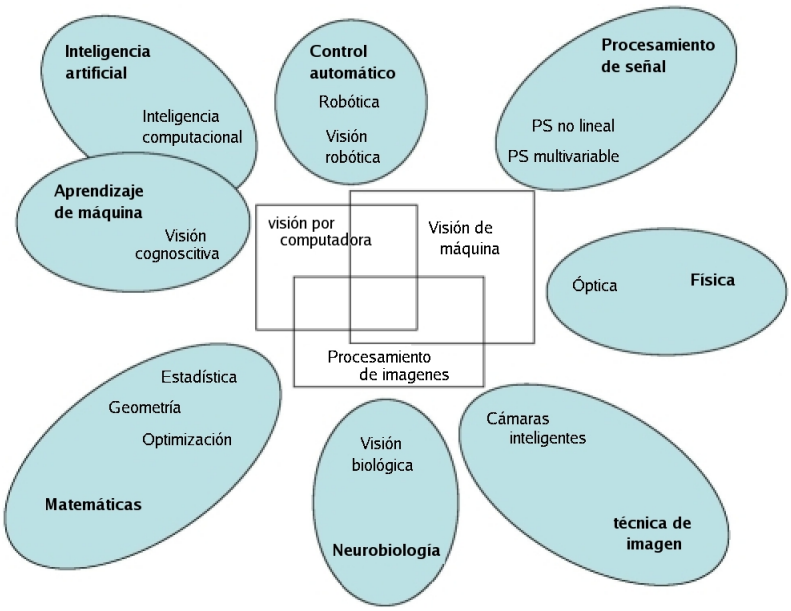


Figura 1.1: Relación de la visión por computador y otras áreas relacionadas.

El origen de este campo se puede situar en 1960 en la tesis doctoral de Larry Roberts, en el Massachusetts Institute of Technology (MIT). En ella discute las posibilidades de extraer información geométrica en 3D a partir de imágenes en dos dimensiones de bloques. Muchos investigadores de inteligencia artificial del MIT y de otras partes del mundo siguieron su trabajo y empezaron a estudiar visión por computador en el contexto de un mundo de bloques. Más tarde, los investigadores se dieron cuenta de que para abordar imágenes más complejas del mundo real, se necesitaba hacer una serie de procesamientos previos a la imagen para luego hacer la extracción de la información, como detección de bordes y segmentación. Una gran referencia sobre esto fue el trabajo de David Marr [1] en el MIT en 1978.

Típicamente, un sistema de visión por computador consta de las siguientes etapas,

como se puede ver en la figura 1.2:

1. Adquisición de imagen. El objetivo de esta etapa es realzar, mediante técnicas como la iluminación, óptica y tipo de cámara, las características visuales de los objetos tales como las formas, texturas y colores.

2. Preprocesamiento. En esta etapa se mejora la calidad informativa de la imagen adquirida. Para ello se aplican técnicas que reducen el ruido, mejoran el contraste, realzan características de la imagen y atenúan imperfecciones de la adquisición debido al sistema de captación de imagen.

3. Segmentación. En esta fase se segmenta la imagen en regiones de interés para la aplicación en la que nos encontremos, y se descarta el resto. Algunas de las técnicas que se utilizan para segmentar comprenden umbralizaciones, discontinuidades y crecimiento de regiones.

4. Extracción de características o medición. Una vez la imagen ha sido segmentada, se extraen sus características, tales como área, perímetro, excentricidad o momento de inercia.

5. Clasificación e interpretación. Llegados a este punto, hemos pasado de una imagen a una serie de características. En esta fase se diseñarán clasificadores para interpretar las características y dar a cada región segmentada una etiqueta de alto nivel. Existe una amplia gama de técnicas de clasificación como redes neuronales, sistemas expertos, lógica borrosa y clasificadores estadísticos.

El incremento exponencial en la potencia de los procesadores y los avances en los distintos tipos de cámaras digitales han llevado a una cantidad enorme de aplicaciones en la actualidad:

- Industria: Las tareas que desempeñan las máquinas utilizando visión artificial en los procesos de manufacturación son extremadamente importantes a día de hoy. En primer lugar, permite llevar a cabo un control de procesos en los que las máquinas son capaces de reconocer piezas para su separación o para el montaje del producto y eliminar las defectuosas. En segundo lugar, permite un control de calidad mucho más riguroso que el que podría conseguirse con un operario en sistemas críticos,

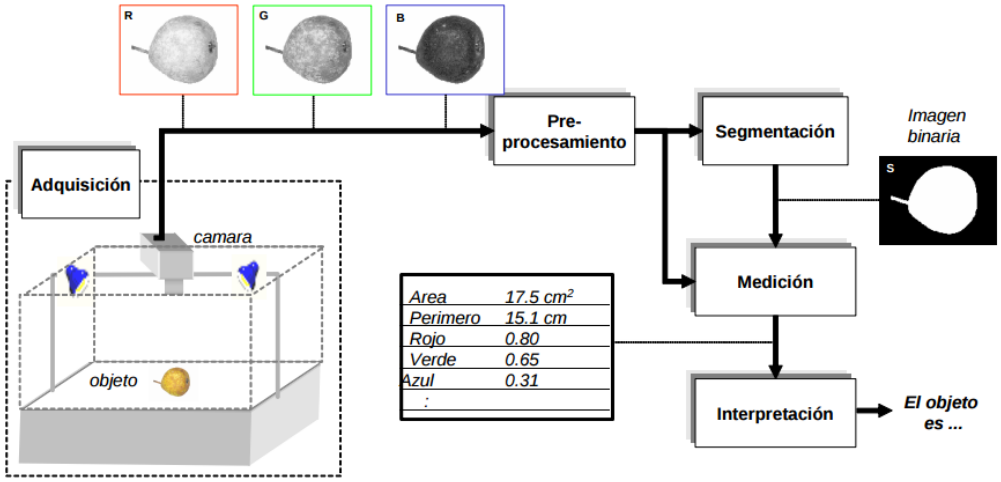
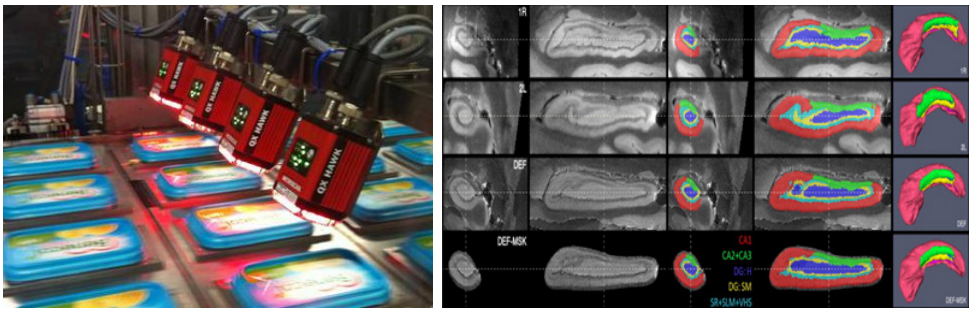


Figura 1.2: Etapas de un sistema de visión por computador.

como la fabricación de partes de un automóvil o de un avión. En la figura 1.3(a) podemos ver un ejemplo de control de calidad.

- Imagen médica: En este área se utilizan diversas técnicas de procesado para conseguir información automáticamente sobre el estado de un paciente para realizar un diagnóstico acertado. Por lo general, se utilizan imágenes microscópicas, tomadas con rayos X, ultrasonidos, resonancia magnética, como puede verse en la figura 1.3(b), y tomográficas. En este área se incluyen también las técnicas que mejoran imágenes para ser interpretadas por humanos.



(a) Control de calidad. (b) Procesado automático en resonancia.

Figura 1.3: Aplicaciones industriales y médicas.

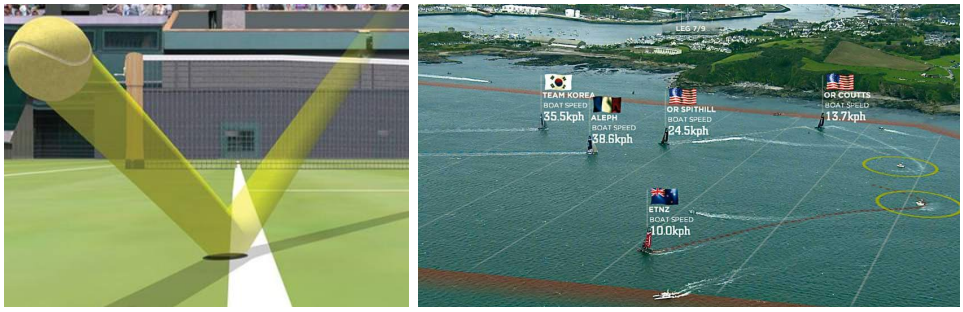
- Seguridad vial: La detección de obstáculos inesperados como peatones así como técnicas para monitorizar el estado del conductor son algunas de las aplicaciones

que ya están evitando muchos accidentes.

- Biometría: El reconocimiento facial y dactilar, entre otros, son fundamentales para tener sistemas de identificación robustos, o para evitar el uso de llaves y tarjetas, y también en aplicaciones forenses.

- Videovigilancia: Monitorización de intrusos, control del estado de las carreteras, prevenir ahogamientos en piscinas o sistemas de detección de caídas de personas mayores que viven solas son algunos de sus usos.

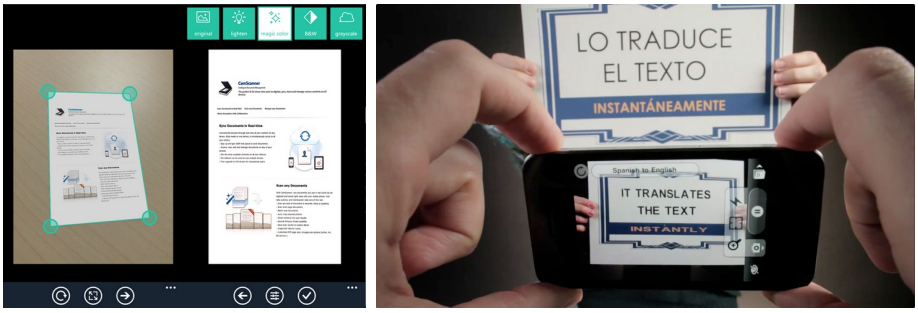
- Deportes: La técnica de ojo de halcón, como se puede ver en la figura 1.4(a), se utiliza en deportes, como el tenis o el fútbol, para determinar si la pelota ha botado fuera del campo o ha entrado en portería. Otra aplicación es incluir mediante realidad aumentada información a las imágenes de televisión, como por ejemplo publicidad, las distancias en un salto de longitud, información sobre la jugada realizada, como se hace en fútbol americano, o información visual en una regata como se puede ver en la figura 1.4(b).



(a) Ojo de halcón. (b) Información en una carrera.

Figura 1.4: Aplicaciones en deportes.

- Aplicaciones móviles: La aplicación CamScanner, figura 1.5(a), permite utilizar una cámara convencional como si de un escáner se tratara, detectando los bordes del documento, cambiando sus proporciones si estas no eran rectangulares y mejorando el contraste de forma automática. Otro ejemplo de aplicación móvil es WordLens que, utilizando realidad aumentada, es capaz de traducir el texto que se ve en la misma imagen, tal y como podemos ver en la figura 1.5(b).



(a) Aplicación CamScanner. (b) Aplicación WordLens.

Figura 1.5: Aplicaciones en smartphones.

- Astronomía: La visión artificial permite estudiar el cosmos en longitudes de onda que el ser humano no puede ver, como la infrarroja, la ultravioleta y rayos x. La misión Kepler de la NASA, en la que se buscan planetas en otros sistemas solares, ya ha encontrado más de 1200 exoplanetas gracias a la medición automática del brillo de más de 150000 estrellas.

- Robótica: Aplicando técnicas de localización, un robot puede ser capaz de estimar la posición y la orientación en un entorno para poder ser capaz de navegar satisfactoriamente por él, así como reconocer y manipular objetos. Un ejemplo es la aspiradora Roomba 980 de la compañía iRobot, que, mediante una cámara, es capaz de navegar por un apartamento así como ir a la estación de carga si se queda sin batería y volver al punto donde dejó la labor para continuar.

- Mapeo topográfico: A partir de imágenes aéreas se pueden reconstruir modelos 3D de forma automática, como por ejemplo se hace en Bing Maps. Por otro lado, StreetView, la aplicación de Google Maps que nos permite navegar por las calles de medio mundo, utiliza un algoritmo para emborronar automáticamente las matrículas y las caras de las personas que aparecen en esas fotos en 360 grados.

- Videojuegos: El sector de la visión por computador ha avanzado bastante gracias al de los videojuegos. Ejemplos claros son la cámara EyeToy, desarrollada por Sony para la consola PlayStation, el dispositivo Kinect, desarrollado por Microsoft para la consola Xbox, o el juego Invizimals, que ofrece realidad aumentada en la PSP de Sony mediante una cámara a color. Mediante estos sensores el usuario puede interaccionar con el juego por medio de gestos y movimientos.

1.1.1. Sensores RGB-D

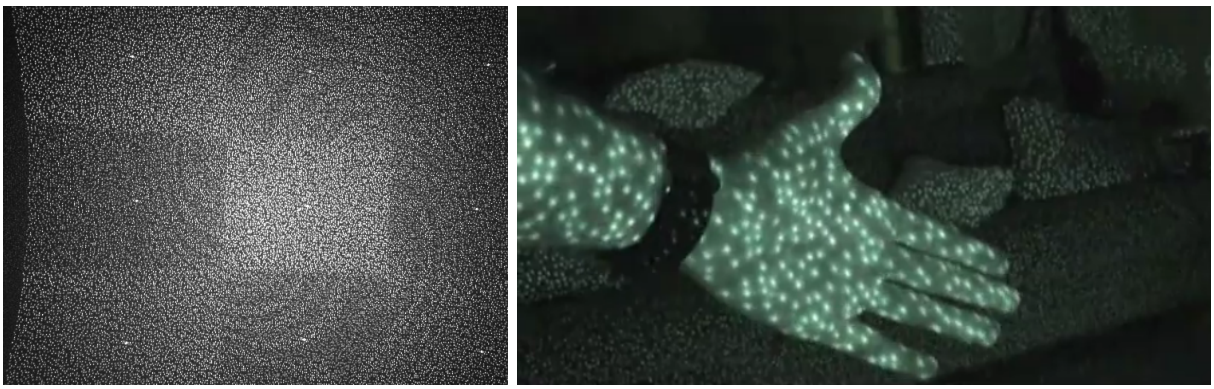
Los avances en la tecnología han revolucionado la visión artificial gracias a la llegada de los sensores de profundidad, que permiten recrear modelos 3D con un coste computacional y un tiempo menores que si se hiciera con cámaras a color convencionales. Un sensor RGB-D es un tipo de sensor de profundidad con una cámara a color convencional y dispositivos capaces de obtener la distancia a cada píxel de la imagen utilizando luz infrarroja. Otros tipos de sensores de profundidad usan técnicas de tiempo de vuelo (TOF), en las que se mide el tiempo que tarda la luz en retornar del objeto, como por ejemplo utiliza el sensor Kinect 2.0 para Xbox One, o visión estereoscópica, en el que a partir de dos cámaras se triangula la posición de los objetos.

La salida del sensor Kinect, figura 1.6(a), para la consola Xbox 360 en noviembre 2010 al precio de 150 euros, aumentó increíblemente la accesibilidad a los sensores de profundidad ya que redujo el coste en un orden de magnitud. Más tarde, en mayo de 2012, Asus lanzó al mercado sus sensores Asus Xtion PRO y Asus Xtion PRO LIVE, figura 1.6(b), por debajo de los 200 euros, aumentando la oferta y competitividad en el sector. El primero solo cuenta con una cámara de profundidad mientras que el segundo también cuenta con una cámara a color como el sensor Kinect.

(a) Kinect. (b) Asus Xtion PRO LIVE.

Figura 1.6: Sensores RGB-D.

Estos sensores son capaces de medir distancias gracias a un receptor y un emisor de luz infrarroja. En primer lugar, se emite un patrón de puntos infrarrojos como puede verse en la figura 1.7. En segundo lugar, el patrón proyectado es capturado por la cámara infrarroja y comparado con patrones de referencia que están guardados en el dispositivo cuya distancia se conoce. Por último, el dispositivo estima la profundidad de cada píxel en base a qué patrón guardado encaja mejor con el proyectado.



(a) Patrón emitido. (b) Puntos más de cerca.

Figura 1.7: Patrón de luz emitido por Kinect.

1.2. Mapeado y localización desde visión

Dos campos importantes en la visión por computador son el mapeado y la localización desde visión. El mapeado desde visión consiste en crear una representación del entorno a partir únicamente de información visual. La autolocalización desde visión consiste en estimar la posición y orientación, típicamente en tres dimensiones, del sensor a partir exclusivamente de las imágenes que recibe.

Estos dos campos han suscitado el interés de muchos investigadores en los últimos años y han sido abordados por dos comunidades científicas de forma distinta. Por un lado, el nombre que recibió por la comunidad de visión es *Structure from Motion* (SfM), en el que la información se procesa típicamente por lotes. Por otro, la comunidad de robótica denominó este problema como *Simultaneous Localization and Mapping* (SLAM), que intenta proporcionar una solución en tiempo real. En esta sección expondremos estas dos líneas de investigación y además veremos técnicas de localización mediante marcadores y técnicas de mapeado a través de sensores RGB-D ya que en este Trabajo Fin de Grado son las que se han utilizado. Por último, veremos algunas de las aplicaciones más conocidas.

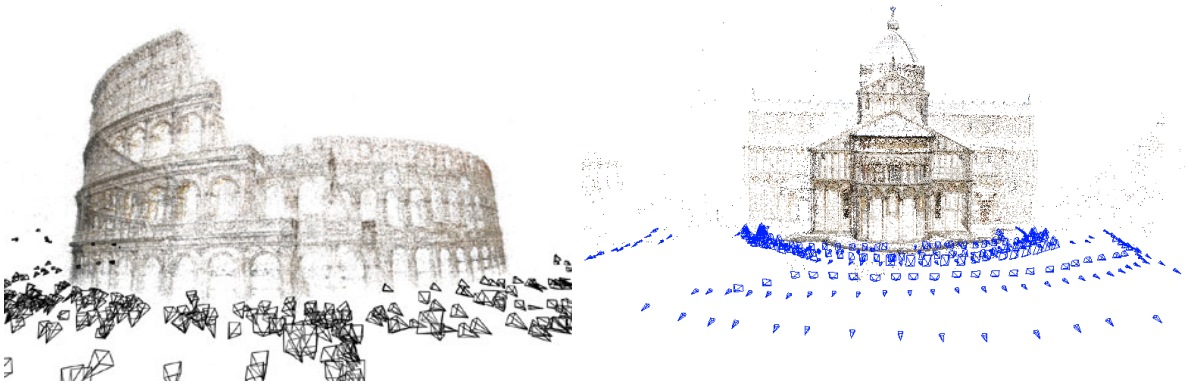
1.2.1. Structure from Motion

Esta línea de investigación se dedica a la reconstrucción de modelos 3D a partir únicamente de un conjunto de imágenes en 2D del objeto utilizando triangulación, teniendo la cámara que tomó las imágenes correctamente calibrada. Para reconstruir el

modelo, se ha de estimar la posición y orientación de la cámara con respecto al objeto en todo momento.

Este tipo de sistema es fuera de línea, analiza la secuencia de imágenes después de haber sido realizadas. Los algoritmos que implementan son muy costosos computacionalmente y no pueden ser empleados en tiempo real. Pueden realizar optimizaciones sobre las trayectorias estimadas y reajustar reprocesando datos ya analizados yendo hacia atrás o hacia delante en la escena. Uno de estos procedimientos es el ajuste de haces.

La implementación de este método se encuentra en un estado de madurez considerable y muchas aplicaciones comerciales ya utilizan estos algoritmos. Una buena muestra de esto es la aplicación PhotoTurism [2], figura 1.8(a), de Microsoft que consigue reconstruir escenas 3D, a partir de fotos hechas en ella, por las que el usuario puede navegar. Otros trabajos en esta línea, como el propuesto por Marc Pollefeys, Reinhard Koch, y Luc J. Van Gool [3], estudian resolver este problema teniendo en cuenta cualquier tipo de restricciones dadas por los parámetros intrínsecos de la cámara, de modo que la reconstrucción se podría hacer con una cámara sin calibrar. Un ejemplo de esto es la aplicación 3DF Samantha desarrollada por la empresa 3Dflow [1], figura 1.8(b).



(a) Microsoft PhotoTourism. (b) 3DF Samantha de 3Dflow.

Figura 1.8: Aplicaciones que implementan técnicas de *Structure from Motion*.

1.2.2. Visual SLAM

El principal problema para cualquier robot autónomo cuya tarea incluya la navegación tiene dos características principales: el mapeado del entorno y la localización del robot.

[1] `http://www.3dflow.net/`

Para que un robot pueda estimar su localización a partir de los sensores que incorpora necesita un mapa en el que pueda decidir dónde está. Pero para hacer ese mapa el robot necesita conocer su posición en el entorno. Este problema, conocido como *Simultaneous Location and Mapping*, localización y mapeado simultáneos, lleva décadas en el corazón de muchas investigaciones en robótica.

Visual SLAM aborda este problema utilizando solo cámaras como sensores principales en lugar de sistemas de posicionamiento de exteriores como GPS, que fracasan en interiores, o sistemas de posicionamiento por láser, que son mucho más caros. Su objetivo es estimar en tiempo real la posición y orientación de una cámara en movimiento a partir de las características del entorno a la vez que realiza un mapeado del mismo. Hay métodos de Visual SLAM con una cámara y con dos cámaras en estéreo.

Visual MonoSLAM fue introducido por Andrew Davison [4] en el año 2007. El algoritmo que implementa utiliza un filtro extendido de Kalman para estimar la posición y orientación de la cámara, además de la posición de una serie de puntos del espacio 3D. Para determinar la posición inicial de la cámara, hace falta dotar al filtro de Kalman de información a priori con la posición en 3D de al menos 3 puntos. De ahí en adelante, el algoritmo es capaz de situar la localización de la cámara en el entorno y de generar nuevos puntos para crear el mapa. El principal problema de este algoritmo es que su tiempo de ejecución aumenta con el número de puntos ya que en cada iteración se calcula tanto la localización de la cámara como el mapa del entorno. Debido a esto, si hay muchos puntos el sistema deja de funcionar en tiempo real.

También cabe destacar la técnica *Parallel Tracking and Mapping*, mapeado y seguimiento paralelo, desarrollada por Georg Klein en 2007 [5] como alternativa a Visual MonoSLAM. En esta técnica, el seguimiento de la localización de la cámara y el mapeado se dividen en dos tareas completamente distintas, procesadas en hilos diferentes. Solo la estimación de la localización funcionaría en tiempo real mientras que el mapeado funcionaría de modo asíncrono, ya que parte de la idea es que únicamente es necesario que la localización funcione en tiempo real.

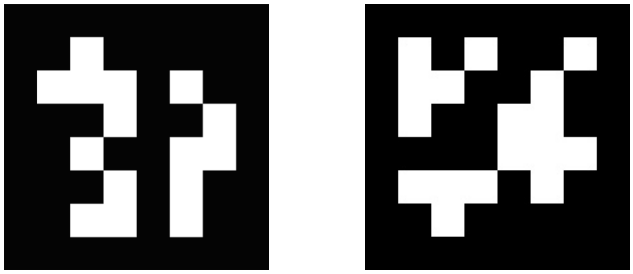
1.2.3. Localización visual con marcadores

Trabajar con imágenes sin restricciones, como hemos visto en la técnica anterior, es un proceso muy costoso computacionalmente debido a la complejidad involucrada de los

algoritmos. La localización visual con marcadores se basa en conocer ciertas partes del entorno en las que estarán posicionados de modo que la cámara, conociendo la localización del marcador, será capaz de estimar su propia posición y orientación con respecto al mundo.

En interiores hay dos tipos de marcadores: activos y pasivos. Por un lado, los activos se pueden detectar con sensores especiales, como por ejemplo un marcador infrarrojo y sensor infrarrojo. Las soluciones basadas en este tipo de marcadores no demandan apenas tiempo de procesado debido a que son fáciles de encontrar. Tienen como desventaja que los marcadores activos son más caros y requieren de fuente de alimentación. Por otro lado, los pasivos son más difíciles de encontrar y requieren más tiempo de computación, pero son más baratos y fáciles de instalar.

Las características principales que tiene que tener un marcador pasivo son el contraste y la forma. El contraste ha de ser lo más alto posible y la forma debe ser distinta a las formas comunes en el entorno. Los marcadores que utilizan las librerías AprilTags y ArUco, figura 1.9, son ejemplos de esta categoría.



(a) AprilTags. (b) ArUco.

Figura 1.9: Ejemplos de marcadores.

1.2.4. Mapeado con sensores RGB-D

Como vimos en la sección 1.1.1, la llegada de los sensores RGB-D al mercado supuso una bajada del coste del acceso a un sensor de profundidad, desplegando un abanico de posibilidades en cuanto a líneas de investigación y proyectos.

En 2012, Felix Endres, Jürgen Hess, Nikolas Engelhard, Jürgen Sturm, Daniel Cremers y Wolfram Burgard presentan un trabajo evaluando el sistema RGB-D [6]. El resultado

de ese trabajo es la generación de un mapa 3D del entorno, figura 1.10, que puede ser utilizado para localización, navegación y planeamiento de ruta.

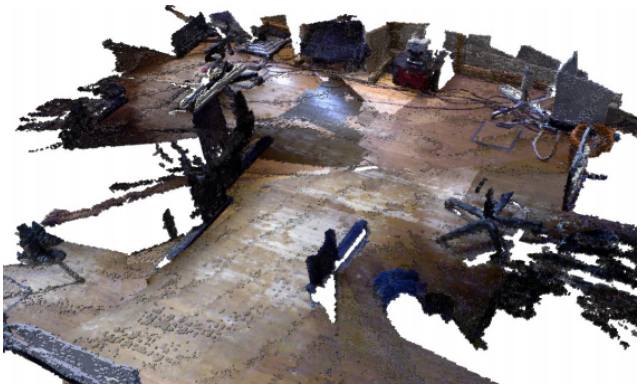


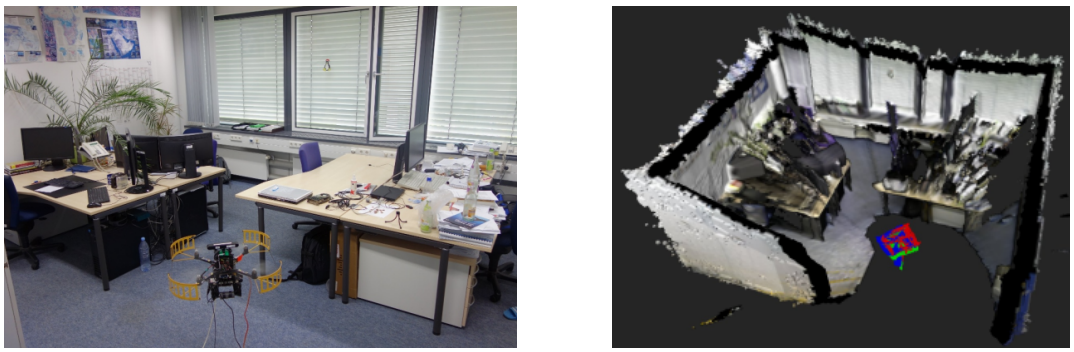
Figura 1.10: Reconstrucción usando RGB-D SLAM.

El trabajo sobre estimación robusta de odometría para cámaras RGB-D de Christian Kerl, Jürgen Sturm y Daniel Cremers [7], publicado en 2013, muestra una solución para el problema RGB-D SLAM. Su sistema funciona en tiempo real en una sola CPU, consumiendo pocos recursos y de forma más robusta que el resto de sistemas hasta la fecha. Este artículo fue finalista al premio como mejor artículo de visión por computador del ICRA (Conferencia Anual en Robótica y Automatización) en el año 2013.

En 2013, Jürgen Sturm, Erik Bylow, Christian Kerl, Fredrik Kahl y Daniel Cremers presentan un trabajo [8] en el que realizan mapeados de interiores utilizando un cuadricóptero autónomo que se mueve siguiendo una ruta prefijada y un sensor RGB-D. En la figura 1.11 se pueden ver imágenes del cuadricóptero en una oficina portando el sensor RGB-D y del modelo en 3D que construye de ese escenario.

En el Microsoft Kinect Challenge, que tuvo lugar en Chicago en 2014, se mostró el estado actual de robots autónomos mapeando y autolocalizándose en entornos reales con todos los muebles típicos que nos podemos encontrar en habitaciones e, incluso, con gente moviéndose. El equipo que ganó utilizó una técnica conocida como RTAB-Map (mapeado basado en apariencia en tiempo real), que es una variación de SLAM. En este vídeo [2], se puede ver al robot ganador moverse por el entorno a la vez que genera un mapa del mismo.

[2] `https://www.youtube.com/watch?v=_qiLAWp7AqQ`



(a) Cuadricóptero con sensor RGB-D. (b) Modelo 3D.

Figura 1.11: Mapas 3D de interiores con sensor RGB-D y cuadricóptero.

El Proyecto Tango [3] de Google es también un ejemplo de mapeado del entorno. Su objetivo es dotar a los teléfonos inteligentes con la capacidad de crear mapas 3D realistas y de navegación por interiores, ya que en estas situaciones los GPS fallan. Su estado actual es de dos dispositivos Andorid para desarrolladores, un smartphone y una tableta. En junio de este año, 2016, la empresa Lenovo anunció la salida al mercado del primer *smartphone* comercial que incorpora la tecnología del Proyecto Tango. Se espera que salga al mercado a lo largo del segundo semestre de 2016.

1.2.5. Aplicaciones

Gracias a los avances en los algoritmos de mapeado y localización en los últimos años, se ha desarrollado un gran número de aplicaciones y se han abierto líneas de investigación que nos traerán productos realmente increíbles, con un rango de aplicaciones que abarcará desde el sector de los negocios al de los videojuegos pasando por el cotidiano.

Una de las aplicaciones que se ha beneficiado de los algoritmos de localización es la realidad aumentada, que consiste en añadir elementos virtuales a una imagen real, como vimos en el ejemplo de los usos deportivos en las aplicaciones de la visión por computador.

Microsoft HoloLens [4], figura 1.12(a), es el primer ordenador, en forma de gafas, que funciona por medio de hologramas. Estos son creados combinando técnicas de realidad aumentada y de realidad virtual en lo que se conoce como realidad mixta. En esta realidad mixta, se puede interaccionar con los objetos posicionados mediante realidad aumentada

[3] `https://www.google.com/atap/project-tango/`

[4] `https://www.microsoft.com/microsoft-hololens/en-us`

como si fueran parte del entorno. Otras gafas que incorporan realidad aumentada son las Google Glass. El potencial de estas líneas de investigación es enorme y es fácil imaginarse, en un futuro cercano, unas lentes de contacto que nos proporcionen todas estas funcionalidades.

Occipital [5], es una empresa que utiliza un sensor de profundidad, llamado Structure y diseñado por ellos mismos, que se conecta a dispositivos móviles, como un iPad, para crear modelos 3D virtuales de objetos, figura 1.12(b), de espacios interiores y utilizar realidad aumentada con distintos fines.

La localización con marcadores simplifica el problema de la localización, como hemos visto, por lo que es un modo sencillo y robusto para utilizar realidad aumentada. En la figura 1.12(c) podemos ver un ejemplo.

(a) Microsoft HoloLens. (b) Occipital con el sensor Structure. (c) ArUco y realidad aumentada.

Figura 1.12: Aplicaciones de realidad aumentada.

La localización y mapeado desde visión se emplean también, junto con otras técnicas, en el sector de los coches autónomos, que en los últimos años ha suscitado muchísimo interés y que sin duda revolucionarán el mercado. Ejemplo de esto es la flota de coches autónomos de Google [6], como los Google Car 1.13(a), que a día de hoy ya han recorrido más de 2000000 de kilómetros en total. Cabe destacar también los robots exploradores de Marte enviados por la NASA, como el Rover Curiosity [7], figura 1.13(b), que hace uso de este tipo de técnicas para navegar por el entorno evitando obstáculos.

[5] `http://structure.io/`

[6] `https://www.google.com/selfdrivingcar/`

[7] `https://www.nasa.gov/mission_pages/msl/index.html`

(a) Google Car. (b) Rover Curiosity en Marte.

Figura 1.13: SLAM para autonomía.

1.3. Antecedentes

En el laboratorio de robótica de la URJC se han realizado algunos trabajos previos con sensores RGB-D, que son precedentes directos de este Trabajo Fin de Frado.

Primero, Juan José García Cantero desarrolla en su Proyecto Fin de Carrera [9], presentado en 2013, una herramienta calibradora de sensores RGB-D, motivado por la necesidad de calibrar estos dispositivos de captación tridimensional para su posterior uso en proyectos como este.

Segundo, Juan Navarro Bosgos resuelve en su Proyecto Fin de Carrera [10], en 2015, la construcción en tiempo real de mapas 3D en entornos simulados en Gazebo utilizando parches planos de gran tamaño. Para ello trabaja con la información de profundidad del sensor simulado y RANSAC como principal algoritmo para transformar la nube de puntos en planos. Este proyecto será uno de los pilares sobre los que se asentará este Trabajo Fin de Grado. Profundizaremos más en su funcionamiento en la sección 3.6.

Y en tercer lugar, Alberto López-Cerón Pinilla presenta en su Trabajo Fin de Máster [11], en 2015, un sistema de autolocalización visual basado en marcadores. La aplicación que desarrolla en él permite estimar la posición y orientación de una cámara a color a partir de marcadores visuales con la restricción de conocer la posición de esos marcadores en el mundo. Este trabajo será otro de los pilares sobre los que se asienta este trabajo. En la sección 3.5 trataremos con más detalle este sistema.

Este TFG combinará estos dos últimos trabajos para construir mapas de parches planos de forma fluida con un sensor RGB-D real, que se mueve por el entorno y se autolocaliza

mediante balizas visuales.

Esta memoria está organizada en cinco capítulos más. En el siguiente capítulo se expondrán los objetivos que persigue y la metodología y plan de trabajo que se han seguido para conseguirlos, mientras que en el tercero se presentan todas las tecnologías empleadas. En el capítulo cuarto se describirá el desarrollo del trabajo, profundizando en las tareas realizadas y los problemas encontrados para, en el quinto, validar los resultados con experimentos. En el sexto y último capítulo se resumirán las conclusiones de los resultados obtenidos, se valorará si se han cumplido los objetivos y se comentarán posibles líneas de trabajo futuro.

Capítulo 2

Objetivos

Tras poner en contexto este Trabajo Fin de Grado, en este capítulo se exponen los objetivos que pretende lograr, así como la metodología y planificación que se utilizarán para conseguirlos.

2.1. Descripción del problema

El objetivo principal de este proyecto es diseñar un sistema que pueda realizar representaciones tridimensionales de entornos reales mediante parches planos a partir de un sensor RGB-D en movimiento. Para ello, como hemos visto en la sección anterior, el sensor ha de conocer en todo momento su posición en el entorno.

Dada la complejidad del sistema, se partirá de un componente desarrollado por Juan Navarro en su Proyecto Fin de Carrera, que genera mapas de parches planos a partir de imágenes de profundidad y la posición del sensor. Solo lo realiza en entornos simulados, donde la posición del sensor es conocida ya que esa información la proporciona el simulador. En entornos reales, no obstante, hay que estimarla para poder realizar el mapeado. Por este motivo, utilizamos también el componente de autolocalización desarrollado por Alberto López en su Proyecto Fin de Máster, que estima la localización de una cámara a color a partir de marcadores visuales reales.

Dicho objetivo principal lo hemos dividido en los siguientes subobjetivos más concretos:

1. Desarrollo de un generador sintético de nubes de puntos que pueda enviar los datos generados a otras aplicaciones. Dado que se realizarán modificaciones en el mapeador de parches planos y hace falta estudiar el efecto de la modificación de parámetros en

su algoritmo para utilizarlo en entornos reales, es necesario tener información cruda sintética que podamos controlar.

2. Estudio del sincronismo de la aplicación generadora de parches planos para conseguir una extracción fluida en simulación mientras el sensor se mueve. Actualmente, existe un desfase variable entre la información de la posición y la imagen de profundidad del sensor RGB-D que desemboca en la extracción de falsos parches planos cuando la cámara los genera mientras se mueve.

3. Integración de los componentes en un mismo sistema para el mapeado de entornos reales y pruebas. Será necesario programar una capa de comunicaciones en la aplicación de autolocalización para que el mapeador pueda conectarse a ella, así como realizar ciertas operaciones geométricas para que la entienda. Además, habrá que añadir al mapeador nuevas funcionalidades para mejorar su funcionamiento en el entorno real así como optimizar los parámetros del algoritmo. Al final se realizarán pruebas para validar el correcto funcionamiento del sistema.

2.2. Requisitos

Aparte de cumplir los objetivos establecidos en la sección anterior, este trabajo deberá también satisfacer los siguientes requisitos:

- Se hará uso del entorno JdeRobot en su versión 5.3 y se desarrollarán componentes de forma modular programados en Python y C++.
- El sistema operativo que se utilizará será ubuntu 14.04 y los componentes podrán ejecutarse sobre las arquitecturas x86 y x64.
- El sistema final tendrá un rendimiento tal que permita la ejecución en tiempo real con movimiento suave pero continuo.
- El código desarrollado en este proyecto será liberado bajo licencia GPLv3.

2.3. Metodología

En la realización de un proyecto es muy importante establecer una metodología adecuada que asegure que se cumplen los objetivos estableciendo la planificación y los

pasos a seguir. En este trabajo se ha elegido como modelo de ciclo de vida el desarrollo en espiral, propuesto por Barry Boehm en 1986 [12].

Este modelo de ciclo de vida nos permite ir obteniendo prototipos funcionales a la vez que se realiza el desarrollo del producto de forma incremental. También está preparado para ir incluyendo los cambios en los requisitos, algo bastante común.

Consta de iteraciones que son llamadas ciclos. Cada ciclo se divide en cuatro fases bien diferenciadas, como se puede ver en la figura 2.1:

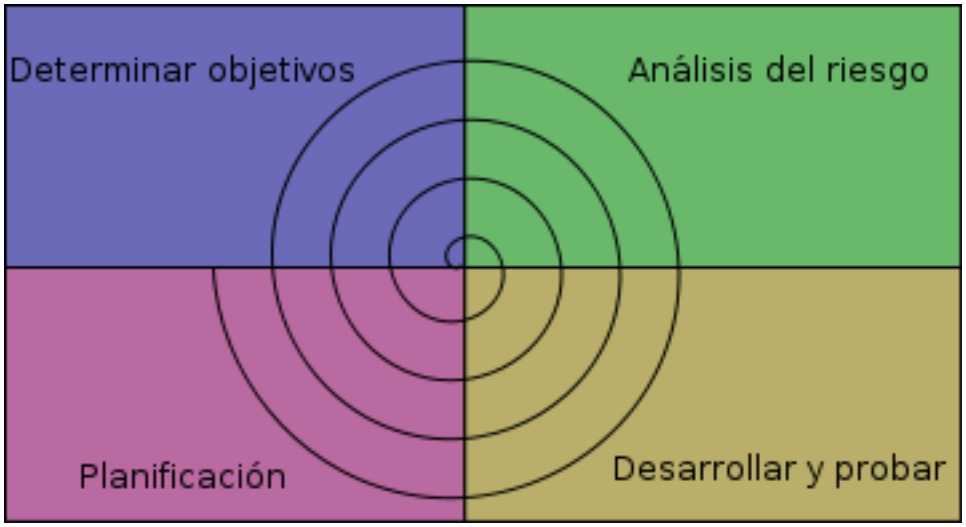


Figura 2.1: Representación del modelo de desarrollo en espiral.

- Determinar objetivos: En esta fase se fijan los objetivos que el producto debe cumplir para que el ciclo actual se considere finalizado en base a los objetivos finales. A medida que se vayan cumpliendo más iteraciones, los objetivos tendrán mayor complejidad y los ciclos serán más costosos.

- Análisis del riesgo: En base a los objetivos planteados en la fase anterior, se analizan las distintas posibilidades mediante las cuales alcanzarlos minimizando el riesgo.

- Desarrollar y probar: Se lleva a cabo el desarrollo del producto o de las partes del producto acordadas en las fases anteriores y de las pruebas pertinentes, como unitarias o de integración, para asegurar calidad en las implementaciones y que sigan funcionando en siguientes iteraciones.

- Planificación: En esta fase se estudian los resultados obtenidos por medio de las pruebas de la fase anterior. En base a ellas, se planifica la siguiente iteración teniendo en cuenta también los posibles errores cometidos.

Para seguir esta metodología en este Trabajo Fin de Grado, se han mantenido reuniones semanales con el tutor en las que se estudiaban los resultados obtenidos en cada iteración así como los problemas encontrados, se fijaban nuevos objetivos para la siguiente y se analizaban posibles vías para conseguirlos. Se ha complementado además con una wiki [1] en la página web de JdeRobot en la que se han ido registrando las tareas realizadas mediante explicaciones, imágenes y vídeos. También, como parte de la metodología, se han ido subiendo semanalmente los componentes software desarrollados a un repositorio propio de Subversion [2], que es una herramienta de control de versiones.

2.4. Plan de trabajo

En esta sección se exponen las distintas etapas en las que se ha dividido este proyecto y que corresponden con los ciclos del modelo en espiral:

- **Familiarización con el entorno JdeRobot y OpenCV**
  Durante esta etapa se descargará e instalará todo el software necesario para la realización del trabajo así como familiarizará con el entorno de JdeRobot. Esto abarca aprender el lenguaje de programación C++ (con Python ya era familiar), uso de la herramienta de compilación CMake, uso de la herramienta de control de versiones Subversion y aprendizaje básico de la librería OpenCV. Para cerrar este ciclo se desarrollarán aplicaciones que utilicen OpenCV en el entorno JdeRobot.

- **Familiarización con Gazebo y sus plugins**
  Durante este ciclo tendrá lugar una familiarización con el simulador Gazebo y se estudiará cómo es la estructura de sus plugins, cómo se compilan y cómo se instalan.

- **Desarrollo de un generador sintético de nubes de puntos**
  En esta fase se programará esta aplicación para, posteriormente, hacer pruebas con el mapeador de parches planos y modificar sus parámetros de configuración para entornos reales.

[1] `http://JdeRobot.org/Samartin-tfg`
[2] `https://svn.jderobot.org/users/samartin/tfg/`

- **Familiarización con el componente mapeador de parches planos**
  Este componente es una de las bases en las que este Trabajo Fin de Grado se asienta, así que durante este ciclo se estudiará el código del componente en profundidad además de los algoritmos que incorpora.

- **Estudio de sincronización del mapeador en entorno simulado para solucionar el problema del desfase existente entre la posición e imagen de profundidad**
  Durante esta fase se estudiarán los motivos que causan que haya un desfase entre los dos tipos de datos y se buscará una solución para conseguir una extracción de planos fluida y sin errores en el entorno simulado.

- **Familiarización con la aplicación de autolocalización y desarrollo de una capa de comunicaciones en ella para exponer la localización estimada**
  De cara a poder enviar la estimación a otras aplicaciones, se desarrollará una capa de comunicaciones en esta aplicación para que pueda interactuar con otros componentes.

- **Integración del estimador de la localización y del mapeador en un mismo sistema cuya fuente de datos sea un sensor RGB-D real**
  En este ciclo, se conectarán los componentes y se realizarán las modificaciones pertinentes para que funcione sin extracciones erróneas, de modo fluido y con movimiento suave.

Capítulo 3

Infraestructura

En este capítulo se muestra la infraestructura en la que se ha apoyado este Trabajo Fin de Grado, tanto software como hardware.

3.1. Hardware

El desarrollo y pruebas realizadas han sido ejecutadas en un ordenador con un procesador i5 3570K, 4 núcleos físicos sin Hyper-Threading funcionando a 4 GHz mediante overcloking, una tarjeta gráfica AMD Radeon HD 7700 y 8 GB de RAM.

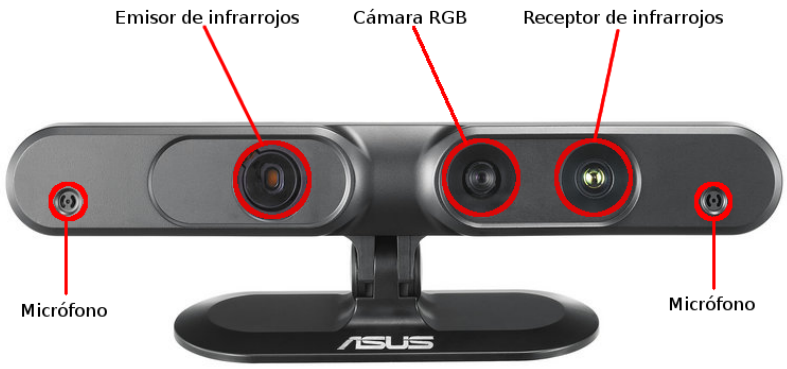


Figura 3.1: Partes del dispositivo Asus Xtion PRO LIVE.

En cuanto al sensor RGB-D utilizado, se ha elegido un Asus Xtion PRO LIVE ya que el estado de madurez del driver en el entorno JdeRobot, como veremos en la sección 3.4, se encuentra en un estado excelente. En la tabla 3.1 se pueden ver las características del dispositivo mientras que en la figura 3.1 se pueden apreciar sus partes más características.

La resolución utilizada ha sido 640x480 a 25 cuadros por segundo.

Propiedad	**Valor**
Campo de visión	58º horizontal, 45º vertical
Fotogramas por segundo	VGA (640 x 480): 30 fps, QVGA (320 x 240): 60 fps
Resolución	SXGA (1280 x 1024)
Distancia de uso	0.8 - 3.5 m
Interfaz	USB 2.0

Tabla 3.1: Especificaciones de Asus Xtion PRO LIVE.

3.2. Gazebo

Gazebo [1] es un simulador tridimensional de software libre que permite simular de manera precisa poblaciones de robots, sensores y objetos en entornos complejos de interior y exterior. Cuenta con un motor de física robusto, gráficos de buena calidad, como se puede ver en la figura 3.2, tutoriales, una API muy documentada y una amplia comunidad detrás.

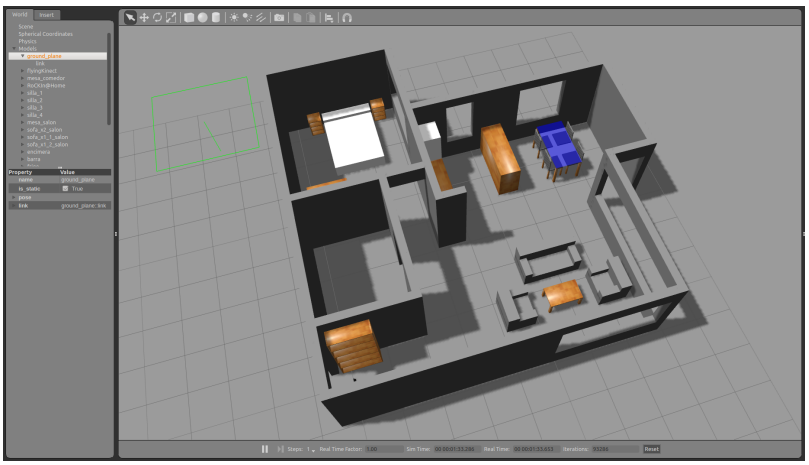


Figura 3.2: Gazebo con el mundo GrannyAnnie del campeonato RoCKin.

El desarrollo de Gazebo comenzó en 2002, en la Universidad del Sur de California, cuando el Dr. Andrew Howard y su estudiante Nate Koening precisaban de un sistema para simular robots en entornos exteriores bajo varias condiciones. Desde el año 2009

[1] `http://gazebosim.org/`

ha sido una de las principales herramientas usadas en la comunidad ROS [2] (Sistema Operativo Robótico). En 2011, empezó a recibir financiación de Willow Garage para después, en 2012, caer bajo el auspicio de la Open Source Robotics Foundation (OSRF) [3]. En 2013, este simulador se utilizó en la Virtual Robotics Challenge, un componente del DARPA Robotics Challenge.

En este proyecto, Gazebo se emplea para simular el comportamiento del sensor RGB-D y probar la solución desarrollada en un entorno controlado para después, pasar a un entorno real. La versión que se ha empleado es la 5.3.0.

3.3. ICE

ICE (Internet Communications Engine) es un entorno que proporciona una capa de abstracción para las comunicaciones entre un cliente y un servidor, con lo que no hay que preocuparse de abrir conexiones, de serializar o de intentar de nuevo conexiones fallidas, entre otros. Funciona mediante llamadas a procedimiento remoto (RPC), que consiste en que el cliente envía una llamada a un servidor para que este haga algo, como por ejemplo cambiar el valor de un dato u obtenerlo. Utiliza la orientación a objetos y cuenta con implementaciones para C++, C#, Java, Objective-C, Python, Ruby JavaScript y PHP.

ICE fue desarrollado por ZeroC [4] y se utiliza en la plataforma JdeRobot como veremos en la siguiente sección. Se encuentra bajo licencia GPLv2 [5], en el caso de utilizarlo en proyectos de software libre como este, y bajo licencia comercial para productos comerciales.

3.4. JdeRobot

JdeRobot [6] es una plataforma de software libre desarrollada por el laboratorio de robótica de la Universidad Rey Juan Carlos cuyo fin es facilitar la creación de aplicaciones en el ámbito de la robótica, visión por computador, domótica y escenarios con sensores,

[2] `http://www.ros.org/`
[3] `http://www.osrfoundation.org/`
[4] `https://zeroc.com/products/ice`
[5] `https://opensource.org/licenses/GPL-2.0`
[6] `http://jderobot.org/`

actuadores y software inteligente. Surgió en 2003 a raíz de la tesis doctoral de José María Cañas [13].

Esta plataforma se basa en componentes, que es la unidad modular funcional e independiente más básica, que a su vez se categorizan en drivers y herramientas, teniendo también una serie de librerías para utilizar en ellos. La mayoría de ellos están programados en C++ aunque también hay algunos programados en C, python y java. Para compilar y generar los ejecutables se utiliza la herramienta de software libre y multiplataforma CMake [7].

La comunicación entre cada una de las partes se hace a través de ICE, lo que simplifica el problema de la comunicación a definir las interfaces e implementarlas. Se definen por medio del leanguaje SLICE (*Specification Language for ICE*). Estas interfaces SLICE se traducen automáticamente a distintos lenguajes listas para ser implementadas en componentes. Todas estas técnicas permiten abstraerse completamente de tareas de bajo nivel, como la conexión entre las distintas partes, y centrarse en su implementación. Las principales interfaces que hemos utilizado en este trabajo son las expuestas a continuación:

Pose3D

Esta interfaz define un método `get` para obtener la localización y un método `set` para definirla. El tipo de dato con el que trabaja es Pose3Ddata y describe una posición y una orientación en un espacio tridimensional mediante un vector (x, y, z, h) y un cuaternión (q0, q1, q2, q3).

Camera

Esta interfaz se utiliza para obtener datos de una cámara, ya provengan de una cámara de profundidad o una cámara RGB. El tipo de dato que devuelve es ImageData, clase en la que viene información básica de la imagen como su descripción (altura, anchura), la imagen en sí y una estampa de tiempo de cuándo fue tomada.

PointCloud

Se utiliza para obtener datos en forma de nube de puntos, ya provengan de un generador sintético de nubes de puntos o de un servidor conectado a un sensor RGB-D. El tipo de dato que utiliza es RGBPoint, que define la posición del punto

[7] `https://cmake.org/`

mediante coordenadas x, y, z y los canales de color R, G, B en caso de que se quiera colorear con el color que ese punto tiene en el entorno.

En la actualidad, hay bastantes drivers desarrollados en la plataforma que permiten obtener datos de sensores físicos o simulados de un modo tan sencillo como llamar a una función mientras el driver está ejecutándose. Ejemplo de ello son drivers para cámaras normales, sensores RGB-D, para el robot KobuKi o para plugins en el simulador Gazebo. Los drivers utilizados específicamente en este Trabajo Fin de Grado han sido los siguientes:

CameraServer

Es un servidor de imágenes basado en OpenCV. Permite obtener fotogramas de un archivo de vídeo, aceptando una gran variedad de formatos, o leer de un dispositivo conectado, como una cámara de vídeo o la cámara a color de un sensor RGB-D.

FlyingKinect

Es un plugin para el simulador Gazebo que tiene varias funcionalidades. Por un lado, expone las imágenes a color y profundidad de un sensor RGB-D simulado y la nube de puntos de la escena. Por otro lado, tiene otra interfaz de Pose3D en el que se puede consultar la posición del sensor en el mundo así como modificarlo, permitiendo que otro componente se conecte a él para teleoperarlo.

OpenniServer

Es un servidor que utiliza la biblioteca OpenNI para dar soporte a sensores RGB-D, como el Asus Xtion. En sus interfaces expone la imagen a color, la imagen de profundidad y la nube de puntos 3D frente al sensor. En la figura 3.3 se puede ver la conexión al sensor y las interfaces en las que expone los datos.

Las herramientas están concebidas para conectarse a drivers y explotar o extender su funcionalidad. En esta categoría entran componentes de tipo visor, controladores manuales, generadores y registradores de datos y algoritmos para detección o actuación. En este trabajo se han utilizado las siguientes:

NavigatorCamera

Permite teleoperar cualquier plugin de Gazebo que tenga una interfaz Pose3D y una cámara a color. Este componente muestra la información de la cámara RGB en una ventana mientras que en otra muestra la información de la localización del sensor

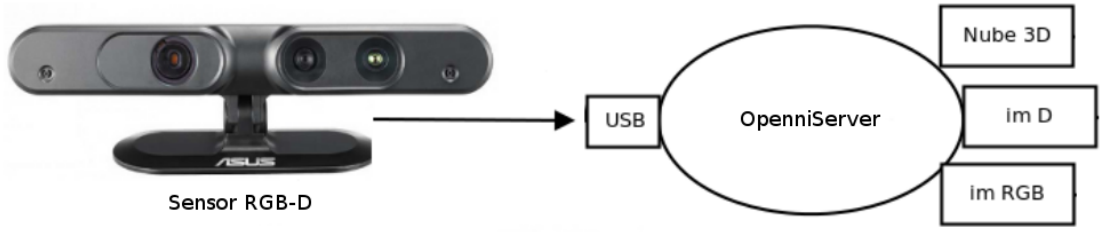


Figura 3.3: OpenniServer y sus interfaces ICE.

o robot y permite modificarla. Se ha utilizado para mover por el entorno el plugin `flyingKinect` descrito arriba.

Recorder y Replayer

Por un lado, `recorder` permite conectarse a interfaces ICE de otro componente y grabar simultáneamente la información que pasa por todas ellas. De este modo se generan datos enlatados. Por otro, `replayer` es capaz de reproducir esos datos y exponerlos en los mismos tipos de interfaz ICE para que otro componente se conecte a ellas. Permiten grabar y reproducir en multitud de tipos de interfaces, como localización, imágenes de cámara, laser, nube de puntos y posición. Uno de los principios básicos del método científico es aislar lo que se está probando tanto como sea posible, y estas dos herramientas lo permiten. Un ejemplo sería grabar la salida de la cámara RGB y de la cámara de profundidad de un sensor RGB-D para luego hacer las pruebas de los algoritmos sobre los mismos datos y poder ver sus variaciones. En la figura 3.4 se pueden ver las interfaces que se configurarían cuando `replayer` reproduce los datos de un sensor RGB-D.

Las librerías están diseñadas para aligerar y mejorar el desarrollo y diseño del resto de componentes. En este trabajo se ha utilizado la librería de JdeRobot ParallelIce. Típicamente, la estructura básica de un componente de JdeRobot constaría de dos hilos, uno encargado de la interfaz gráfica y otro de la captura secuencial de la información a fuentes de datos, como drivers o el componente `replayer`, del procesamiento y control. La captura se realiza mediante ICE utilizando llamadas a procedimiento remoto, como ya hemos visto, lo que introduce algo de latencia desde que se pide hasta que llega el

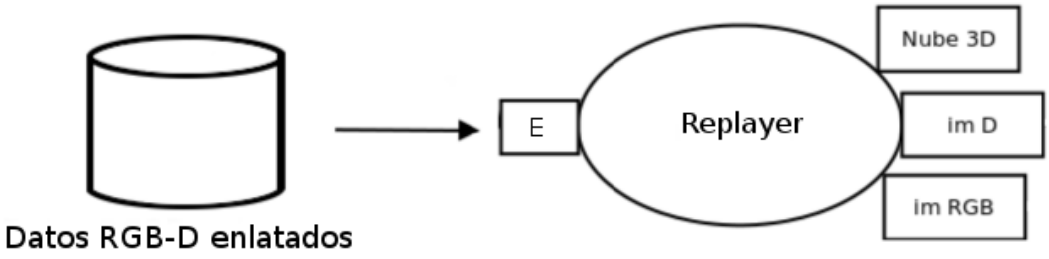


Figura 3.4: Replayer reproduciendo datos RGB-D.

dato. En la figura 3.5 podemos ver este hilo de captura, procesamiento y control pidiendo datos a un servidor. Esta arquitectura secuencial es idónea cuando el tiempo de acceso a la información es bajo y no se utilizan varias fuentes de datos.

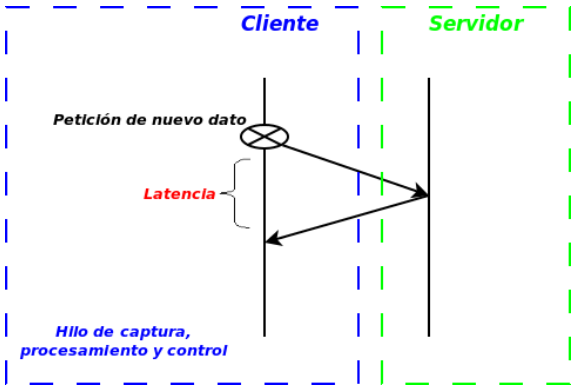


Figura 3.5: Latencia con un único hilo.

Cuando los datos que se mueven a través de ICE son voluminosos o cuando se accede a más de una interfaz, como es el caso de este proyecto, la latencia que se introduce en esperar la respuesta es bastante notable. Aquí es donde entra en juego ParallelIce. Mediante esta librería se crea un hilo por cada conexión ICE, que estará dedicado a hacer peticiones al servidor a un ritmo configurable, para que, cuando el hilo principal quiera el dato, se lo pida al hilo creado por ParallelIce y no al servidor, lo que reduce la latencia drásticamente. En la figura 3.6 podemos ver el funcionamiento descrito.

De este modo, pasamos de una estructura de dos hilos, interfaz gráfica y captura, procesamiento y control, a una estructura de 2+n hilos, una para la interfaz, otro para el

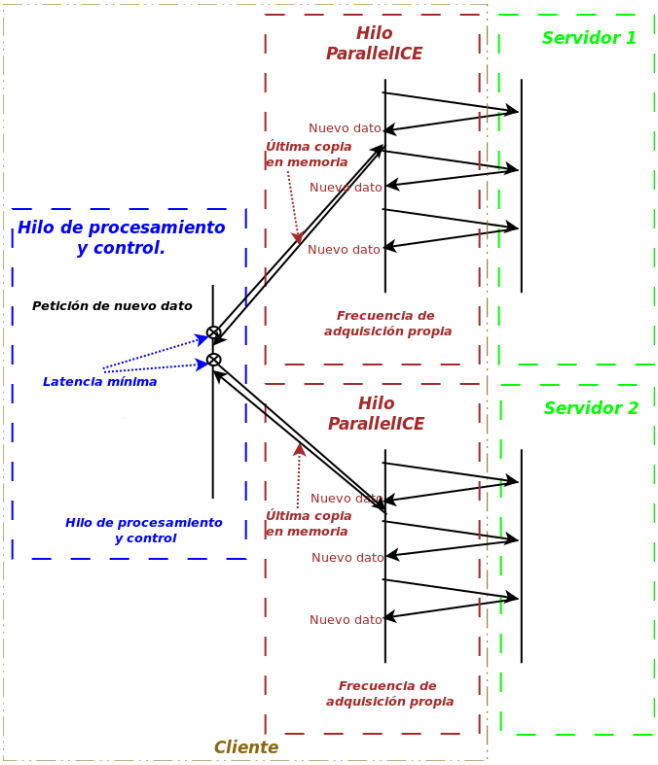


Figura 3.6: Latencia utilizando ParallelIce.

procesamiento y control y n hilos más para la captura, en función del número de interfaces a las que esté conectado el componente.

3.5. Componente cam_autoloc

Este componente es el resultado del Trabajo Fin de Máster [11] de Alberto López-Cerón Pinilla. Permite estimar la posición y orientación de una cámara, simulada en Gazebo o real, a partir de marcadores visuales de AprilTags. Está programada en C++.

El algoritmo que transforma las imágenes en dos dimensiones se puede representar como una caja negra, figura 3.7, que ha de inicializarse con cierta información a priori: los parámetros de calibración intrínsecos de la cámara y la posición y orientación de las balizas en el mundo. De forma opcional también se le puede aportar información de posición verdadera, que junto con la estimada, forma el contenido del fichero de log.

El proceso de localización se lleva a cabo fundamentalmente en cuatro pasos: análisis de imagen 2D, cálculo de 3D instantáneo, fusión espacial y fusión temporal. En primer

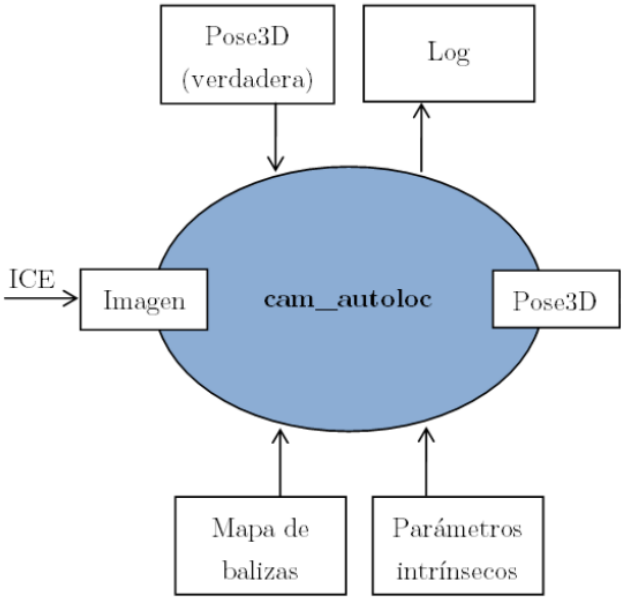


Figura 3.7: Diagrama de caja negra de la aplicación `cam_autoloc`.

lugar se detectan las balizas en la imagen, después se realiza el cálculo de la ubicación relativa para cada una de ellas, a continuación se fusionan las estimaciones obtenidas a partir de cada baliza, con mayor peso las que están más cerca, y se finaliza con la fusión de estimaciones pasadas, para evitar cambios abruptos.

Dispone de una interfaz de usuario, figura 3.8, en la que aparece la estimación en un momento dado, un botón para mostrar un mundo 3D en el que se puede ver la localización de la cámara respecto de las balizas registradas, un menú para activar la fusión temporal, con dos técnicas distintas, y la imagen capturada por la cámara sobre la que se superpone una señalización de las balizas detectadas. Un hilo en la aplicación es el dedicado a actualizar esta interfaz, que lleva a cabo además el procesado de las imágenes y su conversión a localización estimada, otro hilo se encarga del mundo 3D y otro hilo de actualizar la imagen (que funciona como si de un hilo ParallelIce se tratase aunque no utiliza directamente esta librería).

Esta aplicación será la que se utilizará en este Trabajo Fin de Grado para resolver la localización de la cámara en el contexto de mapear un entorno real a partir de un sensor RGB-D. Para ello, será necesario hacerle unas cuantas modificaciones, como por ejemplo, añadirle una capa de comunicaciones para poder enviar la localización estimada a otra aplicación.

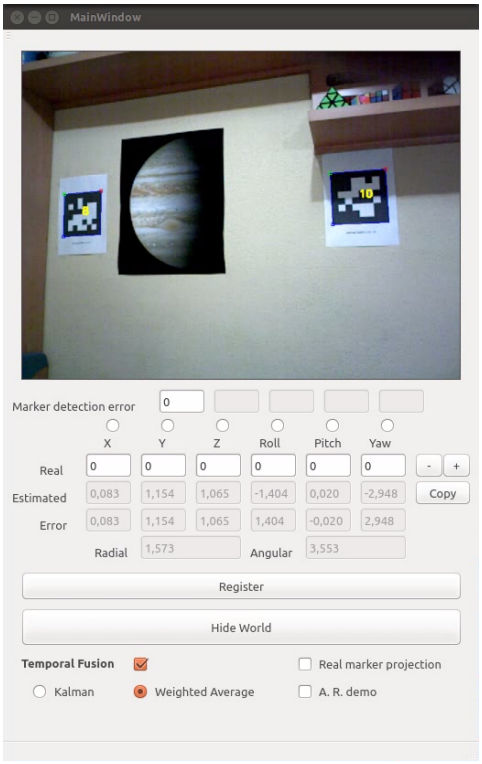


Figura 3.8: Interfaz de la aplicación cam_autoloc.

3.6. Componente mapper3Drgbd

Juan Navarro Bosgos desarrolla este componente en su Proyecto Fin de Carrera [10] en el que implementa un algoritmo, en tiempo real, para la construcción de mapas del entorno que rodea a un sensor RGB-D mediante parches planos. El lenguaje de programación con el que se ha desarrollado es C++.

El algoritmo desarrollado tiene como entrada instantánea la posición e imagen de profundidad del sensor, y como salida mapas extraídos de la nube de puntos que se genera en combinación de las dos entradas citadas y los parámetros de calibración intrínsecos de la cámara. En la figura 3.9 se puede ver el diagrama de caja negra simplificado.

En cada iteración del algoritmo desarrollado, primero se captura la posición y la imagen de profundidad del instante actual para generar la nube de puntos. A continuación, se clasifica cada punto del instante actual como: perteneciente a alguno de los parches planos ya resueltos, cercano a algún parche plano o no explicado. Con los puntos pertenecientes a algún parche no se hace nada mientras que los cercanos se utilizan para redefinir el

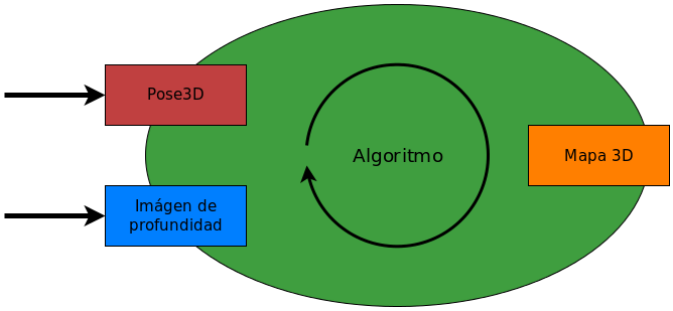


Figura 3.9: Diagrama de caja negra de la aplicación mapper3Drgbd.

contorno de ese parche. Por otro lado, con los puntos no explicados se intenta generar nuevos parches planos empleando el algoritmo RANSAC. Al final de la iteración, se estudian los parches para ver si alguno es parte del mismo plano y están lo suficientemente cerca como para fusionarlos.

Hay un hilo dedicado a ejecutar el algoritmo descrito, otro hilo se utiliza para la interfaz gráfico y otros dos hilos se emplean para capturar la imagen de profundidad y la posición utilizando la librería ParallelIce. En la interfaz gráfico descrito, figura 3.10, se puede activar o desactivar la extracción de planos, modificar los parámetros del algoritmo, utilizar funciones de depuración o ver la nube de puntos, planos extraídos y un modelo de sensor RGB-D.

Esta aplicación solo ha sido probada en entornos simulados en Gazebo, en los que tanto la imagen de profundidad como la posición, vienen del simulador (y la posición es conocida, a diferencia de lo que pasa en el mundo real). Actualmente, existe un desfase temporal entre la imagen de profundidad y la posición que imposibilita una extracción de planos fluida mientras el sensor se mueve. Cuando varía la posición del sensor, a veces, llega antes un tipo de dato que el otro generando una nube de puntos errónea, y por tanto una extracción del plano errónea.

Se realiza el siguiente procedimiento para mapear el entorno: se mueve el sensor, se para, se espera uno o dos segundos, se activa el algoritmo de extracción de planos, se para el algoritmo (una vez ya extraídos) y vuelta a empezar. Juan Navarro muestra este funcionamiento en un vídeo [8] en su wiki de JdeRobot. Por tanto, antes de incorporar esta aplicación en nuestro sistema de mapeado de entornos reales con movimiento suave,

[8] `http://jderobot.org/Jnbosgos#Mapper_3D_RGBD`

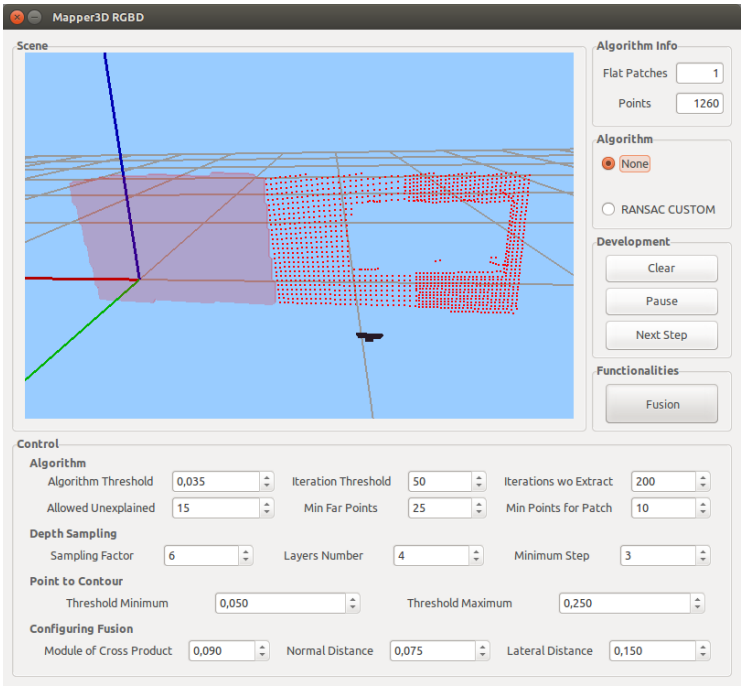


Figura 3.10: Interfaz de la aplicación mapper3Drgbd.

habrá que hacer un estudio del sincronismo en el entorno simulado.

3.7. GTK+ y Glade

GTK+ [9], o the GIMP ToolKit, es un juego de herramientas multiplataforma para crear interfaces gráficas de usuario. Licenciado bajo los términos de GNU LPGL [10], GTK+ es software libre y parte del proyecto GNU. Está escrita en su totalidad en C pero tiene soporte para una gran variedad de lenguajes. En este trabajo se ha utilizado concretamente gtkmm [11], que es la interfaz para C++.

Glade [12] es un diseñador de interfaces visuales para el set de herramientas GTK+ y el entorno GNOME. Las interfaces diseñadas con Glade se guardan en formato XML y son modificables en tiempo de ejecución gracias a la biblioteca `libglade`. Tiene soporte para numerosos lenguajes de programación y es software libre bajo la licencia GNU GPL [13].

[9] `http://www.gtk.org/`
[10] `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`
[11] `http://www.gtkmm.org/`
[12] `https://glade.gnome.org/`
[13] `https://www.gnu.org/licenses/gpl-3.0.html`

3.8. OpenCV

OpenCV (Open Source Computer Vision Library) [14] es una librería de software libre de visión artificial y de *machine learning*. Fue originalmente desarrollada por Intel en 1999 y desde entonces no ha parado de crecer, teniendo en la actualidad más de 500 algoritmos. Cuenta con una licencia BSD [15] por lo que incluso aplicaciones comerciales pueden utilizarla y modificar su código.

OpenCV es multiplataforma, existiendo versiones para GNU/Linux, Windows y Mac OS X. Tiene funciones para C++, C, Python, Java y MATLAB, mientras que la propia librería está programada en C++, aprovechando las capacidades que ofrecen los procesadores multinúcleo y la aceleración GPU.

A día de hoy, esta librería es utilizada tanto por grandes empresas, como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda y Toyota, como por *startups* y plataformas como JdeRobot.

En este trabajo se utilizó OpenCV en la fase de familiarización con el entorno. En esta fase se desarrollaron dos componentes dentro de la plataforma JdeRobot que toman imágenes de la cámara del ordenador gracias al driver `cameraServer` del que hablamos en la sección 3.4. Ambos componentes tienen una estructura de dos hilos en la que uno se dedica a la interfaz gráfica y el otro a la captura, procesamiento y control de datos. Para la interfaz gráfica se utilizaron las herramientas descritas en la sección anterior 3.7. La funcionalidad implementada en los componentes abarca detección de caras y ojos, detección de bordes y esquinas, distintos tipos de filtros de desenfoque y filtros de color. En la figura 3.11 se puede ver uno de ellos. La funcionalidad completa de estos componentes se puede ver en la wiki [16] mientras que el código en el repositorio de Subversion [17]. La versión utilizada ha sido la 2.4.8.

[14] `http://opencv.org/`

[15] `https://www.freebsd.org/doc/es_ES.ISO8859-1/articles/explaining-bsd/article.html`

[16] `http://jderobot.org/Samartin-tfg#Playing_with_OpenCV`

[17] `https://svn.jderobot.org/users/samartin/tfg/trunk/components/`

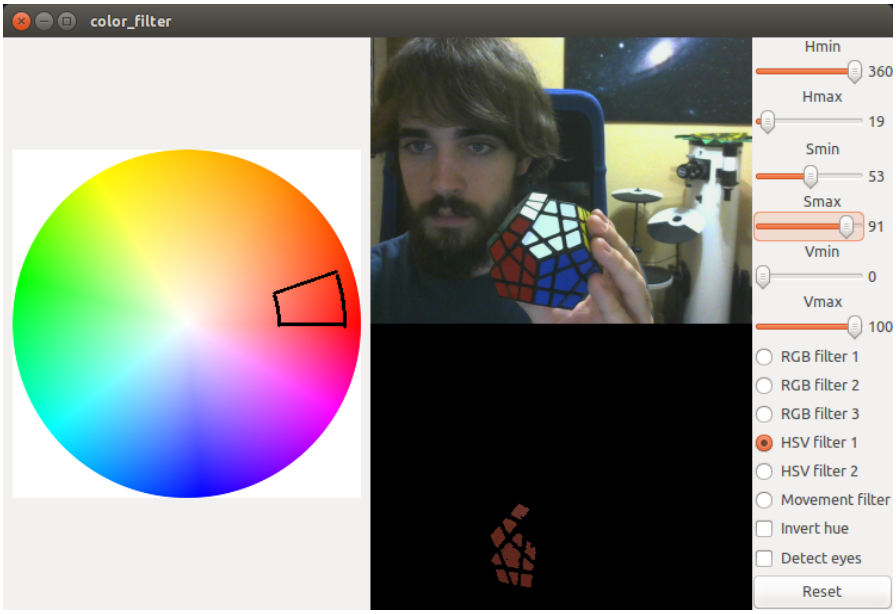


Figura 3.11: Componente propio color_filter.

3.9. AprilTags

AprilTags [18] es un sistema de detección de balizas o marcadores visuales en 2D desarrollado en 2011 por Edwin Olso [14]. Para ello, propone un método para detectar de manera robusta las balizas y diseña un algoritmo de segmentación basado en gradientes locales que consigue estimar las líneas con precisión. También describe un sistema de codificación para los marcadores que aborda problemas específicos de los sistemas de códigos de barras 2D: robustez frente a rotación y frente a falsos positivos.

La librería permite estimar la posición 3D, orientación e identidad de los marcadores respecto de la cámara. Esta librería se ha utilizado en este trabajo ya que el componente `cam_autoloc` la utiliza.

3.10. Eigen

Eigen [19] es una librería de C++ de alto nivel para álgebra lineal con licencia MPL2 [20], por lo que es software libre. Facilita las operaciones con vectores y matrices y ofrece

[18] `https://april.eecs.umich.edu/wiki/AprilTags`
[19] `http://eigen.tuxfamily.org/`
[20] `https://www.mozilla.org/en-US/MPL/2.0/`

métodos de resolución de sistemas lineales.

En este trabajo se ha utilizado para hacer operaciones con la orientación de la cámara, que involucran aṕgulos de rotación en los ejes y cuaterniones, y para generar matrices de rotación y traslación necesarias para el correcto funcionamiento de la aplicación final. La versión utilizada ha sido la 3.2.0.

Capítulo 4

Desarrollo software

Hasta ahora, el componente `mapper3Drgbd` solo era capaz de construir mapas de parches planos en entornos simulados, con la limitación de tener que ir moviendo y parando el sensor en cada extracción. Este problema se debía fundamentalmente a un problema de sincronización entre las imágenes de profundidad y la posición, que desembocaba en la extracción de parches planos en posiciones erróneas. Desde este punto de partida, el objetivo principal es que el mapeador sea capaz de construir mapas de parches planos de forma fluida y, además, en entornos reales.

En este capítulo se describe la solución desarrollada. En primer lugar, se dará una visión global de los distintos sistemas utilizados y desarrollados. Después, se detallará el componente generador sintético de nubes de puntos. A continuación, se expondrán los resultados del estudio del sincronismo del mapeador de planos en entornos simulados, así como las soluciones diseñadas para evitar el desfase entre imagen de profundidad y posición. Finalmente, se describirá la integración del sistema realizada para que funcione en un entorno real y también todas las modificiones implementadas.

4.1. Diseño global

Como solución final, se ha diseñado el sistema que se puede ver en la figura 4.1. Las cajas representan las interfaces ICE de los componentes mientras que las flechas indican el sentido de la información.

Se han utilizado los componentes `openniServer`, `cam_autoloc` y `mapper3Drgbd`, ya existentes en la plataforma JdeRobot pero que ha habido que adaptar y modificar. Como

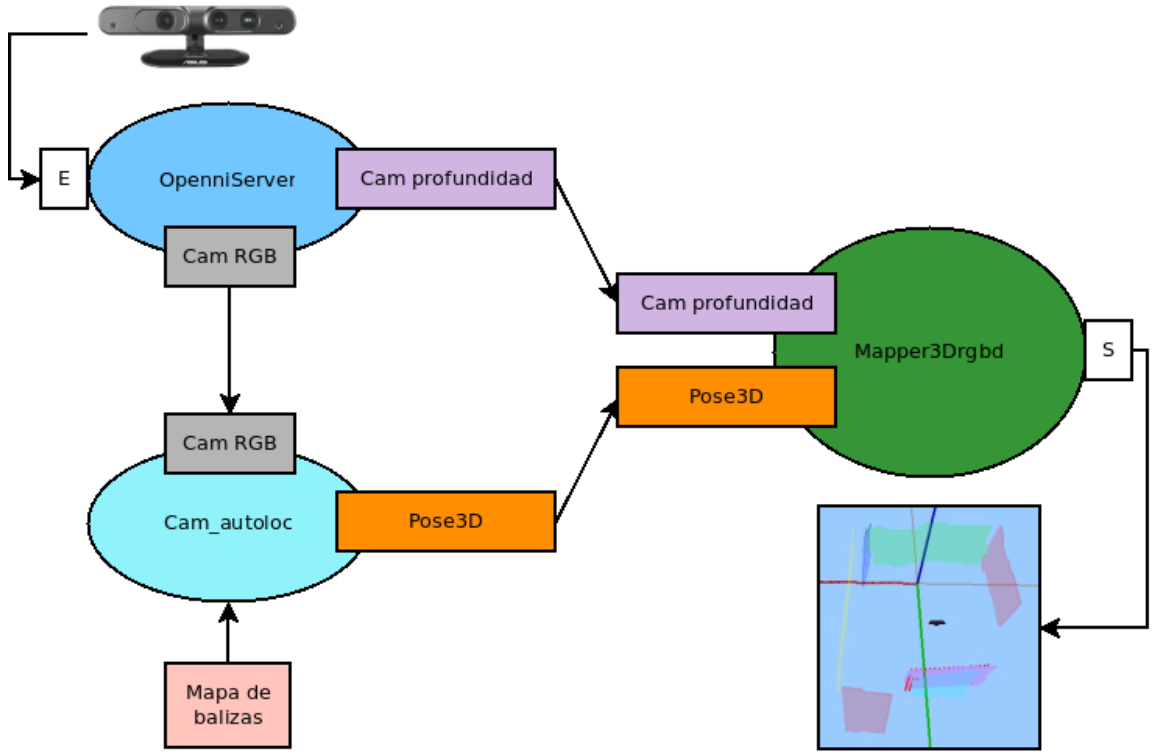


Figura 4.1: Esquema de conexiones del entorno real.

fuente de datos tenemos el dispositivo Asus Xtion PRO LIVE conectado al driver de JdeRobot `openniServer`, que sirve la cámara a color y la cámara de profundidad. El componente `cam_autoloc` se conecta a la cámara a color del servidor para estimar la posición y orientación del dispositivo en el mundo, y exponer esta estimación a través de una interfaz de Pose3D. Para realizar la estimación, necesita también la localización de las balizas situadas en el mundo. Finalmente, el mapeador se conecta a la interfaz de cámara de profundidad del driver y a la interfaz de Pose3D del componente de autolocalización para construir la nube de puntos con la posición y orientación del sensor ya resuelta.

Para que todos los componentes puedan estar conectados y funcionar en entornos reales ha sido necesario realizar una serie de modificaciones. En primer lugar, se ha desarrollado un generador sintético de nubes de puntos para hacer un ajuste de los parámetros del algoritmo de mapeador para su optimizacion en entornos reales. En segundo lugar, ha sido necesario realizar un estudio de la sincronización y buscar soluciones para el desfase que presentaba el componente `mapper3Drgbd` entre las imágenes de profundidad y las posiciones. En tercer lugar, se ha añadido una capa de comunicaciones ICE al componente `cam_autoloc` para que pueda proporcionar la localización que estima a otros componentes.

En cuarto lugar, se ha realizado un encaje de los sistemas de referencia que utiliza el componente de autolocalización y el mapeador ya que eran distintos y la posición que el mapeador recibía no resultaba en una correcta nube de puntos al ser combinada con la imagen de profundidad. Finalmente, se ha realizado una serie de modificaciones y optimizaciones que son exclusivos de este sistema, como por ejemplo, no extraer planos si no se ven balizas visuales o reducir la carga que el sistema tiene en la CPU.

Además del sistema principal con datos reales que se acaba de ver, en este trabajo se han utilizado tres entornos más que corresponden a distintas fuentes de datos: datos sintéticos, simulados y enlatados (grabados). Estos entornos han servido para realizar mejoras y pruebas. En la figura 4.2 se puede ver el esquema de conexiones.

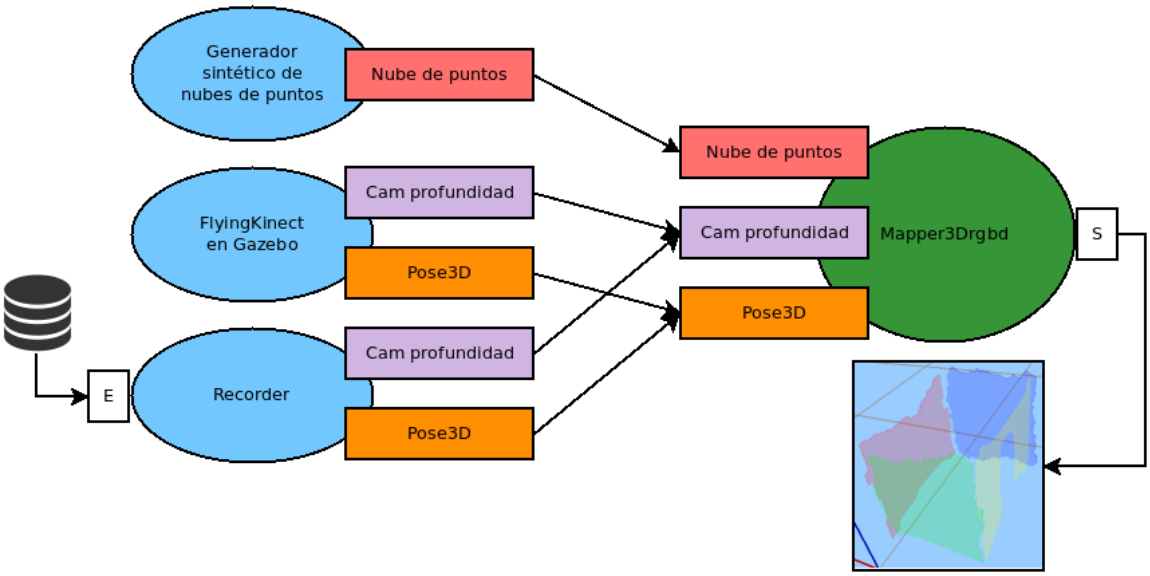


Figura 4.2: Esquema de conexiones de los distintos entornos.

En este trabajo se ha desarrollado un generador sintético de nubes de puntos. Se conecta al mapeador a través de un interfaz de nube de puntos y cuando este último recibe datos de esta fuente, las interfaces de Pose3D y cámara de profundidad se deshabilitan puesto que no son necesarias. En este caso, tampoco se dibuja el sensor RGB-D en el mundo que aparece en la interfaz de usuario del mapeador. Este es el único sistema en el que se utiliza la interfaz de nube de puntos del extractor de planos, en el resto estará deshabilitada y se utilizarán las otras dos. Este sistema principalmente se ha utilizado para optimizar los parámetros del algoritmo de extracción de parches planos para entornos reales.

Como fuente de datos en el sistema simulado, se utiliza el plugin `flyingKinect` ejecutándose en el simulador Gazebo que, como ya vimos en la sección 3.4, simula un sensor RGB-D. Posee una interfaz de cámara de profundidad, otra de cámara a color y otra de Pose3D. El mapeador se conecta a las interfaces de Pose3D y cámara de profundidad del plugin para construir la nube de puntos y extraer planos de ella. Con este entorno se ha estudiado el sincronismo de los datos que llegan al mapeador y buscado una solución.

Los datos enlatados son una gran manera de probar las modificaciones en los algoritmos ya que, al no variar, se ven con claridad los resultados de distintas variantes algorítmicas sobre exactamente los mismos datos. Para exponer los datos enlatados se utiliza la herramienta `replayer`, que los toma como entrada para exponerlos por interfaces de Pose3D y cámara de profundidad para que, a continuación, el mapeador se conecte y los obtenga. Este sistema se ha utilizado para comparar en simulación el mapeador con las mejoras desarrolladas para evitar el desfase de datos con el mapeador original sin las mejoras.

La salida del mapeador, sea el entorno o la fuente de datos que sea, es un mapa tridimensional compuesto por parches planos grandes. Los planos se extraen de la nube de puntos mediante un algoritmo basado en RANSAC en base a unos parámetros configurables. Estos parámetros habrá que ajustarlos para el correcto funcionamiento en entornos reales.

4.2. Generador sintético de nubes de puntos

Este componente genera un número configurable de puntos pertenecientes a uno o varios planos que estén dentro de los límites de un paralelepípedo (volumen de trabajo), con cierto error gaussiano en la dirección perpendicular al plano. Los puntos resultantes simulan la nube de puntos obtenida a partir de un sensor RGB-D, que también tendrá cierto error gaussiano. Este componente lo hemos llamado `pointCloudGenerator` y en la figura 4.3 se puede ver su diagrama de caja negra.

Tiene una única entrada que consiste en un fichero de configuración, siguiendo el convenio de la plataforma JdeRobot, con la información ICE para la conexión, los planos en los que generar puntos, cuántos puntos por plano, el error gaussiano a aplicar y los puntos que definen el volumen de trabajo.

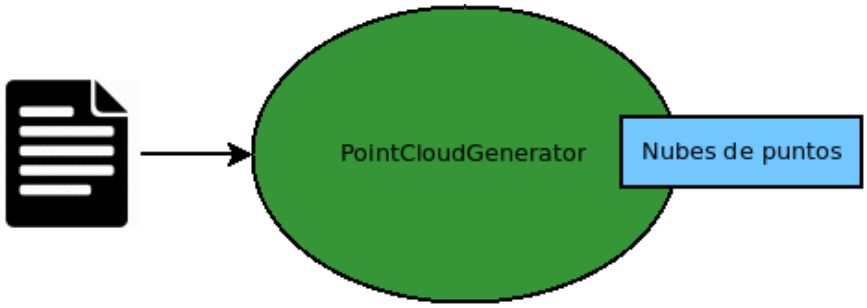


Figura 4.3: Diagrama de caja negra del componente pointCloudGenerator.

El desarrollo se ha realizado en Python siguiendo orientación a objetos y buenas prácticas, ocupando aproximadamente unas 460 líneas de código en total. El código fuente está disponible en el repositorio de Subversion [1].

Para los cálculos geométricos se han creado clases personalizadas de plano, línea, vector y punto que contienen métodos para realizar operaciones básicas de geometría. Para mostrar los puntos generados en un entorno tridimensional, se ha utilizado la librería de Python `matplotlib` [2].

Se ha implementado un servidor ICE dentro del componente para exponer las nubes de puntos generadas a través del interfaz pointCloud de JdeRobot. Para ello se ha añadido el código necesario que consiste en: una incialización del objeto adaptador de ICE y una clase que implemente el interfaz pointCloud con un método `get`. De este modo, cualquier otra aplicación que utilice esta interfaz puede conectarse a él.

En la figura 4.4 se puede ver un ejemplo de la gráfica tridimensional que ofrece la aplicación para observar las nubes de puntos creadas. Además de los puntos, se puede ver un eje de coordenadas, el paralelepípedo y las líneas de intersección entre el plano y el volumen en rojo.

La generación de los puntos en el espacio tridimensional consta de los siguientes pasos:

1. Se calculan los puntos de intersección de las doce aristas del paralelepípedo con el plano. En caso de que no intersecten no se seguirá con el algoritmo pues el plano está fuera del volumen de trabajo.

2. De esos puntos calculados, se obtiene el valor mínimo y máximo que tienen

[1] `https://svn.jderobot.org/users/samartin/tfg/trunk/components/pointCloudGenerator/`
[2] `http://matplotlib.org/index.html`

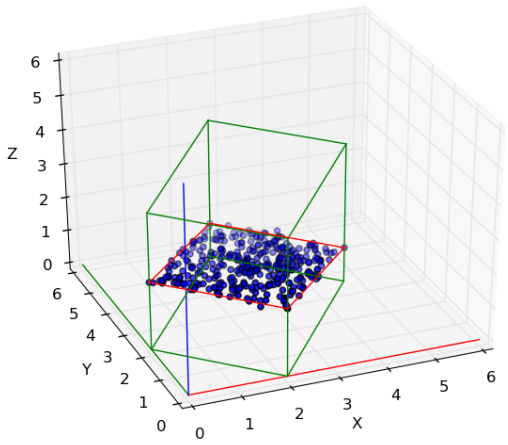


Figura 4.4: Gráfica tridimensional del componente pointCloudGenerator.

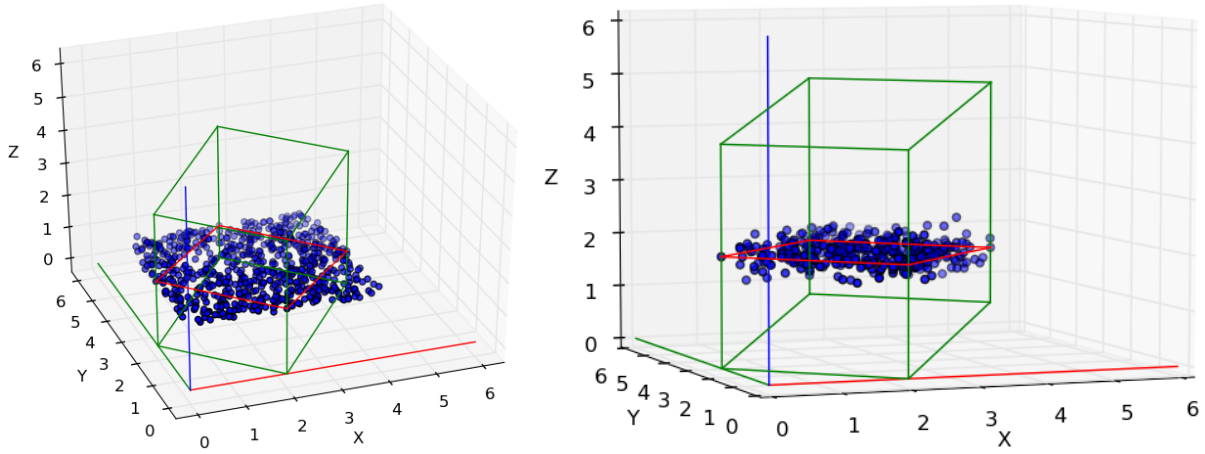
en cada eje para, a continuación, generar con esos límites puntos utilizando la ecuación general del plano 4.1. Se dan valores generados de forma aleatoria a dos componentes, calculándolos con una distribución uniforme cuyos parámetros serán los límites obtenidos, y se obtiene la tercera despejándola. Si el plano es constante en alguna de las componentes, como el eje Z en la figura 4.4, se tiene en cuenta para que esa componente sea la que se despeja, no en la que se dan valores. En este momento los puntos generados se pueden ver en la figura 4.5(a).

$$A \cdot x + B \cdot y + C \cdot z + D = 0 \tag{4.1}$$

3. Para realizar el filtrado de los que no están en el volumen de trabajo, en primer lugar, se calculan los seis planos a los que pertenecen las caras del volumen. Después, para cada punto generado, se calculan sus proyecciones ortogonales a esos seis planos, que resulta en seis puntos. El punto estará dentro del volumen de trabajo si ninguna de sus componentes es mayor o menor que las componentes de los puntos de proyección obtenidas de los planos paralelos.

4. Para finalizar, a cada punto dentro del volumen de trabajo se le añade un valor de ruido obtenido de una distribución gaussiana en la dirección perpendicular al plano. En la figura 4.5(b) se puede ver el resultado de una generación de nube de puntos a la que se le ha añadido bastante ruido gaussiano.

Para que el componente `mapper3Drgbd` pueda obtener puntos de este componente ha



(a) Puntos generados sin filtrar.

(b) Puntos filtrados con error gaussiano.

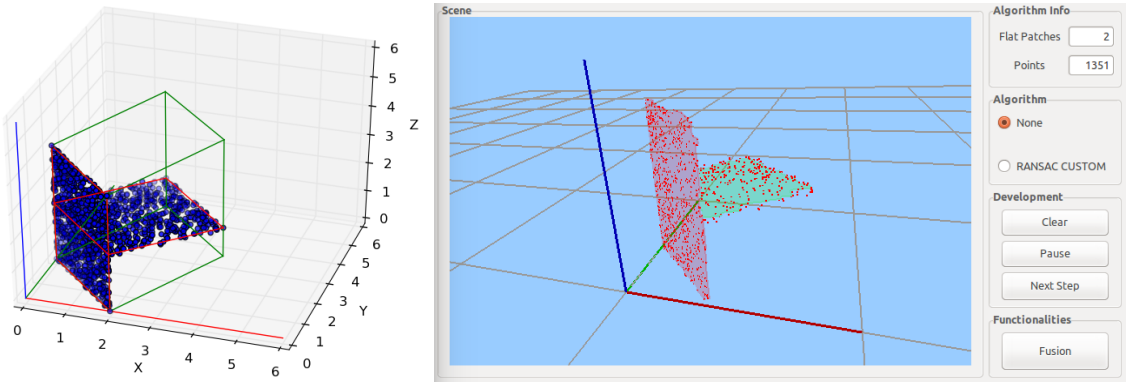
Figura 4.5: Distintas fases de la generación.

hecho falta realizarle una serie de modificaciones. En primer lugar, esta aplicación debe ser capaz de realizar labores distintas a cuando está funcionando en simulación, que es como lo hacía hasta ahora. Es por ello que el primer paso ha sido añadir un campo en su fichero de configuración en el cual se indica si los datos provienen de una fuente sintética o simulada en Gazebo. Después, se le ha añadido una interfaz de nube de puntos, ya que solo tenía interfaces de cámara de profundidad y Pose3D. Cuando se conecta a esta interfaz, elegible desde el archivo de configuración, las otras dos se deshabilitan automáticamente. Para finalizar, se ha implementado otro método para que el mapeador no modifique la nube de puntos con la Pose3D del sensor, debido a que en este escenario no lo hay. En la figura 4.6(a) se pueden ver dos nubes de puntos generadas y en la 4.6(b) al mapeador recibiéndolas y extrayendo dos planos a partir de ellas.

4.3. Sincronización profundidad-posición en simulación

Como ya se ha comentado en otras secciones, cuando los datos de profundidad del sensor y de su posición no están sincronizados se generan extracciones erróneas de parches planos. Para que la extracción de planos del entorno pueda realizarse de forma fluida y sin fallos, es imprescindible la sincronización entre ambos.

Para mover el sensor simulado en Gazebo y estudiar la sincronización se ha utilizado



(a) Nubes de puntos en pointCloud-Generator.

(b) Nubes de puntos en mapper3Drgbd.

Figura 4.6: Conexión entre los dos componentes.

el componente `navigatorCamera`. Este componente de JdeRobot se conecta a la interfaz de Pose3D del plugin `flyingKinect`, para cambiar los valores de posición de este, y a la interfaz de cámara a color, para mostrar en una ventana las imágenes de su cámara RGB. En la figura 4.7 se puede ver el esquema de conexiones del entorno simulado.

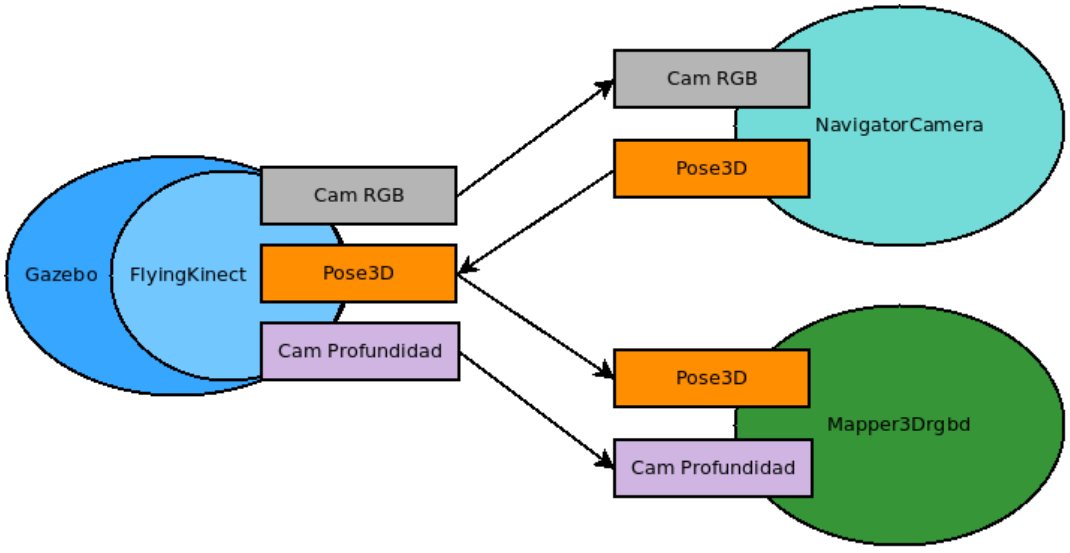
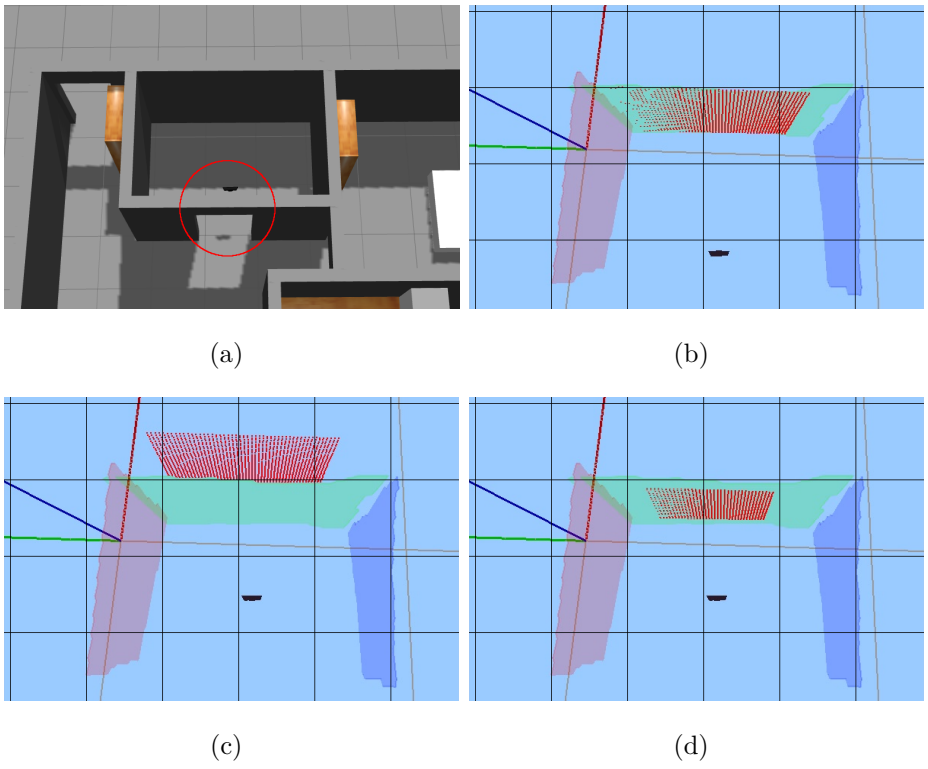


Figura 4.7: Esquema de conexiones del entorno simulado.

El desfase entre los dos datos se puede ver con una sencilla demostración. Partimos de un sensor parado al lado de la puerta en la habitación cuadrada de la figura 4.8(a). En las siguientes tres imágenes se puede ver una sucesión del movimiento del sensor en el mundo del mapeador a las que se les ha añadido una rejilla para que se aprecie mejor su

posición. En la figura 4.8(b) se puede observar la posición de inicio y también tres planos ya extraídos realizando el procedimiento descrito al final de la sección 3.6, que consistía en mover el sensor sin que el algoritmo estuviese activo, pararlo, esperar un poco y activar la extracción de los planos. Partiendo de esa posición, hacemos un único movimiento en el que el sensor avanza 60 centímetros de golpe. En la figura 4.8(c) se puede ver la posición del sensor inmediatamente después del movimiento. La nube de puntos, no obstante, está mal colocada, y es debido a que está haciendo su cálculo a partir de la imagen de profundidad del instante anterior, cuando la pared estaba más lejos. Esperando unos instantes más, la imagen de profundidad del momento actual llega y la nube de puntos se calcula donde corresponde, como se puede ver en la figura 4.8(d).

Con este ejemplo se puede deducir que la imagen de profundidad llega algo más tarde que la posición. Esto solo sucede en algunas ocasiones, pero este es un sistema en el que el sincronismo es crítico y aunque solo estén desfasados durante 1 milisegundo, es suficiente para que un plano se extraiga en una localización errónea. En el ejemplo anterior no se generó un plano en la nube de puntos de 4.8(c) porque el algoritmo de extracción estaba desactivado para que se viesen mejor los puntos mal posicionados.

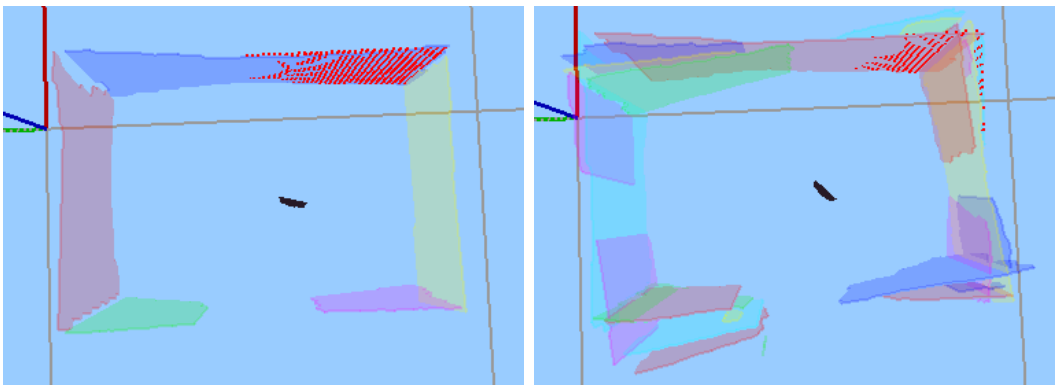


(a) (b)

(c) (d)

Figura 4.8: Ejemplo de desfase entre los datos.

Cuando se extraen planos mientras se mueve el sensor, haciendo pequeños pero continuos movimientos de rotación como en movimientos reales, el efecto es más visible. En la figura 4.9(a) se puede ver la habitación del ejemplo anterior reconstruida con el procedimiento de mover y parar el sensor para extraer los planos. En la figura 4.9(b) se puede observar la misma habitación reconstruida mientras el sensor rota sobre sí mismo con el algoritmo de extracción activo para que se haga de forma fluida, lo que da lugar a multitud de extracciones erróneas.



(a) (b)

Figura 4.9: Desfase con movimiento continuo.

4.3.1. Análisis del sincronismo

Para estudiar por qué un dato llega antes que el otro, es preciso conocer todos los hilos involucrados, la frecuencia a la que funcionan y todas las partes por las que la información pasa, desde su origen en Gazebo hasta la recepción en el mapeador. En la figura 4.10 se puede ver este diagrama de bloques.

En la parte del mapeador, como ya hemos visto en la sección 3.6, encontramos un hilo de control en el que se ejecuta el algoritmo y dos hilos de captura que utilizan la librería `ParallelIce`, uno para la Pose3D y otro para la imagen de profundidad. Se probó a subir la frecuencia de estos hilos de captura por si iban desfasados entre ellos, pero la información seguía llegando igual de desfasada, no menos. De modo que, a priori, el problema residía en `flyingKinect` o en Gazebo.

El plugin `flyingKinect` está dividido fundamentalmente en dos: un módulo dedicado a la posición y otro a las cámaras. Gazebo los actualiza a frecuencias distintas a través de

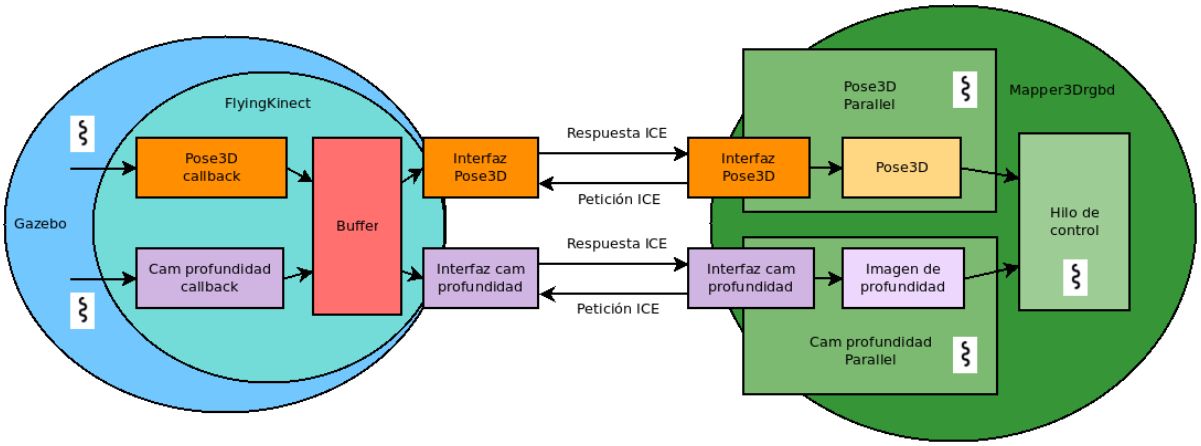


Figura 4.10: Diagrama de bloques del entorno simulado.

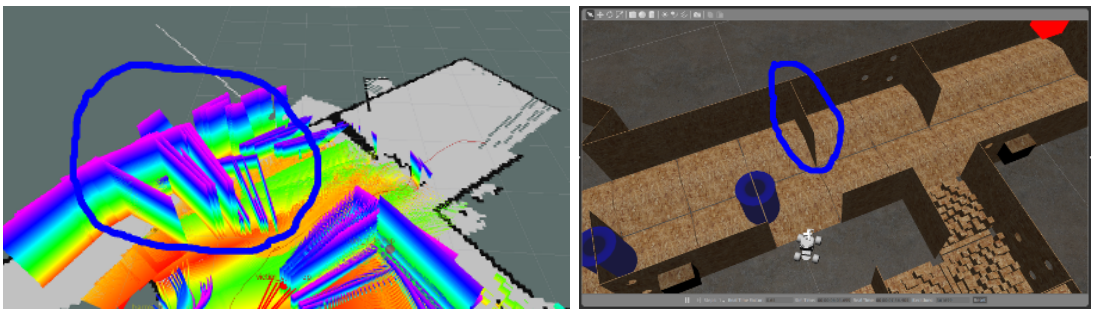
callbacks. Las frecuencias son configurables desde el archivo de configuración del mundo que se utilice, en este caso el apartamento del mundo `GrannyAnnie`. La posición del sensor la actualiza con el motor de físicas del mundo. Si se pone un valor en la frecuencia del motor de físicas del mundo y el ordenador sigue teniendo recursos, Gazebo cumplirá con esa velocidad. No obstante, este simulador no asegura la frecuencia de actualización de las cámaras. Lo hará en la medida de lo posible y siempre será inferior a la frecuencia de actualización del motor de físicas, y por tanto, de la posición.

Se aumentó la frecuencia de actualización del motor de físicas del mundo a 1000 Hz, de modo que la posición del sensor se refrescaba cada milisegundo, y la frecuencia de actualización de la cámara se estableció en 30 Hz. Se implementó el buffer que se puede ver en la figura 4.10 dentro de `flyingKinect`. Su función principal es que las imágenes y las posiciones no tengan desfase debido a la estructura de hilos del plugin. Cada vez que Gazebo llama al *callback* de la cámara, se guarda en él la imagen de profundidad recién actualizada y una estampa de tiempo del momento actual. En ese momento, también se guarda la última actualización de la posición con la estampa de tiempo de cuándo se actualizó. Debido a que se ha establecido en 1000 Hz la frecuencia de actualización de la posición, las imágenes de profundidad y la posición estarán desfasadas en el buffer un máximo de 1 milisegundo. También se añadió una estampa de tiempo en la información de la posición que viaja al mapeador para compararla en recepción con la de la imagen, que ya tenía una en su estructura de datos.

Por un lado, comparando las estampas de tiempo de la posición y la imagen de

profundidad que llegaban al mapeador se observó que, en ocasiones, la estampa de tiempo de la imagen era de un momento anterior a la de la localización. Esto se debe a que, a veces, las imágenes llegan más tarde porque son datos más voluminosos que los de posición y tardan más en enviarse. De modo que una de las causas que originan el desfase es la diferencia de tamaño del dato que se envía a través de ICE, que ocasiona que el que pesa más (imagen de profundidad) tarde más tiempo en llegar que el que pesa menos (posición).

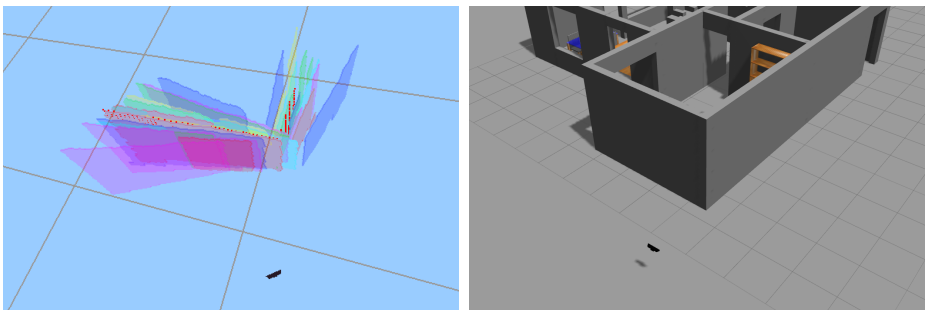
Por otro lado, algunas de las parejas de imagen y localización que tenían la misma estampa temporal estaban desfasadas también en cuanto a la información que contenían, de modo que tenía que haber otra causa que originase el desfase y debía de estar en el lugar más improbable, en Gazebo. Buscando en las incidencias del BitBucket (un servicio de control de versiones web) del proyecto de Gazebo se encontró una [3] en la que tenían caracterizado este fallo pero que todavía no tenían solucionado. En la figura 4.11 se pueden observar dos imágenes que subió la persona que creó la incidencia. En la primera, se ve la reconstrucción en tres dimensiones del muro que se aprecia en la de la derecha, hecha mediante la rotación continua de la cámara. Esas falsas medidas son fruto del desfase y es justo lo que pasa en este caso, como podemos ver en la reconstrucción resultante en el mapeador tras realizar el mismo test en la figura 4.12. Aprovechando, escribí en los comentarios de la incidencia dándoles más información sobre el error y las conclusiones sacadas de este estudio del sincronismo. Nathan Koenig, uno de los creadores de Gazebo, propuso una solución que probé descargándome, modificando y compilando el código fuente de Gazebo, pero no funcionó.



(a) (b)

Figura 4.11: Reconstrucción errónea por fallo en Gazebo.

[3] `https://bitbucket.org/osrf/gazebo/issues/1748`



(a) Muro reconstruido con fallos. (b) Muro a reconstruir.

Figura 4.12: Reconstrucción errónea por fallo en Gazebo.

En resumen, tras el estudio del sincronismo hecho del sistema, se han detectado dos motivos independientes que causan que la posición llegue en ocasiones antes que la imagen de profundidad y haya extracciones erróneas:

- La imagen de profundidad tarda más tiempo en llegar, por ser un dato mucho más voluminoso, que la posición.
- Gazebo tiene un fallo que causa que la cámara a color y de profundidad sean actualizadas un poco más tarde que la posición.

4.3.2. Soluciones

Una de las causas principales del desfase de la información se debe a un fallo del simulador Gazebo y solucionarlo sale fuera de los objetivos. Sin embargo, la otra causa es consecuencia de la comunicación entre el plugin `flyingKinect` y el componente `mapper3Drgbd`, y sí cae dentro de los objetivos abordables.

Para solucionar el desfase causado por que un dato sea más pesado que el otro, se ha hecho uso de la estampa de tiempo añadida en la interfaz Pose3D. Se ha implementado un filtro en el mapeador que compara la diferencia de las estampas temporales con cierto umbral. Este umbral depende de la frecuencia de actualización de los *callback* del plugin `flyingKinect` que, como vimos en la anterior subsección, eran frecuencias distintas para la cámara y la posición, y además la cámara siempre se actualizaba a un ritmo igual o inferior. Debido a esto, es la frecuencia de actualización del motor de físicas del mundo que se está ejecutando en Gazebo el valor que se añade en un nuevo campo en el archivo de

configuración de la aplicación mapper3Drgbd. A partir de este valor se genera el umbral, en milisegundos, mediante la ecuación:

$$Umbral[ms] = \frac{1000}{FrecuenciaGazebo[Hz]} \tag{4.2}$$

Si la diferencia de las estampas de tiempo de la imagen de profundidad y la pose3D es mayor que ese umbral de tiempo, se descartan esas dos medidas evitando así el desfase debido al mayor tiempo de envío de las imágenes.

Llegados a este punto, solo se pueden extraer planos exactos con el algoritmo de extracción activo todo el rato si el movimiento continuo es muy lento, para que el desfase introducido por el fallo de Gazebo no se note. Aunque con movimiento continuo no se pueda lograr una extracción fluida, sí que se puede lograr mediante movimientos simples. Un movimiento simple sería mover el sensor hacia delante una distancia considerable de golpe, pero solo una cada vez y esperando un tiempo breve hasta el próximo movimiento. De este modo, se pueden obtener reconstrucciones de entornos simulados de forma rápida y sin fallos.

Para conseguir una reconstrucción fluida mediante movimientos simples se ha implementado otro filtro en el mapeador, que compara la imagen de profundidad y posición de cada momento con la del momento anterior. Filtra aquellas parejas en las que encuentra que solo uno de los datos del sensor simulado ha cambiado, dejando pasar el resto.

4.4. Integración en entorno real

En esta sección se detallan todas labores realizadas para integrar los componentes `mapper3Drgbd` y `cam_autoloc` en un sistema que realice mapas en tres dimensiones de parches planos a partir de un sensor RGB-D real en movimiento. Los datos reales se obtienen a través del dispositivo Asus Xtion PRO LIVE conectado al driver de JdeRobot `openniServer`.

El primer paso para conseguir que funcione el sistema en entornos reales fue modificar la aplicación de autolocalización existente para que pueda enviar al mapeador la posición y orientación estimadas.

4.4.1. Modificando la aplicación cam_autoloc

Esta aplicación no contaba con un modo de proporcionar a otros componentes la estimación de la localización del sensor que obtiene, de modo que la primera labor realizada ha sido añadirle una capa de comunicaciones. Se le ha añadido un servidor ICE y una clase llamada Pose3DI, siguiendo el convenio de nombres de ICE, que implementa la interfaz de JdeRobot Pose3D.

Para que el hilo de comunicaciones del componente cam_autoloc pueda obtener la posición y orientación estimadas en el hilo principal de la aplicación, se ha creado un área de memoria compartida. Cada vez que la localización se estima, se guarda en este área, y cuando le llega alguna petición a cam_autoloc para enviar la localización, la coge de ahí.

No obstante, esta aplicación para definir la orientación utiliza ángulos de rotaciones en los ejes, mientras que la interfaz de la Pose3D de JdeRobot emplea cuaterniones de rotación. Por un lado, estos ángulos constituyen un conjunto de tres coordenadas angulares que definen la orientación de un objeto en el mundo mediante una secuencia de rotación de los mismos. El orden de rotaciones sobre los ejes no es conmutativo, por lo que se ha de especificar. El signo del ángulo se determina mediante la regla de la mano derecha, donde el pulgar apunta hacia el sentido del vector del eje y el resto de dedos indican el sentido de giro positivo, figura 4.13. Por otro, los cuaterniones de rotación también proporcionan una notación para definir la orientación de un objeto, mediante cuatro componentes complejas. Son más simples que los ángulos de rotación y es más eficiente realizar operaciones con ellos.

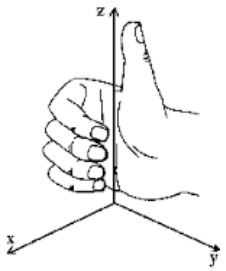


Figura 4.13: Regla de la mano derecha.

Para realizar la conversión correcta de ángulos de rotación a cuaternión, lo primero ha sido conocer el orden de giro de los ejes de la aplicación, que es XYZ. Con esta información

ya se puede construir la matriz de rotación empleando la librería Eigen y pasar de ahí a cuaternión con una función de la misma.

El comportamiento del sistema implementado queda reflejado en la figura 4.14. Las flechas de color negro son los eventos que ocurren constantemente y las de color verde los que suceden tras una llamada ICE por parte de un cliente.

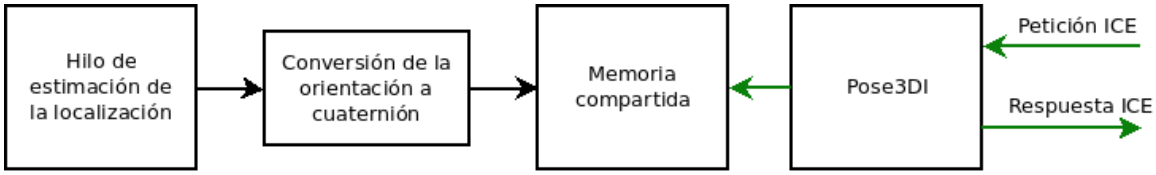


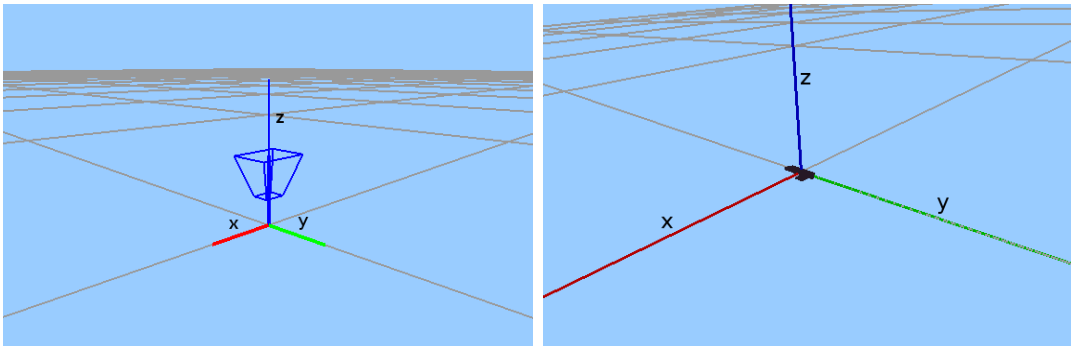
Figura 4.14: Diagrama de bloques de las modificaciones.

4.4.2. Encaje de sistemas de referencia 3D

El sistema de referencia 3D que utiliza el componente `cam_autoloc` no es exactamente igual al que utiliza el `mapper3Drgbd`, de modo que los puntos que se representan en ellos no son calculados en las mismas posiciones.

Los ejes XYZ sí que están colocados de la misma manera como se puede ver en la figura 4.15. El giro de los ejes del mundo del mapeador para aplicar orientaciones es también XYZ, pero la posición inicial de la que parte para aplicarlos (rotación 0º para los tres ejes), es distinta que la que tiene el mundo del autolocalizador. En el mundo de la aplicación `cam_autoloc` está apuntando hacia el eje Z, figura 4.15(a), y en el del mapeador hacia el eje X, figura 4.15(b). Esto hacía que el movimiento de rotación del sensor en el mapeador no fuese análogo a la realidad y las nubes de puntos no se generaban en el lugar correcto.

La solución que se ha elegido en este caso es aplicar ciertas rotaciones fijas en el mundo del mapeador para que el estado inicial sea igual que el del autolocalizador. Las rotaciones en el mapper3Drgbd son XYZ también, por lo que rotando en el eje x 0º, en el y -90º y en el z 90º, estaría en la misma orientación para aplicar las rotaciones que le envía. Esto se ha implementado generando la matriz de rotación y traslación (RT) para estos giros, mediante la librería Eigen, y multiplicándola por la matriz RT que ya se utilizaba para convertir la imagen de profundidad (2D) a nube de puntos (3D).



(a) Mundo de `cam_autoloc`. (b) Mundo de `mapper3Drgbd`.

Figura 4.15: Rotaciones 0º para los tres ejes.

4.4.3. Funcionalidades para mapeo de entornos reales

Llegados a este punto, el mapeador ya es capaz de extraer planos satisfactoriamente a partir de datos reales. En la figura 4.16 se puede ver la aplicación `cam_autoloc` generando la estimación de localización del sensor RGB-D y la aplicación `mapper3Drgbd` utilizando esa estimación, junto con las imágenes de profundidad del sensor, para construir el mapa en tres dimensiones del entorno. No obstante, hay una serie de funcionalidades y mejoras que hay añadir que únicamente suceden en un escenario real de estas características.

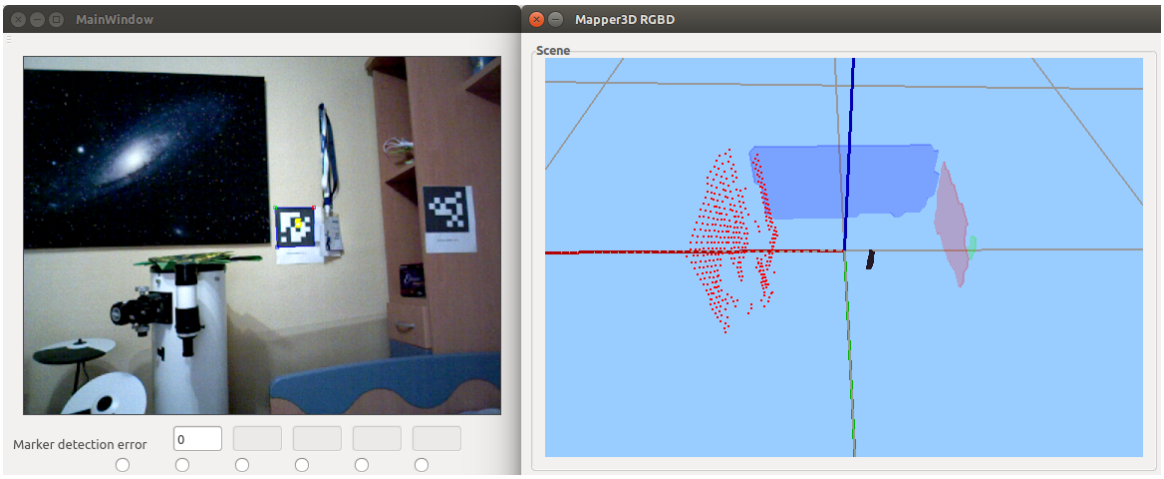


Figura 4.16: Mapper3Drgbd funcionando con datos reales.

En primer lugar, se ha estudiado el sincronismo que tienen los datos de profundidad y de posición que llegan al mapeador. Por un lado, la posición se obtiene en el componente `cam_autoloc` a partir de la imagen a color que pide a `openniServer`, y luego, esa posición

estimada, se le envía al mapeador cuando este la pida. Por otro, la imagen de profundidad viaja directamente desde `openniServer` hasta el mapeador. Por este motivo, los datos que llegan al mapeador no estarán completamente sincronizados debido al paso intermedio del componente de autolocalización. Sin embargo, se ha observado que el retardo introducido no es apreciable para un movimiento suave del sensor, de modo que no ha hecho falta tomar medidas como tener en cuenta las estampas de tiempo.

En segundo lugar, se ha comprobado la carga del sistema cuando están ejecutándose simultáneamente todas las partes de él. Por un lado, el componente `cam_autoloc` tarda en media unos 150 milisegundos en procesar cada fotograma y obtener una posición de él, por lo que su frecuencia se ha establecido en 6 Hz. También se ha comprobado que las modificaciones introducidas en él no han añadido una carga extra a la CPU apreciable. Por otro, el componente `mapper3Drgbd` tarda unos 100 milisegundos en realizar cada iteración de su algoritmo pero, como la posición solo se refresca con un ritmo de 6 Hz, esta es la frecuencia que se le ha establecido para hacer la prueba de carga. Con estos componentes arrancados junto con `openniServer`, refrescando las imágenes a 20 Hz, y el algoritmo de extracción activo, la carga del procesador, cuyas características se definieron en la sección 3.1, es del 70 %. Es una carga algo elevada pero que no llega a colapsar el sistema y permite que funcione perfectamente en tiempo real.

En tercer lugar, se ha percibido que la extracción de planos no tiene por qué funcionar a tanta frecuencia ya que el movimiento del sensor es suave. Por este motivo, se ha añadido un nuevo campo en el archivo de configuración que define cada cuánto tiempo se extraen los planos mientras que el cálculo de la nube de puntos se sigue haciendo en el resto de iteraciones del ciclo. De este modo se puede ver la nube de puntos cambiar en el mundo del mapeador a, por ejemplo 6 Hz, mientras que solo se extraen planos a 1 Hz. Esta modificación es aplicable para todas las fuentes de datos, sintética, simulada o real. De este modo, la extracción de planos no es tan exhaustiva para la CPU y se puede combinar mejor con la ejecución de otros componentes. Bajando la extracción de planos de 6 Hz a 1 Hz se consigue que la carga de la CPU baje del 70 % al 60 % .

Cuarto, se ha utilizado el campo del fichero de configuración del mapeador, que determina si los datos proceden de una fuente sintética o simulada, para definir también si provienen de una fuente real, ya que el resto de medidas que se exponen solo afectan a entornos reales.

Quinto, se ha hecho un ajuste de los parámetros que utiliza el algoritmo de extracción de planos, configurables desde la interfaz del mapeador, para optimizarlos para escenarios reales. Para ello se ha utilizado el generador sintético de puntos. Los parámetros que se han modificado son: distancia para fusión normal de planos, distancia para fusión lateral de planos y número mínimo de puntos para extraer un plano. En el capítulo 5 de experimentos se verá de forma detallada el procedimiento.

En sexto lugar, se ha incluido en la Pose3D, que viaja desde `cam_autoloc` hasta el `mapper3Drgbd`, información sobre si hay balizas en el campo de visión o no. En este escenario, dado que la estimación de la localización depende de si se están viendo balizas visuales o no, puede darse el caso de que en algunos momentos no se vean y no exista una estimación de la localización del sensor y por lo tanto no se deban extraer planos. De este modo, el mapeador puede descartar la información de localización e imágenes de profundidad que le llega cuando no las hay.

Y séptimo, se le ha añadido otra modificación más al mapeador derivada de encontrar o no balizas visuales. En la aplicación autolocalizadora, cuando se encuentra una baliza que está lejos de la última, la estimación de la localización en los primeros instantes no es exacta, ya que utiliza filtros temporales para tener en cuenta la estimación de pasados momentos y que la estimación no tenga cambios abruptos por fallos puntuales. Tras perder la visión de balizas, no podrá estimar la localización y el filtro temporal utilizará las últimas estimaciones calculadas. Por un lado esto mejora la estimación si se pierde de vista el marcador o se calcula de forma errónea por problemas de calidad de imagen en un par de fotogramas. Por otro, hace que si se pierde de vista un marcador y se salta a otro que esté lejos (porque hemos tapado el sensor y hecho que apunte a otra pared, por ejemplo), la estimación tarde un tiempo en ser exacta, hasta que en el filtro temporal solo se usen imágenes del actual marcador. Por este motivo, si la extracción de planos ha estado parada porque no se encontraban marcadores y se vuelve a encontrar uno, la extracción espera cierto tiempo hasta volverse a iniciar. Esta cantidad de tiempo es configurable en el archivo de configuración de la aplicación, y se ha establecido en dos segundos por ser un valor suficiente.

Con todo este sistema funcionando, en el siguiente capítulo se validará su correcto funcionamiento mediante experimentos.

Capítulo 5

Experimentos

En este capítulo se expondrán las pruebas que se han hecho con el fin tanto de validar el trabajo realizado, como de mostrar los resultados obtenidos. Para empezar, se dará una visión de los distintos entornos en los que se han realizado las pruebas. Después, se realizará un ajuste de parámetros para el algoritmo de extracción de planos en entornos reales. Para finalizar, se expondrán diversas pruebas organizadas en distintas localizaciones, tanto en simulación como en un entorno real, que mostrarán los resultados finales.

5.1. Entornos de pruebas

En esta sección se detallan los distintos entornos en los que se han realizado las pruebas y las peculiaridades en cada uno de ellos. También se expondrán los componentes de JdeRobot involucrados.

5.1.1. Simulación

Las pruebas en simulación se han realizado con el componente `mapper3Drgbd` mejorado conectado directamente al plugin `flyingKinect` (mejorado también) corriendo en el simulador Gazebo mientras es teleoperado con el componente `navigatorCamera`. Este esquema de conexiones se puede ver en la figura 4.7 del capítulo anterior.

El objetivo de las pruebas con datos simulados es validar la fluidez de la extracción de planos realizando mapas de distintas zonas del apartamento en el menor tiempo posible. Para ello, se moverá el sensor a voluntad utilizando el componente `navigatorCamera`.

El entorno de simulación elegido para las pruebas ha sido el apartamento del mundo GrannyAnnie del campeonato RoCKin [1]. En la figura 5.1 se puede ver la vista aérea del apartamento en el simulador Gazebo. Se puede ver que tiene las habitaciones y muebles típicos de una vivienda como dormitorio, sala de estar y cocina. Posee además una habitación pequeña vacía que servirá para realizar algunas pruebas específicas.

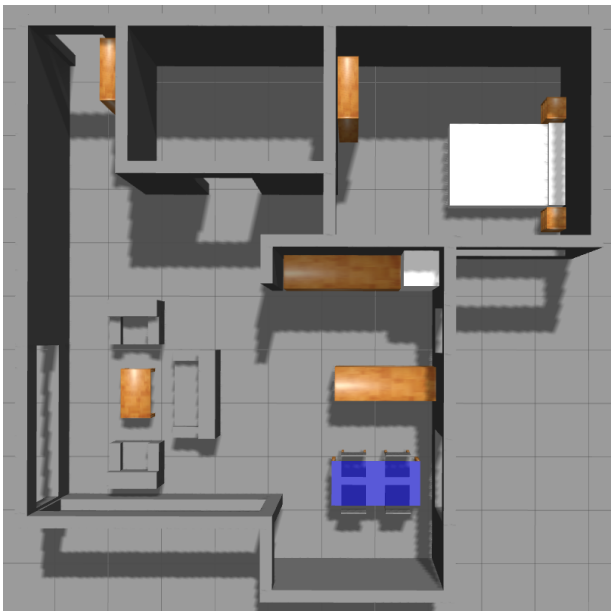


Figura 5.1: Vista aérea del apartamento GrannyAnnie en Gazebo.

5.1.2. Datos enlatados

Los datos enlatados son una gran manera de probar los cambios que ha habido en un sistema ya que permiten utilizar exactamente la misma entrada para observar las diferencias a la salida cuando se emplean algoritmos diferentes. De este modo, se aísla el objeto de estudio, uno de los principios básicos del método científico.

Para obtener los datos enlatados se ha utilizado el componente `recorder` de JdeRobot con el esquema de conexiones que se puede ver en la figura 5.2. Por un lado, se arranca el plugin `flyingKinect` en Gazebo y el componente `navigatorCamera` para poder teleoperar el sensor simulado. Por otro, se conecta `recorder` a las interfaces de Pose3D y cámara de profundidad del plugin y se empieza a mover el sensor. Para reproducirlos se ha utilizado el componente `replayer`, que toma como entrada los datos enlatados y los expone en

[1] `http://rockinrobotchallenge.eu/home.php`

interfaces de Pose3D y cámara de profundidad para que el componente `mapper3Drgbd` se conecte a ellos, como muestra el esquema 5.3.

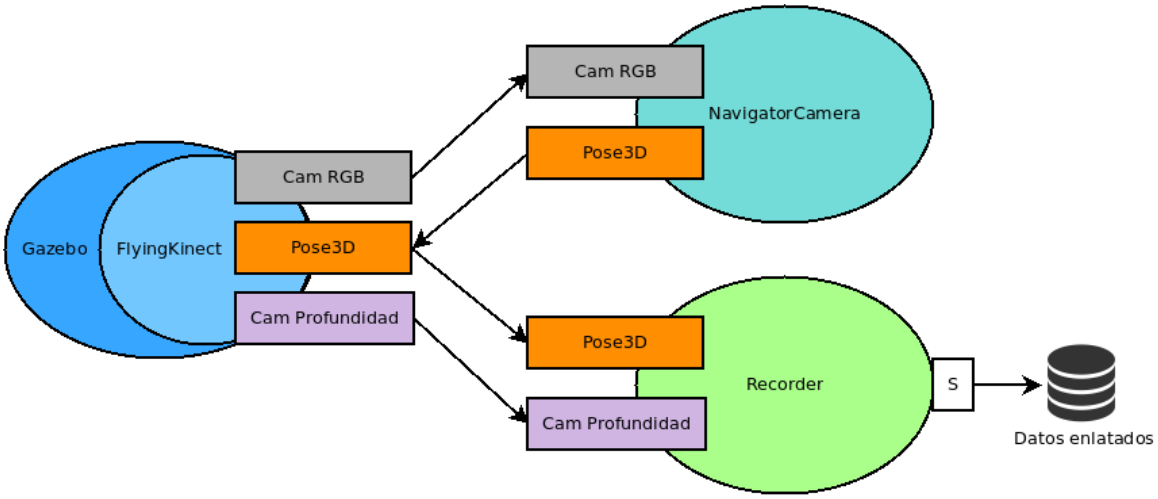


Figura 5.2: Esquema de conexiones para obtener los datos enlatados.

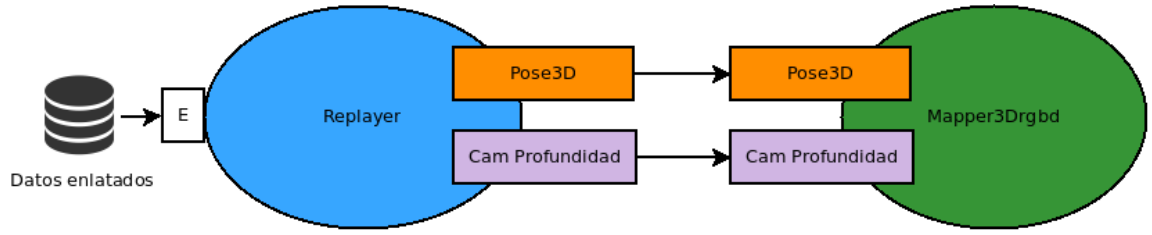


Figura 5.3: Esquema de conexiones para utilizar los datos enlatados.

El lugar donde se han grabado los datos ha sido, nuevamente, el apartamento del mundo GrannyAnnie del campeonato RoCKin.

El plugin `flyingKinect` también se modificó para la inserción de las estampas de tiempo en la información de la posición y la mejora del buffer para la exactitud de estas. No obstante, el mapeador antiguo no utiliza la ventaja de estas modificaciones en `flyingKinect`, así que los datos se grabarán utilizando el plugin modificado en este trabajo sin que esto afecte al funcionamiento del mapeador sin modificar.

5.1.3. Datos sintéticos

En estas pruebas, se ha empleado el componente generador sintético de nubes de puntos, `pointCloudGenerator`, desarrollado en este trabajo, y la aplicación `mapper3Drgbd`.

Las pruebas en este entorno tienen un doble objetivo: validar el correcto funcionamiento del generador sintético al mismo tiempo que se ajustan algunos de los parámetros del algoritmo de extracción de planos con el fin de que funcione mejor en entornos reales, ya que los datos no son tan exactos como en simulación.

5.1.4. Entorno real

Las pruebas en el entorno real se han realizado con el sistema final, es decir, con el componente `mapper3Drgbd` recibiendo imágenes de profundidad del sensor real por medio del driver `openniServer` y la localización estimada por el componente `cam_autoloc`. El esquema de conexiones puede verse en la figura 4.1 del capítulo anterior. El objetivo principal es validar que la extracción de mapas del entorno se realiza de forma fluida mientras el sensor tiene un movimiento suave y constante.

Para realizar las pruebas, se han puesto 9 marcadores visuales distintos en el entorno. En las dos imágenes de la figura 5.4 se pueden ver las balizas en las paredes. Para ello ha hecho falta establecer un origen de coordenadas en la habitación, medido la localización respecto de ese origen de cada uno y añadido esa información al fichero de balizas para que la aplicación `cam_autoloc` sea capaz de proporcionar la estimación de la localización.

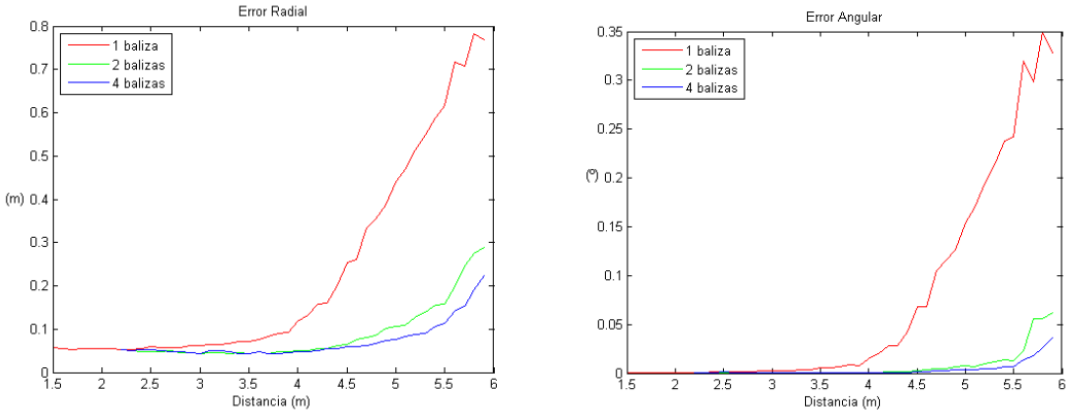
(a) (b)

Figura 5.4: Entorno con marcadores visuales en el que se han hecho las pruebas.

5.2. Ajuste de parámetros del mapeador para el entorno real

En esta sección se ajustarán los parámetros del algoritmo de construir mapas del componente `mapper3Drgbd` para que funcione de manera óptima en entornos reales. No se realiza un ajuste de parámetros para el entorno simulado puesto que eso ya lo realizó Juan Navarro [10] cuando desarrolló el componente mapeador original.

Es importante caracterizar el error que tiene el componente `cam_autoloc` al estimar la localización del sensor para elegir los parámetros de forma correcta. Para ello, se ha hecho uso del estudio de precisión realizado en simulación que hizo Alberto López-Cerón [11] al desarrollar el componente. En la figura 5.5 se puede observar el error expresado en metros frente a la distancia en función del número de balizas encontradas en el momento. El error radial para distancias pequeñas, menos de 3 metros, es de unos 6 centímetros. En entornos reales habrá más error que en el simulado, de modo que se dará un margen de unos 20 centímetros a la hora de considerar dos parches planos como el mismo o distintos. Esto es aplicable a la fusión normal y lateral de planos como veremos en las dos pruebas a continuación.



(a) Error radial.

(b) Error angular.

Figura 5.5: Error frente a distancia según el número de balizas.

Para comprobar que la fusión normal funciona correctamente con el nuevo valor se han generado dos nubes de puntos con el generador sintético separadas 19 centímetros, como

se puede ver en la figura 5.6(a). En la figura 5.6(b) se pueden observar las dos nubes de puntos en el componente `mapper3Drgbd`. Con el parámetro de fusión normal establecido en 18 centímetros se extraen los dos parches planos que se puede ver en la figura 5.6(b) pero no se fusionan, ya que no están lo suficientemente cerca. En cambio, si se cambia este parámetro a 20 centímetros, estos dos planos se extraen y se fusionan en uno solo, como se puede ver en la figura 5.6(d). Con este ejemplo, además, se ha validado la correcta creación de dos nubes de puntos en un mismo volumen.

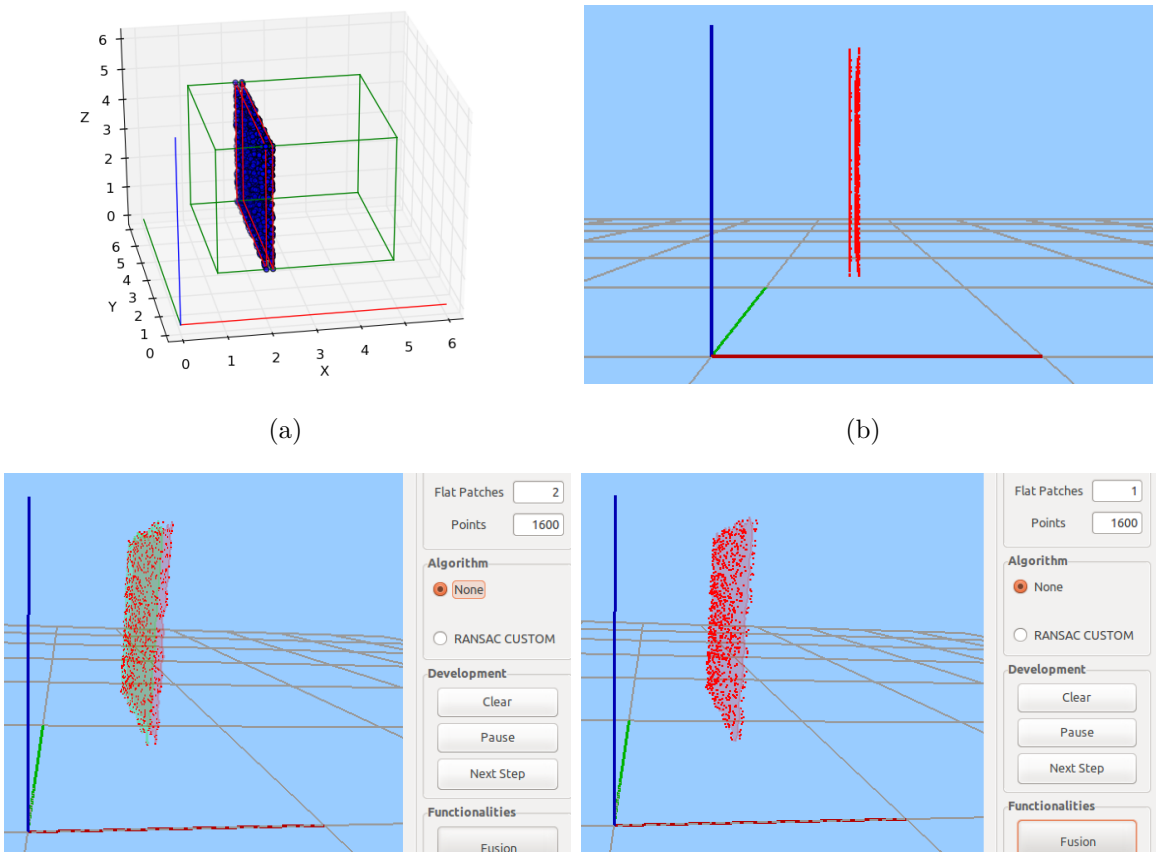
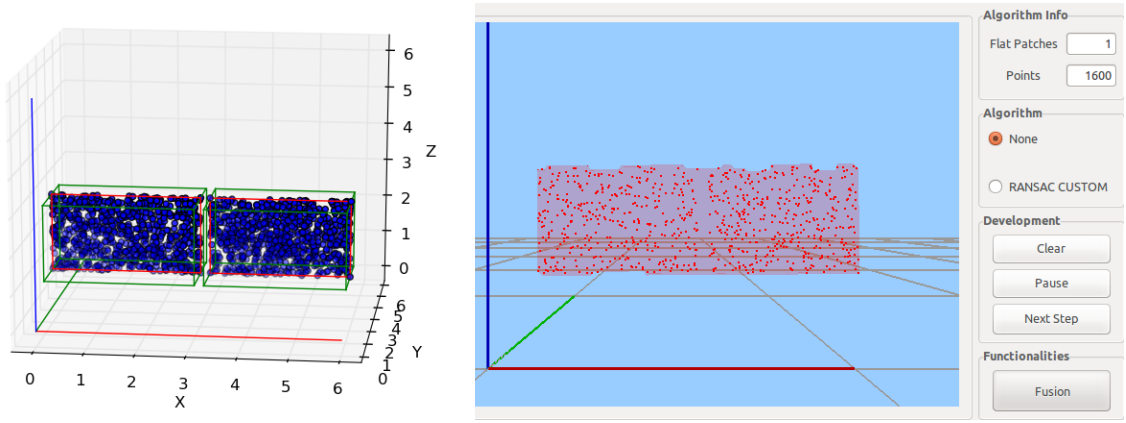


Figura 5.6: Varias nubes de puntos en el mismo volumen y fusión normal de parches planos.

Otro parámetro a ajustar es la fusión lateral entre parches, a la vez que se comprueba la generación de distintas nubes de puntos en distintos volúmenes. En la figura 5.7(a) se pueden ver dos nubes de puntos generadas en distintos volúmenes de trabajo separadas lateralmente 19 cm entre sí. En la figura 5.7(b) se pueden observar esas dos nubes de puntos en el mapeador así como un único parche sacado de ambas debido a que el parámetro de

la fusión lateral está definido en 20 centímetros.



(a) (b)

Figura 5.7: Nubes de puntos en distintos volúmenes y fusión lateral de parches planos.

Además de los parámetros comentados, fusión lateral y normal de planos, se ha modificado el número de puntos necesario para crear un parche plano debido a que en el entorno hay muchos objetos, como cosas en la estantería, que no interesa que sean mapeados. En estos experimentos se ha subido a 40 puntos necesarios para extraer un parche plano.

En las gráficas del error también se puede ver que, cuantas más balizas están en el campo visual de la cámara, menor es el error en función de la distancia. Por ello, en la prueba final del entorno real, se verán varias balizas a la vez la mayor parte del tiempo.

5.3. Prueba de estrés en la habitación pequeña del apartamento simulado

En este experimento se utilizan datos enlatados, que consisten en este caso en rotaciones del sensor durante 1 minuto para comprobar si se extrae algún plano erróneo con las modificaciones realizadas. También se utilizarán los mismos datos como entrada al mapeador original, para observar las diferencias.

Comienza con el sensor situado en el centro de la habitación cuadrada que se ve en la figura 5.8. Se han empleado rotaciones simples, que consisten en rotar el sensor una cantidad considerable, 30º en este experimento, a ritmo de una vez por segundo y el

algoritmo de extracción ha estado activo todo el rato. El sensor realizará rotaciones a la izquierda durante 30 segundos y rotaciones a la derecha durante otros 30 segundos. El objetivo de dar tantas vueltas es probar si con las modificaciones realizadas ocurre alguna falsa extracción y ver cómo se comportaba el antiguo mapeador para comparar la salida. El algoritmo de extracción en este caso se ha configurado a 6 Hz.

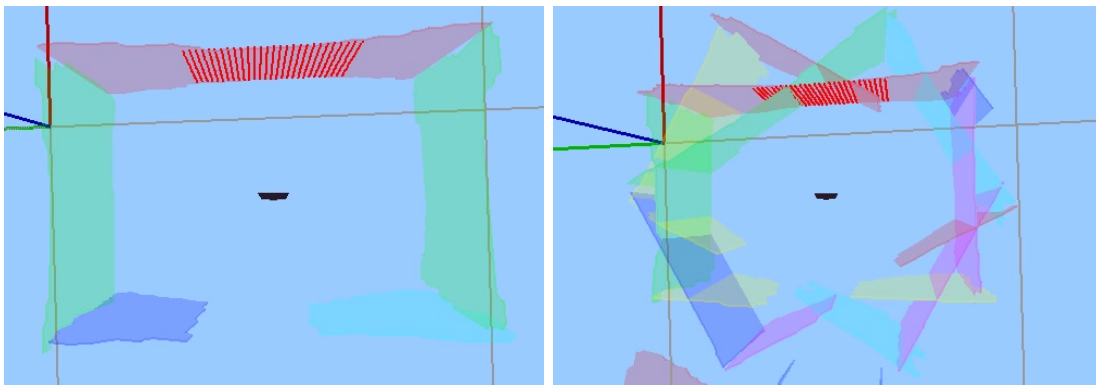
Figura 5.8: Posición inicial del sensor.

En la figura 5.9(a) se muestra el mapa generado con los cambios realizados al mapeador, mientras que en la figura 5.9(b) se puede ver el resultado obtenido con el mapeador sin modificar. Se puede observar que los planos generados por el mapeador mejorado corresponden a las paredes de la habitación y que no ha habido ni una extracción errónea en el minuto que ha durado la prueba. En cambio, en el mapeador original, ha habido múltiples extracciones de falsos planos frutos del desfase entre posición e imagen de profundidad. Ambos mapeados están disponibles en la sección de tests de la wiki de este trabajo [2].

5.4. Prueba de fluidez en la habitación pequeña del apartamento simulado

En esta prueba se valida la fluidez en la extracción de los planos correspondientes a todas las paredes de la habitación. Se ha teleoperado continuamente el sensor simulado para moverlo en este caso.

[2] `http://jderobot.org/Samartin-tfg#Rotation_test_in_square_room`



(a) `Mapper3Drgbd` mejorado. (b) `Mapper3Drgbd` original.

Figura 5.9: Resultado del experimento en la habitación cuadrada.

Se sitúa el sensor en el centro de la habitación cuadrada que se ve en la figura 5.10(a) y se realizan 11 rotaciones de 30º hacia la izquierda, estando las rotaciones separadas temporalmente aproximadamente medio segundo, con lo que todo el proceso solo dura 5 segundos y medio. El algoritmo de extracción ha estado funcionando durante este test a 6 Hz.

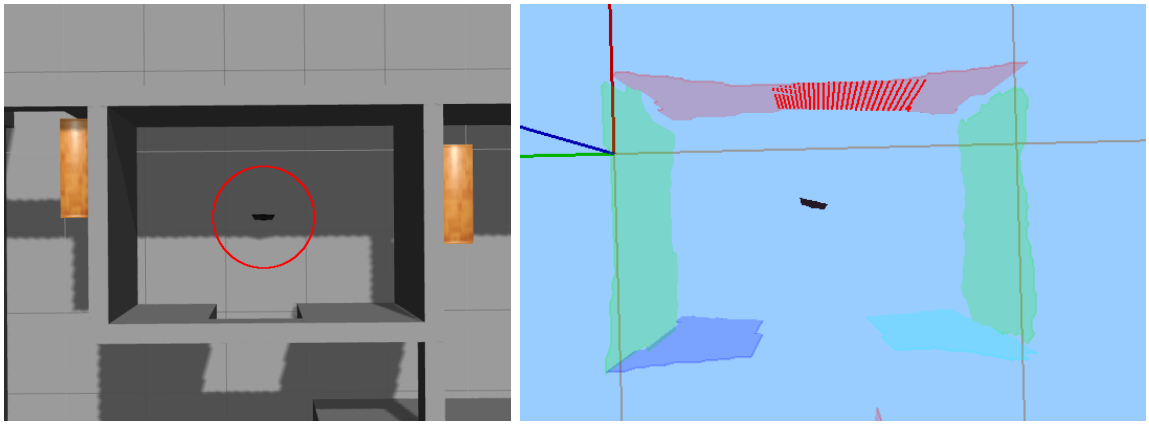
El mapa de parches planos extraído de esta rotación se puede ver en la figura 5.10(b). En esta figura se puede observar que en menos de 6 segundos se ha conseguido hacer un mapa tridimensional de los mapas de la habitación sin que haya ninguna extracción errónea. El vídeo de este experimento está disponible en la página web de la sección de tests de la wiki de este trabajo [3].

5.5. Prueba de estrés en la sala de estar del apartamento simulado

Para realizar este experimento se han empleado datos enlatados que consisten en movimientos hacia delante y hacia atrás del sensor durante 40 segundos. Su objetivo es validar que no se extrae ningún plano de forma errónea con las modificaciones implementadas en el mapeador. También se han utilizado estos datos enlatados para comparar el mapa de planos generado con el del mapeador original.

En este experimento el sensor comienza situado en la sala de estar mirando hacia

[3] `http://jderobot.org/Samartin-tfg#Room_mapping_test`



(a) Posición inicial. (b) Parches planos generados.

Figura 5.10: Experimento con datos simulados en la habitación.

el comedor, como se puede ver en la figura 5.11. Se han realizado sucesiones de nueve movimientos simples hacia delante y nueve hacia atrás en la dirección de las flechas que se ven en la figura, a ritmo de un movimiento por segundo durante 40 segundos. En cada movimiento se ha desplazado el sensor medio metro. El algoritmo de extracción ha funcionado a 6 Hz en esta prueba.

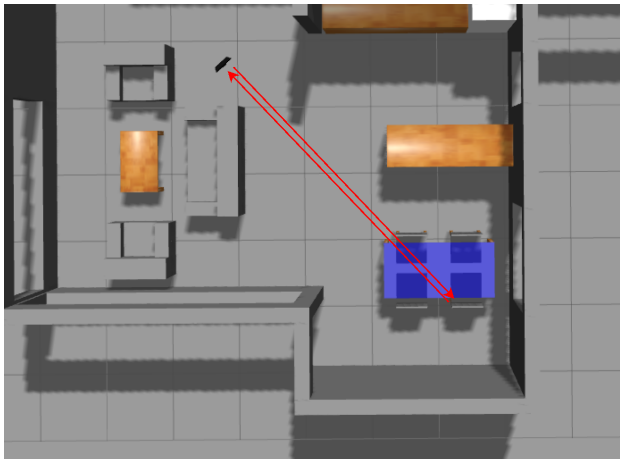
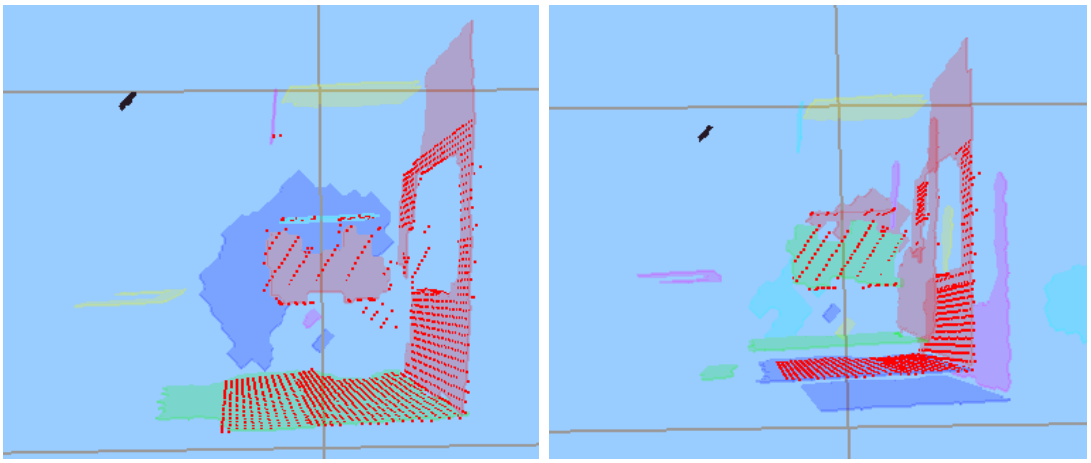


Figura 5.11: Posición inicial y trayecto del sensor.

En la figura 5.12(a) se puede ver el resultado del mapeador mejorado y en la figura 5.12(b) la salida del mapeador original. En el primer mapa se han extraído un total de 10 parches planos mientras que en el segundo han sido 22. Se puede observar que en la salida del mapeador mejorado no hay ningún parche erróneo, mientras que en la del mapeador original, se ven varios parches erróneos al lado del parche que corresponde a la esquina de

la pared. Los vídeos de ambas pruebas están disponibles en la sección de tests de la wiki de este trabajo [4].



(a) `Mapper3Drgbd` mejorado. (b) `Mapper3Drgbd` original.

Figura 5.12: Resultado del experimento en la sala principal.

5.6. Mapeado de todo el apartamento simulado

En esta prueba se ha teleoperado continuamente el sensor para comprobar que las modificaciones realizadas en el mapeador y en el plugin `flyingKinect` permiten generar el mapa de parches planos de un apartamento entero de forma fluida.

Se ha hecho el recorrido por el apartamento, que se puede ver en la figura 5.13(a), para obtener un mapeado de todas sus paredes. Se han realizado movimientos de medio metro y rotaciones de 30º a un ritmo de dos movimientos o rotaciones por segundo. La frecuencia del algoritmo de extracción en esta prueba ha sido 6 Hz.

El conjunto de parches planos extraídos, un total de 52, se puede ver en la figura 5.13(b). El recorrido para mapear todo el apartamento dura unos 50 segundos. Como se puede apreciar, no hay ninguna extracción errónea y se ha realizado un mapa de todo el apartamento. Los parches que se pueden ver fuera del apartamento corresponden al suelo que se veía a través de las ventanas y puertas de las paredes. Además de los contornos de las paredes y de parte del suelo, se han generado parches planos de muebles, como la

[4] `http://jderobot.org/Samartin-tfg#Movement_test_in_living_room`

mesa del comedor o parte de la cama. El vídeo de esta reconstrucción se puede ver en la wiki [5].

(a) Recorrido realizado en el apartamento. (b) Parches planos generados.

Figura 5.13: Reconstrucción del apartamento.

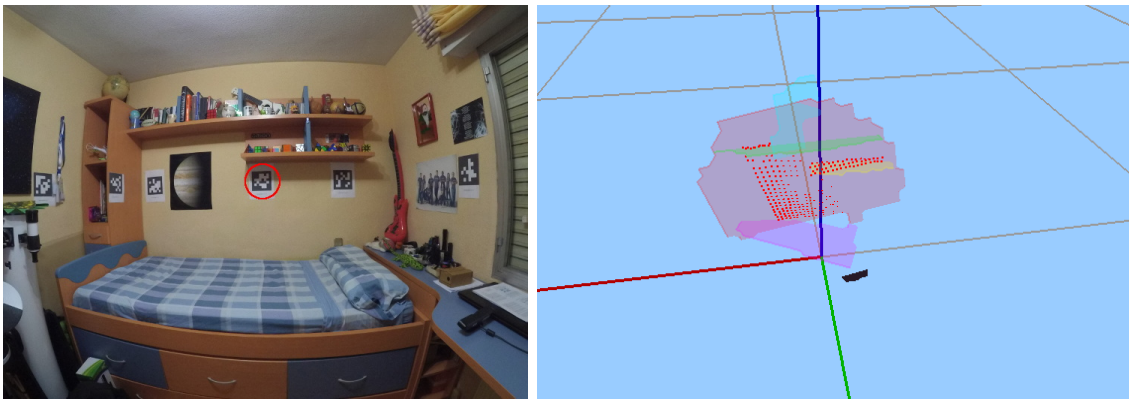
5.7. Prueba en entorno real con un solo marcador visual

En esta prueba se ha empleado el marcador visual que se ve en la figura 5.14(a). El objetivo ha sido comprobar que, a distintas distancias y ángulos, la nube de puntos se genera en el lugar correcto y no hay extracciones de planos erróneas. El sensor ha estado moviéndose de forma continua hacia el marcador y luego alejándose, para después realizar una serie de rotaciones, siempre mirando hacia el marcador. El algoritmo de extracción ha estado funcionando a 1 Hz durante 60 segundos.

Los parches planos generados se pueden observar en la figura 5.14(b). La nube de puntos se ha generado en el mismo lugar ya fuera la distancia al marcador pequeña o grande y también ha respondido bien el sistema ante rotaciones de la cámara. No ha habido ninguna extracción de parches errónea. El parche de color naranja corresponde a la balda pequeña de la estantería, el verde a la larga, el azul celeste a una serie de libros

[5] `http://jderobot.org/Samartin-tfg#Apartment_mapping_test`

apoyados sobre esta última, el rosa a la cama y el rojo a la pared. El vídeo de esta prueba está disponible en la wiki [6].



(a) Marcador utilizado. (b) Parches planos generados.

Figura 5.14: Prueba con un único marcador.

5.8. Generación de mapa del entorno real con un marcador en el campo de visión

En este experimento se ha realizado un mapeado de la habitación utilizando algunas de las balizas para que en el campo visual de la cámara solo haya una a la vez. Para ello se han habilitado únicamente las balizas marcadas en la figura 5.15 y se ha mantenido una distancia, respecto de ellas, pequeña, aproximadamente un metro. En este test también se validará las funcionalidades incorporadas al mapeador para que no extraiga planos cuando el sensor pierde de vista balizas visuales. También se validará que haya un pequeño tiempo de espera desde que se encuentra una baliza hasta empezar a extraer de nuevo. Este tiempo de espera se ha establecido en dos segundos. El sensor se ha movido de forma continua y suave completando todo el mapeado en 60 segundos y la extracción de planos ha estado funcionando a 1 Hz.

En la figura 5.16 se pueden ver los siete parches planos generados. También se ha dibujado la traza del sensor para que se vea el movimiento que ha realizado (la orientación era siempre hacia las paredes). Se puede apreciar que no hay ninguna extracción errónea. En los momentos en los que no había balizas en el campo de visión no se han extraído

[6] `http://jderobot.org/Samartin-tfg#Single_marker_test`

(a) (b)

Figura 5.15: Marcadores activos durante el experimento en el que solo hay un marcador en el campo visual.

planos y el algoritmo de extracción ha esperado dos segundos tras encontrar una baliza de nuevo antes de volver a activarse. El parche verde corresponde a la pared del fondo que se ve en la figura 5.15(a) y los parches de abajo a la izquierda corresponden al pequeño pasillo que tiene la habitación que se ve en la figura 5.15(b) al fondo. El vídeo de este experimento está disponible en la wiki [7].

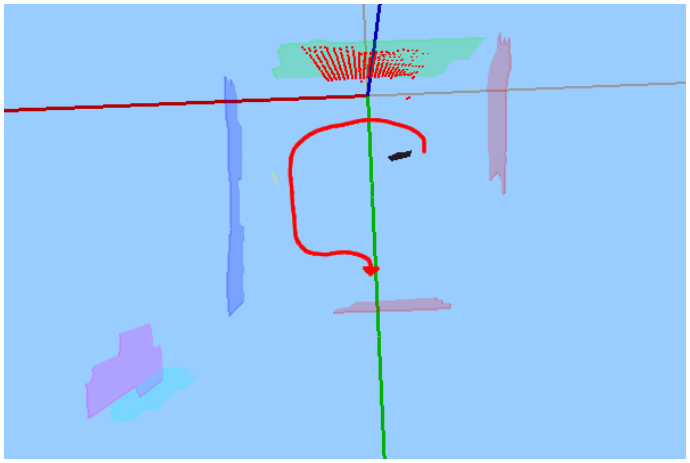


Figura 5.16: Parches planos generados en la prueba.

[7] `http://jderobot.org/Samartin-tfg#One_marker_at_a_time_test`

5.9. Generación de mapa del entorno real con varios marcadores en el campo de visión

En este experimento se han habilitado la mayoría de marcadores visuales de la habitación, exactamente los que se pueden ver rodeados por un círculo en la figura 5.17. De este modo, el sensor podrá, en ocasiones, ver más de una baliza al mismo tiempo, lo que disminuye considerablemente el error cuando el sensor está lejos de las balizas. Se ha tardado aproximadamente un minuto y diez segundos en realizar el mapa de la habitación y la frecuencia de extracción de planos ha sido de 1 Hz.

(a) (b)

Figura 5.17: Marcadores activos durante el experimento con varios marcadores en el campo visual.

En la figura 5.18 se puede ver el mapa de parches planos generados así como una traza del movimiento que ha realizado el sensor. Los parámetros para el algoritmo de fusión seleccionados han sido suficientemente altos como para tener en cuenta el error y no tener demasiados planos, pero afortunadamente no demasiado como para empezar a perder detalles del entorno. Esto se puede apreciar con el mueble que se ve en en la parte de la izquierda de la figura 5.17(a) y el parche de color amarillo que le corresponde. Si los valores de fusión hubiesen sido demasiado altos, se habría fusionado con el plano correspondiente a la pared de la izquierda. En la mitad de la prueba, la cámara pasa por una zona en la que se pierde la visión de marcadores y la extracción se pausa hasta que se vuelve a ver uno, esperando 2 segundos para retomarla. Con esta prueba y las anteriores

dos en el entorno real queda validado también que la extracción de planos se realiza de forma fluida y sin fallos. Este experimento se puede ver en la wiki [8].

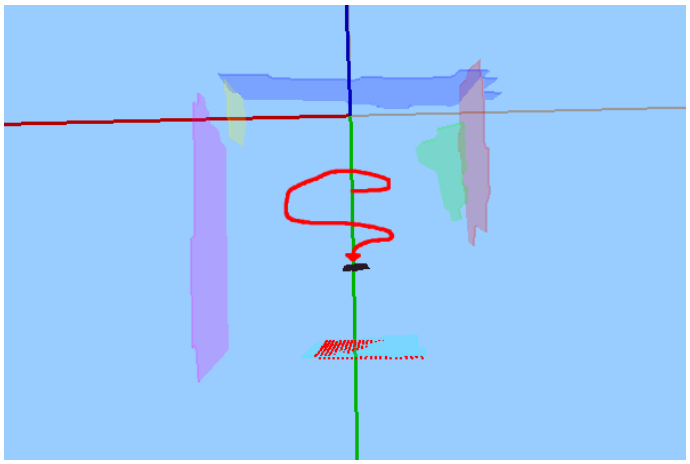


Figura 5.18: Reconstrucción de la habitación.

[8] http://jderobot.org/Samartin-tfg#Various_markers_at_a_time_test

Capítulo 6

Conclusiones

A lo largo de los capítulos anteriores se han expuesto los problemas abordados en este Trabajo Fin de Grado, las soluciones desarrolladas y los experimentos realizados para validar dichas soluciones. En este capítulo se resumen las conclusiones obtenidas y también se proponen algunas líneas de trabajo futuro que abre.

6.1. Conclusiones

La solución diseñada en este trabajo permite el mapeado fluido mediante parches planos grandes de entornos reales con movimiento suave y continuo, a partir de balizas visuales utilizando un sensor RGB-D, como muestra la prueba final realizada en la sección 5.9. Además, se pasó de un mapeador en simulación que no generaba mapas de planos sin errores de forma fluida a uno que sí lo hace. A continuación se repasan los subobjetivos planteados en el capítulo 2 y se resume cómo hemos resuelto cada uno de ellos:

- *Desarrollo de un generador sintético de nubes de puntos que pueda enviar los datos generados a otras aplicaciones.* Como se ha explicado en la sección 4.2, se ha desarrollado un componente de JdeRobot escrito en Python, que crea distintas nubes de puntos en distintos volúmenes de trabajo y que simula las nubes de puntos generadas por un sensor RGB-D. Cada nube de puntos consiste en puntos generados en un plano, siempre y cuando estén dentro del volumen de trabajo, con un valor de ruido gaussiano añadido en la componente normal del plano al que pertenecen. Se le ha implementado además un servidor ICE con el que se expone la información de las nubes de puntos generadas y que le permite conectarse con otros componentes.

Para ello, utiliza el interfaz de JdeRobot `pointCloud`.

- *Estudio del sincronismo de la aplicación generadora de parches planos para conseguir una extracción fluida en simulación mientras el sensor se mueve.* Tal y como se ha visto en la sección 4.3, se ha llevado a cabo un análisis del sincronismo de la aplicación `mapper3Drgbd` para encontrar las razones por las que la posición llegaba antes que la imagen de profundidad, generando extracciones de parches planos falsos cuando el sensor estaba en movimiento. Se han ajustado las frecuencias a la que funcionan los distintos hilos de los componentes, se ha implementado un buffer que asegura que las imágenes de profundidad y las posiciones salen sincronizadas de origen (el plugin `flyingKinect` en Gazebo) y añadido una estampa de tiempo a la información de la posición para compararla con la estampa de tiempo, que ya existía en la información de las imágenes, en recepción. Con las estampas de tiempo se han podido detectar dos causas independientes que producían el desfase: en primer lugar, las imágenes en ocasiones tardan más en llegar al mapeador porque tienen más tamaño que las posiciones y, en segundo, la posición es actualizada por Gazebo, en ocasiones, más rápido que las imágenes. Como solución a la primera causa se ha implementado un filtro en el mapeador que compara las estampas de tiempo que tiene la información, descartando aquellas desfasadas. Para la segunda causa se ha implementado un filtro que compara la posición e imagen que llegan en un instante determinado con las que llegaron en el momento anterior, filtrando aquellas parejas en las que solo ha cambiado uno de los dos tipos de datos.

- *Integración de los componentes en un mismo sistema para el mapeado de entornos reales y pruebas.* Como se ha podido comprobar en la sección 4.4, se ha implementado un servidor ICE en el componente `cam_autoloc` para proporcionar la localización que estima al componente `mapper3Drgbd`. Para que el mapeador pudiera entender esta localización, ha hecho falta realizar un encaje del sistema de referencia que utiliza, ya que era distinto del que tenía el componente de autolocalización. También ha sido necesario añadir una serie de funcionalidades al mapeador, así como ajustar los parámetros del algoritmo de extracción para su correcto funcionamiento en entornos reales. Las pruebas expuestas en el capítulo 5 validan, tanto el ajuste de algunos de los parámetros del algoritmo de mapeado, como el resultado final del sistema.

Del mismo modo que se han repasado los subobjetivos, a continuación se repasan los requisitos expuestos también en el capítulo 2 y cómo se han cumplido:

- *Se hará uso del entorno JdeRobot en su versión 5.3 y se desarrollarán componentes de forma modular programados en Python y C++.* El componente que se ha desarrollado en este trabajo desde cero, el generador sintético de nubes de puntos, se ha programado en Python haciendo uso de las interfaces de JdeRobot para exponer la información que en él se crea. El resto de componentes del entorno con los que se ha trabajado activamente y modificado, `mapper3Drgbd` y `cam_autoloc`, estaban escritos en C++. Todos los entornos de pruebas que se han expuesto además hacen uso de los componentes de JdeRobot.

- Las soluciones desarrolladas se ejecutan sobre Ubuntu 14.04 en las arquitecturas x86 y x64.

- *El sistema final tendrá un rendimiento tal que permita la ejecución en tiempo real con movimiento suave pero continuo.* Como se ha explicado en la sección 4.4.3 y validado en el capítulo 5 con los experimentos en el entorno real, la solución desarrollada permite ser ejecutada en tiempo real. Concretamente a 6 Hz, ya que es el tiempo que tarda en estimar la localización a partir de una imagen el componente `cam_autoloc`.

- Todo el código desarrollado en este trabajo ha sido liberado bajo licencia GPLv3 y se puede encontrar en el repositorio de Subversion [1].

Durante la realización de este Trabajo Fin de Grado se han adquirido conocimientos sobre técnicas de visión por computador utilizando sensores RGB-D. También se han adquirido conocimientos sobre el software y herramientas de infraestructura utilizado: C++, Python, Gazebo, OpenCV, ICE, CMake, Subversion, OpenCV, Eigen, Git, Glade y LaTeX.

[1] `https://svn.jderobot.org/users/samartin/tfg/`

6.2. Trabajos futuros

Para finalizar, en esta sección se proponen algunas líneas de investigación que pueden partir de este trabajo, ya sea para fijar nuevos objetivos o para mejorarlos:

- **Algoritmo de navegación autónoma**. A partir del sistema diseñado en este trabajo se puede diseñar un algoritmo de navegación autónoma que realice el mapeado del entorno o interaccione con él sin colisionar. Una idea interesante es utilizar un cuadricóptero para ello.

- **Implementar la posibilidad de que los parches puedan tener huecos internos**. En la implementación actual del componente `mapper3Drgbd`, no se tiene en cuenta que los parches planos, tal como están definidos, puedan tener huecos internos, como por ejemplo, el hueco de una ventana pequeña en una pared. Una mejora sería permitir que los parches planos, aparte de tener un contorno que defina su perímetro externo, también puedan tener contornos internos que definan huecos en su superficie.

- **Optimización del algoritmo del componente cam_autoloc**. El algoritmo de localización tarda en procesar cada imagen a color y estimar la posición a partir de ella unos 150 milisegundos en un ordenador normal. Esta cantidad es suficiente para que funcione en tiempo real y fluido en un ordenador de potencia media, pero si se emplean ordenadores de placa reducida, como la `Raspberry Pi`, o móviles y tabletas, el tiempo de procesamiento subirá y puede que ya no funcione en tiempo real.

Bibliografía

[1] David Marr. Representing and Computing Visual Information. *Massachusetts Institute of Technology*, 1978.

[2] Noah Snavely, Steven M. Seitz y Richard Szeliski. Photo Tourism: Exploring Photo Collections in 3D. *SIGGRAPH Conference Proceedings*, pages 835–846, New York, NY, USA, 2006.

[3] Marc Pollefeys, Reinhard Koch, y Luc J. Van Gool. Self-calibration and metric reconstruction inspite of varying and unknown intrinsic camera parameters. *International Journal of Computer Vision*, 32(1):7–25, 1999.

[4] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton y Olivier Stasse. MonoSLAM: Real-Time Single Camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052-1067, Junio 2007.

[5] Georg Klein y David Murray. Parallel Tracking and Mapping for Small AR Workspaces. In *6th IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR)*, pp.225-234, Noviembre 2007.

[6] Felix Endres, Jürgen Hess, Nikolas Engelhard, Jürgen Sturm, Daniel Cremers y Wolfram Burgard. An Evaluation of the RGB-D SLAM System. *International Conference on Intelligent Robot Systems (IROS)*, Octubre 2012.

[7] Christian Kerl, Jürgen Sturm y Daniel Cremers. Robust Odometry Estimation for RGB-D Cameras. *2013 IEEE International Conference on Robotics and Automation (ICRA)*, Mayo 2013.

[8] Erik Bylow, Jürgen Sturm, Christian Kerl, Fredrik Kahl and Daniel Cremers. Dense Tracking and Mapping With a Quadrocopter. 2013.

[9] Juan José García Cantero. Herramienta de Calibración de sensores RGBD en *JdeRobot. Proyecto Fin de Carrera Ing. Telecomunicación, Universidad Rey Juan Carlos*, 2013.

[10] Juan Navarro Bosgos. Construcción de Mapas 3D Compactos desde Sensores RGBD. *Proyecto Fin de Carrera Ing. Telecomunicación, Universidad Rey Juan Carlos*, 2015.

[11] Alberto López-Cerón pinilla. Autolocalización visual robusta basada en marcadores. *Trabajo Fin de Máster en Visión Artificial, Universidad Rey Juan Carlos*, 2015.

[12] Barry Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

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

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