UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Localización y generación de mapas del entorno (SLAM) de un robot por medio de una Kinect.

Proyecto Final de Carrera

Ingeniería Informática Superior

Autor: Joaquín Viñals Pons

Director: Ángel Valera, Enrique Bernabeu

29 de septiembre de 2012

Resumen

En este proyecto exploraremos las posibilidades del sensor Kinect de Microsoft en el ámbito de la robótica, así como la historia y el amplio repertorio de herramientas Open Source que nos permitirán desarrollar aplicaciones de navegación y mapeado haciendo uso del mismo.

Palabras clave: robot móvil, Kinect, robótica, SLAM

Índice general

1. Introducción **4**
1.1. Justificación 4
1.2. Objetivos 5

2. Desarrollo teórico **6**
2.1. Conocimientos previos 6
2.1.1. Hardware 6
2.1.2. Odometría 11
2.1.3. Dispositivo para la medición de distancias 12
2.1.4. Modelos matemáticos 16
2.2. SLAM 17
2.2.1. Introducción 17
2.2.2. Historia del problema del SLAM 17
2.2.3. Modelado y solución 19
2.2.4. *Landmarks* y extracción 19
2.2.5. RANSAC 20
2.2.6. Asociación de datos 21
2.2.7. Mapeado 23
2.3. Navegación 24
2.3.1. Algoritmos de muestreo 24
2.3.2. Trayectorias basadas en celdillas 25
2.4. Algoritmos de búsqueda 26
2.4.1. Búsqueda primero en anchura (BFS) 27
2.4.2. Búsqueda primero en profundidad (DFS) 27
2.4.3. Búsqueda informada. Algoritmos A*. 28
2.5. Historia de los controladores avanzados 29
2.5.1. Primeros controladores 29
2.5.2. Controladores durante los 80 y 90 30
2.5.3. Primeros intentos de control avanzado 31
2.5.4. Finales de los 90 34
2.5.5. Nuevos intentos 35

2.6. Generación actual 37
2.6.1. Nintendo Wii 37
2.6.2. Sony Playstation 3 38
2.6.3. Microsoft Xbox 360 38
2.7. Recursos y entornos de programación 41
2.7.1. LibFreenect. El conjunto de herramientas MRPT. . . . 41
2.7.2. OpenNI. El conjunto de herramientas ROS. 42
2.8. Caso de estudio: RGBDSLAM 49
2.9. HOGMAN 51
2.10. SURF 52
2.11. ICP 53
2.12. Entorno de desarrollo 54

3. Desarrollo práctico 56
3.1. Preámbulos y configuraciones 56
3.1.1. Instalación del Sistema Operativo 56
3.1.2. Instalación del *framework* ROS 60
3.1.3. Instalación de las dependencias necesarias para el proyecto 61
3.1.4. Instalación del programa RGBDSLAM 65
3.1.5. Estudio de RGBDSLAM 67
3.1.6. Instalación del entorno de desarrollo Eclipse y adaptación del proyecto 67
3.2. Desarrollo del proyecto 70
3.2.1. Introducción 70
3.2.2. Modelado 71
3.2.3. Estructuras de datos 71
3.2.4. Sistema de visión 77
3.2.5. Modelado del sistema de visión 79
3.2.6. Implementación del sistema de visión 80
3.2.7. Integración con RGBDSLAM 83
3.3. Plataforma final y adaptaciones 84
3.4. Análisis de una ejecución del programa. 87

4. Conclusiones y futuros proyectos 92

Capítulo 1

Introducción

1.1. Justificación

Desde siempre, el ser humano ha utilizado los recursos a su disposición como herramientas para ayudarle en la realización de las tareas de la forma más efectiva, rápida y segura.

A medida que la tecnología avanza, estas herramientas se convierten en maquinaria cada vez más compleja y capaz de realizar trabajos complicados de manera precisa y, muchas veces, mejor de lo que podría haberlo hecho un humano.

Con el fin de eliminar la necesidad de un operario humano en trabajos repetitivos, peligrosos o realizados en entornos donde una persona podría resultar herida o para explotar al máximo las ventajas que ofrecen estas herramientas, nace el deseo de automatizar todo lo posible estos procesos.

Una de las disciplinas que aparecen gracias a este deseo es la del estudio de los robots autónomos, máquinas capaces de tomar sus propias decisiones para conseguir cumplir un objetivo, como podría ser la exploración de entornos desconocidos.

Aunque en principio parezcan disciplinas separadas, existe una cercana relación entre los campos de la electrónica con aplicaciones recreativas, como las consolas y la telefonía móvil, y la investigación científica, más orientada a la búsqueda de soluciones con un impacto práctico en nuestra vida cotidiana.

Con un mercado en constante expansión, las empresas que se encargan del diseño y la construcción de video consolas y teléfonos móviles han dedicado grandes cantidades de dinero y recursos al desarrollo de dispositivos capaces de ofrecer al usuario una experiencia lo más inmersiva y completa posible, así como la posibilidad de diferenciarlas de la competencia.

De esta forma, aparecen en el mercado herramientas con costes de produc-

ción reducidos y precios competitivos, pero con gran potencial en el ámbito científico. Herramientas como los diversos mandos de la consola Wii de Nintendo, o la cámara Kinect de Microsoft, han contribuido a que el público general tenga alcance a equipamiento para la investigación que no habría podido obtener de otra manera.

En este proyecto utilizaremos una cámara Kinect y las herramientas a nuestra disposición para encontrar una aproximación a la solución del problema de la autolocalización y el mapeado de un robot móvil, uno de los principales problemas de la navegación autónoma de los robots móviles.

1.2. Objetivos

El objetivo de este proyecto es explorar las posibilidades de la cámara Kinect como sensor en el ámbito de la robótica móvil, y conocer las diversas herramientas de desarrollo disponibles en este campo. Se pretende implementar un algoritmo que permita extraer la información disponible del sensor y con ella construir un mapa del entorno, al tiempo que el robot provisto con la cámara obtiene las órdenes de movimiento necesarias para cumplir su objetivo.

Una lista de los objetivos que se han marcado para este proyecto:

- Estudio del sensor Kinect y sus restricciones físicas.
- Estudio del software disponible para el sensor.
- Estudio de las estructuras de datos proporcionadas por el sensor para su utilización en el proyecto.
- Estudio de algoritmos de generación de mapas y de navegación.
- Análisis de las imágenes capturadas por la cámara para la navegación y detección de obstáculos.
- Desarrollo de una aplicación que englobe los conocimientos adquirido.

Capítulo 2

Desarrollo teórico

2.1. Conocimientos previos

SLAM, del Inglés *Simultaneous Localization And Mapping*, o en Español: Localización Y Mapeado Simultáneos o también Localización y Modelado Simultáneos. Es una técnica usada por robots y vehículos autónomos para construir un mapa de un entorno desconocido en el que se encuentra, a la vez que estima su trayectoria al desplazarse dentro de este entorno.

Antes de explicar en qué consiste y cómo es posible resolver este problema, es necesario analizar las distintas alternativas que tenemos a nuestro alcance, ya que formaran parte de la solución[1]:

2.1.1. Hardware

En esencia, tenemos dos componentes hardware necesarios para poder implementar un algoritmo que resuelva el problema del SLAM: el robot móvil y sus sensores, en particular los dispositivos usados para la medición de distancias.

Robot móvil

Un robot móvil es una máquina automática capaz de moverse en un determinado entorno.

Los robots móviles, tienen la capacidad de desplazarse alrededor de un determinado entorno y no están fijos en una posición determinada. En contraposición, otros robots, como los industriales, generalmente consisten en un brazo articulado y una herramienta que está anclado en una superficie fija.

Los robots móviles son foco de investigación actual, y casi todas las grandes universidades tienen un laboratorio dedicado al desarrollo de aplicaciones

en robótica móvil. Los robots móviles también pueden ser encontrados en la industria y en entornos militares y de seguridad.

También posible encontrarlos en entornos domésticos, tanto como juguetes como realizando tareas domésticas (por ejemplo, el robot aspirador *Roomba*).

Clasificación de los robots móviles

Atendiendo a sus capacidades de movimiento, podemos clasificar los robots móviles en las siguientes categorías:

Robots rodantes Son aquellos que, como su nombre indica, se desplazan haciendo uso de ruedas, generalmente montadas por pares en una configuración 2+2 como las de un vehículo por mera simplicidad. Habitualmente solo dos de sus ruedas presentan tracción y otras dos dirección, de forma que sea posible maniobrar el robot con un solo servomotor.

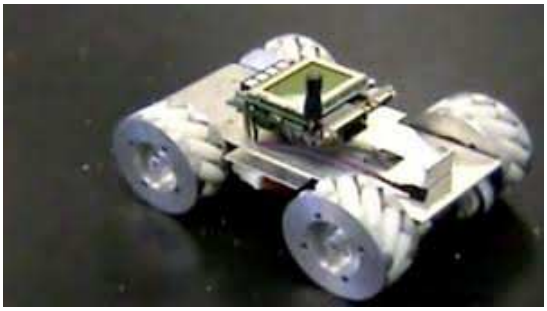

Figura 2.1: Robot rodante con cuatro ruedas.

También es frecuente encontrar distribuciones de ruedas montadas en modo triciclo, donde una rueda sirve para la dirección y las otras dos aportan la tracción. Otra opción es que la tercera rueda simplemente sea una rueda 'loca' y las otras dos aporten tanto la tracción como la dirección, mediante el método de las orugas tratado más adelante.

Figura 2.2: Mini robot espía con seis ruedas.

Existen algunos casos especiales en los que se usan otras configuraciones que dotan al robot de mejor adaptación a terrenos difíciles. En estos casos los algoritmos de control de movimiento adquieren una mayor complejidad, proporcional al número de elementos direccionables de forma independiente.

Por último cabría considerar a los robots con orugas como un tipo de robot odante en el que se substituyen las ruedas por un mecanismo de oruga para la tracción. La dirección se consigue parando una de las orugas o haciéndolas girar en sentido contrario.

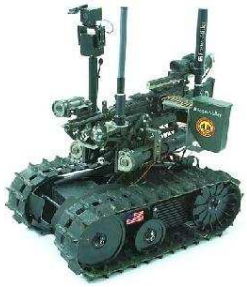

Figura 2.3: Robot dotado de orugas.

Robots andantes Respecto a los robots construidos a imagen y semejanza humana, con dos piernas, las técnicas de control necesarias son varias, pero todas ellas hacen uso de complejos algoritmos para poder mantener el equilibrio y caminar correctamente. Todos ellos son capaces de caminar bien sobre suelos planos y subir escaleras en algunos casos, pero no están preparados para caminar en suelos irregulares.

Algunos incluso pueden realizar tareas como bailar, luchar o practicar deportes, pero esto requiere una programación sumamente compleja que no

siempre está a la altura del hardware del robot y de su capacidad de procesamiento.

Figura 2.4: Robot bípedo humanoide.

Robots reptadores Una clase curiosa de robots, creados basándose en animales como las serpientes, su forma de desplazarse es también una imitación de la usada por estos animales. Están formados por un número elevado de secciones que pueden cambiar de tamaño o posición de forma independiente de las demás pero coordinada, de forma que en conjunto provoquen el desplazamiento del robot.

Figura 2.5: Robot reptador.

Robots nadadores Estos robots son capaces de desenvolverse en el medio acuático, generalmente enfocados a tareas de exploración submarina en zonas donde no es posible llegar pos ser de difícil acceso o estar a profundidades que el cuerpo humano no tolera.

Aparte de lo puramente anecdótico, se ha demostrado que la estructura corporal de los peces así como el movimiento que realizan durante su desplazamiento en el agua, es uno de los métodos más óptimos de movimiento

submarino dado que aprovecha la energía de forma muy eficiente y permite mayor control en la navegación, produciendo mucho menos ruido y turbulencias. Es por todo esto que se está tendiendo a estudiar y emular en lo posible el comportamiento de estos animales a la hora de crear nuevos robots subacuáticos.

Figura 2.6: Robot nadador con forma de pez.

Robots voladores Conquistados los dominios del mar y la tierra solo queda una meta por alcanzar en el mundo de la robótica: ser capaces de poner robots en el cielo. Para ello y por el momento existen dos aproximaciones, en función de su principio de vuelo y estructura:

Helicópteros Habitualmente helicópteros RC convencionales a los que se les añade la electrónica necesaria para tener visión artificial y capacidad de toma de decisiones autónoma.

Figura 2.7: Quadracoptero.

Drones o aviones no tripulados Actualmente en uso por el ejército de EEUU para tareas de logística en operaciones militares, apoyo cartográfico así como tareas de espionaje. Aunque no es habitual pueden estar dotados tanto de contramedidas para repeler agresiones como de armamento para realizar ataques.

Figura 2.8: Avión espía no tripulado o *drone*

2.1.2. Odometría

El concepto de odometría se define como el estudio de la estimación de la posición de vehículos con ruedas durante su navegación. En robots móviles se utiliza para estimar su posición relativa a su localización inicial. Es bien sabido que dada una buena aproximación respecto a su localización inicial, la odometría proporciona una buena precisión a corto plazo, no obstante, la odometría es la integración de información incremental del movimiento a lo largo del tiempo, lo cual conlleva inevitablemente a la acumulación de errores.

La odometría se basa en ecuaciones sencillas de implementar, las cuales hacen uso del valor de los encoders de las ruedas del robot para traducir las revoluciones de las ruedas a un desplazamiento lineal relativo al suelo. Este concepto básico puede llevar a imperfecciones y a cálculos erróneos si por ejemplo las ruedas patinan, o se produce una sobre-aceleración (errores no sistemáticos).también depende directamente de la exactitud de la localización y orientación inicial del robot.

Situar un robot a mano, en una posición determinada con una precisión de milímetros, puede no resultar excesivamente complicado, pero cuando se trabaja con varios robots los cuales deben compartir las coordenadas absolutas de un escenario, puede resultar realmente complejo e imperfecto. Además, el correcto funcionamiento de la odometría depende también de errores siste-

máticos como la medición de los diámetros de las ruedas, su alineamiento, la resolución discreta del encoder o la distancia de separación entre las ruedas.

A pesar de sus limitaciones, el uso de la odometría en investigación es uno de los pilares más importantes del sistema de navegación de un robot [2].

Características importantes

A la hora de diseñar nuestro algoritmo, es importante conocer algunas características del robot que usaremos, como por ejemplo su facilidad de uso y su precio, pero sobre todo su precisión odométrica.

La precisión de la odometría del robot nos permite saber qué tan bien es capaz el robot de estimar su propia posición basado únicamente en el conocimiento de su forma de desplazamiento (mediante encoders en las ruedas, por ejemplo). El robot no debe tener un error superior a 2 centímetros por cada metro desplazado, ni superior a 2º por cada 45º girados.

2.1.3. Dispositivo para la medición de distancias

Es necesario poseer algún tipo de sensor que nos permita extraer información en tres dimensiones del entorno en que se encuentra.

Según el tipo de tecnología que se utilice, podemos clasificar estos dispositivos en las siguientes categorías:

Láser

El escáner láser es el tipo de sensor que está en continua expansión en mundo de la robótica. Las técnicas dominantes para la obtención de medidas de distancia basadas en láser son las denominadas técnicas de tiempo de vuelo (TOF, por sus siglas en inglés) y las técnicas de cambio de fase.

Un sistema TOF lanza un pulso láser de corta duración y se mide el tiempo que tarda éste en volver. La distancia se calcula entonces como un medio de la velocidad de la luz por el tiempo que ha tardado el láser en realizar un viaje de ida y vuelta.

En sistemas de cambio de fase, se transmite una ola continua de luz. La idea es comparar la fase la señal retornada con una señal de referencia generada por la misma fuente.

Un escáner láser presenta multitud de ventajas:

- Es rápido. Es decir, la medida, para tareas domésticas, puede considerarse instantánea. En la práctica, esto significa que no es necesario

compensar el movimiento de la paltaforma en la que está instalado el escáner.

- La precisión es bastante elevada. Las nuevas generaciones de lásers tienen una precisión con un margen de error inferior a 10 milímetros.
- La resolución angular de 0.5º o de 0.25º dependiendo de la operación llevada a cabo, es superior a la ofrecida por un sónar.
- Los datos obtenidos de un escáner láser pueden ser interpretados directamente como la distancia hasta un obstáculo en una posición determinada, en oposición a la imágen extraída de una cámara, que requiere mucho más esfuerzo para obtener la misma información.

Entre otras, también presenta las siguientes desventajas:

- El sensor ofrece información limitada a un cierto plano. Cualquier cosa por encima o por debajo de dicho plano no pueden ser detectadas por el escáner.
- Los sensores son muy caros.
- Algunos materiales son invisibles para el escáner, como los cristales.

El escáner láser es el mejor sensor para extraer propiedades de superficies planas, como las paredes, gracias a la densidad de la información de distancias que ofrece. Por el mismo principio, también es perfecto para detectar marcos de puertas.

Figura 2.9: Sensor láser Hokuyo.

Sónar

Este tipo de sensores funcionan tanto como emisores como receptores de ultrasonidos, de forma análoga a los sensores láser. Igual que estos, utiliza técnicas TOF para detectar distancias.

El éxito obtenido al aplicar este tipo de tecnologías en robots móviles depende en gran medida del acercamiento utilizado en la clasificación de los datos obtenidos.

Las ventajas principales de este tipo de sensores son:

- Bajo coste. Un dispositivo sónar de gran calidad cuesta en torno a los 10 dólares (En comparación con los 5000$ que puede costar un escáner láser).
- Gran precisión. El error en la medida es, usualmente, inferior al 1 %.
- Pueden operar sin requisitos de visibilidad ni de luz.

Sin embargo, este tipo de sensores también tienen sus desventajas:

- Reflexiones múltiples. La onda de sonido puede rebotar en varios objetos antes de volver al receptor, causando falsas medidas.
- Interferencias. Cuando otro sónar envía una onda idéntica en el rango de acción del sensor pueden producirse errores de lectura.
- Mala resolución angular.
- Rango de medidas limitado. En principio el rango de detección para algunos sónar es de hasta 10 metros, pero son incapaces de detectar objetos pequeños que se encuentran más allá de 5 metros.
- En comparación con otro tipo de sensores, como el láser, la velocidad del sonido para los sónar supone una limitación en cuanto a velocidad.

Figura 2.10: Sensor de distancias por sonar.

Visión

Una cosa que falta a todos los sensores de detección de distancias es la habilidad de extraer propiedades de las superficies observadas e identificar objetos. Las imágenes a color (o escala de grises) nos permiten utilizar un amplio repertorio de información para identificar y localizar componentes del entorno.

Las principales ventajas de los sensores de visión son:

- Gran cantidad de información.
- Capacidad de extraer información en tres dimensiones del entorno.
- Las cámaras son sensores pasivos, que no necesitan emitir sonido o luz como los sónar y los escáneres láser.

Sus inconvenientes son:

- Alto coste computacional para extraer la información de las imágenes.
- La visión está altamente influenciada por la cantidad de luz disponible.
- Son dispositivos caros.

Es mucho más fácil interpretar la información obtenida por los otros tipos de sensores, con menor coste computacional y, por tanto, velocidad. Por este motivo la visión no suele usarse con tanta asiduidad.

Figura 2.11: Cámara estereoscópica.

El algoritmo que creemos estará íntimamente ligado a las elecciones de hardware que hagamos. La precisión, la tolerancia al ruido, el tipo de entorno en el que nuestro robot y nuestros sensores puedan trabajar nos condiciona en el desarrollo del modelo matemático que usaremos para buscar nuestra solución al problema del SLAM.

2.1.4. Modelos matemáticos

Antes de adentrarnos en el la resolución del problema del SLAM, necesitamos conocer algunos fundamentos estadísticos en los que se basan sus soluciones.

Teorema de Bayes

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}. \tag{2.1}$$

El teorema de Bayes es importante ya que es capaz de vincular la probabilidad de que un suceso A suceda conociendo la probabilidad condicional de dicho suceso con respecto a otro suceso B, siempre que se conozca la distribución de probabilidad marginal de A[3].

Cuando tratamos de aplicar esta regla de Bayes a la resolución exitosa del SLAM, podemos considerar el suceso A como uno de los diferentes estados en los que se encontrará el sistema (explicado en 2.2.3), basado en la información del suceso B, por ejemplo, las medidas obtenidas por un sensor. La regla dice que podemos resolver este cálculo de manera muy sencilla multiplicando dos términos: la probabilidad (en nuestro modelo) de obtener la medida B en el estado A, y el grado de confianza que damos a que A sea precisamente el estado del sistema antes de recibir los datos.

Filtro de Kalman

El filtro de Kalman, o estimación lineal cuadrática, es un algoritmo que usa una serie de medidas observadas a lo largo del tiempo, que contienen un cierto nivel de ruido (como las medidas de un sensor o la propia estimación de la posición del robot) y otras inexactitudes, y produce estimaciones de variables desconocidas que tienden a ser más precisas de lo que serían basadas en una simple medición. Más formalmente, un filtro de Kalman opera recursivamente en flujos de datos con ruido para producir una estimación estadísticamente óptima del estado del sistema.

El algoritmo funciona como un proceso en dos fases: en la fase de predicción, el filtro de Kalman produce estimaciones del estado actual de las variables, junto a algunas incertidumbres. Una vez capturada la salida del siguiente estado, el filtro actualiza sus estimaciones utlizando una media ponderada, con mayor peso a las predicciones con un mayor nivel de certeza.

Este filtro asume que el sistema que intenta predecir se comporta de forma lineal, y que todas las medidas de error tienen una distribución Gaussiana.

Existen diversas variaciones de este filtro, entre ellas el filtro extendido de Kalman, usado comúnmente en la resolución del SLAM.

Estas extensiones presentan algunas complicaciones con respecto al filtro original. Por ejemplo, el filtro extendido de Kalman no es un estimador estadístico óptimo, y si la estimación inicial del estado es errónea el filtro divergirá rápidamente del sistema real, debido a la linealización subyacente que el filtro lleva a cabo de estos sistemas no lineales.

2.2. SLAM

2.2.1. Introducción

El problema de la localización y mapeado simultáneos consiste en descubrir si es posible para un robot móvil navegar a través de un entorno desconocido y construir, de manera incremental, un mapa consistente del mismo mientras que determina, al mismo tiempo, su posición dentro de este mapa. Una solución a este problema se ha considerado "el Santo Grial" por la comunidad de especialistas en robótica móvil, ya que dotaría a los robots con las herramientas necesarias para ser completamente autónomos.

La "solución" del problema del SLAM ha sido uno de los sucesos más notables para la comunidad de los últimos años. SLAM ha sido formulado y resuelto como problema teórico en distintas formas. SLAM ha sido, también, implementado en diversos campos de la robótica móvil, como los robots de interiores, en exteriores, robots subacuáticos y robots voladores. A un nivel teórico y conceptual, SLAM puede considerarse un problema resuelto. Sin embargo, aún quedan varios problemas en la aplicación práctica de soluciones más generales de SLAM, y particularmente en la utilización y construcción de mapas perceptualmente ricos como parte de un algoritmo de SLAM[4].

2.2.2. Historia del problema del SLAM

El nacimiento del problema del SLAM ocurre durante la Conferencia sobre Robótica y Automática del IEEE, en la ciudad de San Francisco, en 1986. Durante estos años, los métodos probabilísticos comenzaban a ser introducidos en los campos de la robótica y la inteligencia artificial. Un grupo de asistentes mantuvieron una conversación sobre la aplicación de métodos de estimación a los problemas de mapeado y localización.

Como resultado de esta conversación, se reconoció que el mapeado consistente probabílistico era un problema fundamental de la robótica que merecía ser tratado. Durante los siguientes años se describen en diferentes estudios

las bases estadísticas para describir las relaciones entre los objetos de referencia (conocidos como *landmarks*) y la manipulación de la incertidumbre geómetrica. Un elemento clave de estos trabajos fue mostrar que debía existir una gran correlación entre las estimaciones de la localización de diferentes *landmarks*, y que ésta debía crecer con las sucesivas observaciones.

Al mismo tiempo, se estaban produciendo los primeros avances en la navegación visual y en la navegación por sónar, usando algoritmos basados en filtros de Kalman. Estas dos ramas separadas de investigación combinadas ayudaron en la investigación de la navegación de robots móviles basada en *landmarks*.

Estas investigaciones demostraron que para un robot móvil moviéndose a través de un entorno desconocido y capturando observaciones relativas de puntos de referencia, las estimaciones de éstas están necesariamente correlacionadas entre sí a causa del error común en la estimación de la posición del vehículo. La implicación de esto era profunda: una solución consistente al problema combinado de la localización y el mapeado requeriría un estado conjunto compuesto por la posición de cada una de los objetos de referencia, que debería ser actualizado con cada observación de unos de estos objetos. Esto requeriría que el estimador de posición almacenara un gran vector de estados (de orden igual al número de *landmarks* que se conserven del mapa), con un coste computacional escalado al doble del cuadrado de *landmarks*.

Lamentablemente, estos primeros estudios no observaron las propiedades convergentes del error en las estimaciones o del mapa. De hecho, se asumió que los errores en la estimación del mapa no convergirían y que, por el contrario, exhibirían un comportamiento errático sin cota para el error. Así, dada la complejidad computacional del problema del mapeado y con un completo desconocimiento del comportamiento convergente del mapa, los investigadores se centraron en una serie de aproximaciones a la solución al problema del mapeado consistente que asumía, e incluso forzaba, la minimización de las correlaciones entre los objetos de referencia, reduciendo así el filtro completo en una serie de filtros vehículo-referencia sin relación entre sí.

El descubrimiento más importante fue que el problema del mapeado y la localización, una vez formulado como un único problema de estimación, era de naturaleza convergente. Aun más, se reconoció que las correlaciones entre los objetos de referencia, que otros científicos intentaban minimizar o eliminar, eran la pieza clave del problema y que, de hecho, cuanto más ricas fueran estas correlaciones, mejor sería la solución[5].

2.2.3. Modelado y solución

El proceso del SLAM consiste en un cierto número de pasos: extracción de características, asociación de datos, estimación del estado y actualización de las características. La finalidad del proceso es usar el entorno para actualizar la posición del robot. Dado que la odometría del robot no es enteramente fiable, no podemos depender directamente de ella. Debemos usar sensores de distancia para corregir esta posición.

Esto se consigue extrayendo características del entorno y re-observándolas mientras el robot se mueve. Un filtro extendido de Kalman (EKF, por sus siglas en inglés) es responsable de actualizar la estimación de la posición del robot basándose en estas características, a las que hemos llamado puntos de referencia o *landmarks*.

El filtro mantiene un estimado de la incertidumbre en la posición del robot y también de los *landmarks* que el robot ha visto en su entorno.

2.2.4. *Landmarks* y extracción

Los puntos de referencia son características del entorno que pueden ser fácilmente distinguidas y re-observadas. Son utilizadas por el robot para poder saber dónde se encuentra.

Hay una serie de requisitos para considerar una característica como un buen *landmark*[1]:

- Debe ser posible observarla desde distintas posiciones y ángulos.
- Debe ser lo suficientemente única como para poder ser distinguida de otras características. Es decir, si uno observa dos características y las re-observa en el futuro, debe ser posible identificar cuál es cuál. Cuanto más cerca se encuentran unas de otras, más difícil es realizar esta tarea.
- Aquellas características que decidamos deben ser *landmarks*, deben ser suficientes como para que el robot no circule durante largos periodos de tiempo sin observar ningún *landmark*, ya que el robot podría perderse.
- Si decidimos que algo debe ser un *landmark*, este debe permanecer estacionario. Una referencia en movimiento provocaría datos y estimaciones erróneas.

La extracción de características puede conseguirse de diversas formas. Explicamos el método RANSAC, ya que será utilizado por el algoritmo que veremos en 2.8.

2.2.5. RANSAC

RANSAC (del inglés, *RANdom Sampling Consensus*, consenso en un muestreo aleatorio) es un método iterativo que puede ser utilizado para extraer líneas rectas de una lectura de muestras de un sensor de distancias.

Es usado en estadística como una herramienta para extraer datos anómalos, valores que difieren de la distribución normal de la población a la que pertenecen.

Gracias a estas características, es posible utilizar este método para distinguir líneas rectas en un entorno en tres dimensiones, como las formadas por las paredes, techos, suelos y muebles que pueden encontrarse en un ambiente de interior.

RANSAC encuentra estas líneas escogiendo una muestra aleatoria de lecturas del sensor de distancias y utiliza una técnica de análisis numérico que intenta encontrar la función que mejor se ajusta a la distribución de puntos.

Esta técnica se conoce con el nombre de mínimos cuadrados, ya que pretende minimizar el cuadrado del error producido al restar la posición de la estimación con respecto a los puntos estimados. Al elevar el error al cuadrado obtenemos un conjunto de valores que siempre serán positivos, mientras que se otorga un mayor peso a los errores más grandes y un menor peso a los errores más pequeños.

Una vez realizada esta estimación, RANSAC comprueba cuantas lecturas de su muestra aleatoria están cerca de esta aproximación. Si el número de coincidencias es superior a un límite, o consenso, podemos estar seguros de haber observado una línea recta, y por tanto un segmento de pared (o puerta o ventana).

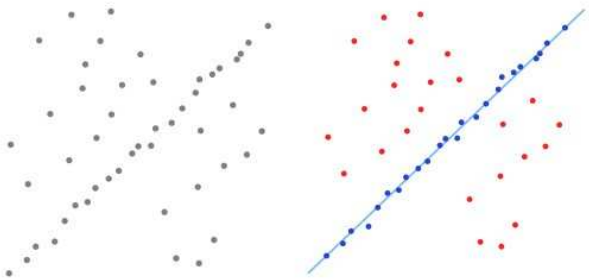

Figura 2.12: Filtrado de datos anómalos y extracción de línea por RANSAC.

2.2.6. Asociación de datos

El problema de la asociación de datos consiste en hacer corresponder las distintas observaciones de un mismo *landmark*. Es decir, ser capaces de identificar cuándo estamos observando una referencia ya extraída en una iteración anterior.

En la práctica, pueden surgir los siguientes problemas:

- Es posible que no se re-observen *landmarks* en cada paso del algoritmo.
- Es posible observar y extraer un *landmark*, y luego no ser capaz de volverlo a ver en la duración de la ejecución del algoritmo.
- Es posible asociar, erróneamente, un *landmark* a otro visto anteriormente.

Como se ha visto en 2.2.4 no debería ser posible extraer *landmarks* que causaran los dos primeros problemas, lo que quiere decir que hemos creado MALOS *landmarks*.

Aun con buenos algoritmos de generación de *landmarks*, es posible crear malas referencias. Para poder lidiar con este problema, es necesario crear una política de asociación de datos que permita minimizar estos errores.

Una solución común se basa en la creación de una base de datos que contendrá los objetos de referencia. Esta tabla está usualmente vacía. La primera regla que debe usarse es la de no considerar un objeto de esta tabla a no ser que se haya visto un número N de veces; de esta forma, eliminamos los casos en los que extraemos un *landmark* incorrecto.

En cada nueva observación, extraemos todas las referencias que encontremos y las asociamos a la referencia más cercana que exista en nuestra base de datos (y que hayamos visto más de N veces). El cálculo de las distancias puede hacerse por distancia euclídea, por ejemplo. A esta técnica se la conoce como "nearest-neighbor", vecino más cercano.

Una vez asociada, la pareja de *landmarks* se analiza usando un filtro (como un filtro de Kalman) que nos dice si podemos considerar, dentro de un margen de incertidumbre, que ambas referencias se corresponden. De ser así, se incrementa el número de veces que se ha observado esta *landmark* en 1. En caso contrario, se incluye esta referencia como una nueva *landmark* en la base de datos vista 1 vez.

Tan pronto el proceso de extracción de características y la asociación de datos finaliza, el proceso de SLAM puede considerarse dividido en tres fases:

- Actualizamos el estado estimado del robot usando la información dada por la odometría. Conociendo las acciones de control aplicadas al robot y la posición (estimada) anterior, es fácil estimar la posición actual.

- Corregir la estimación tras la re-observación de los *landmarks*. Usando la estimación de la posición es posible estimar dónde deberían estar ubicados. Usualmente, encontraremos diferencias. A estas diferencias las llamaremos "innovación". Básicamente, la innovación es la diferencia entre la posición estimada del robot y la posición real del robot, basada en lo que el robot es capaz de "ver". En este paso, la incertidumbre de cada *landmark* re-observado es actualizado también para reflejar los cambios recientes.

- Finalmente, comenzaremos nuevamente el ciclo, añadiendo nuevos *landmarks* al mapa del robot. Utilizaremos la información actualizada sobre posición e incertidumbre obtenida en este paso.

Así pues, el proceso del SLAM puede esquematizarse de la siguiente manera:

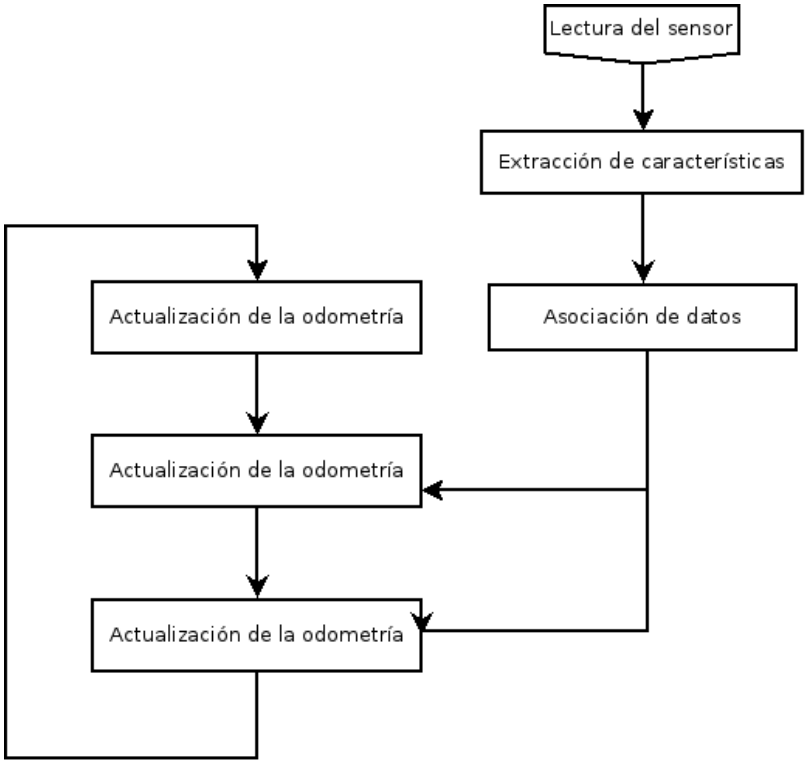

Figura 2.13: Esquema de un algoritmo de SLAM.

2.2.7. Mapeado

El objetivo del mapeado es representar la información obtenida del entorno de forma que sea posible utilizarla para planificar la navegación y la resolución de tareas. Existen dos enfoques diferentes: topológico y métrico.

Las representaciones topológicas únicamente consideran "lugares" y las conexiones entre ellos. Estos mapas tienen forma de grafos, donde los nodos representan distintas posiciones alcanzables por el robot y las aristas, la posibilidad de acceder a uno desde el otro. Es posible almacenar las distancias entre posiciones como peso de las aristas que las unen.

Las representaciones geométricas consideran el espacio bidimensional en el que se producirá el desplazamiento y ubica en éste los objetos. Las coordenadas de los objetos son elegidas con precisión. Ésta es la representación preferida al resolver el problema del SLAM.

Un ejemplo de representación geométrica es la generación de mapas por celdillas de ocupación. El método se basa en discretizar el espacio, dividiéndolo en unidades de tamaño predefinido, que se clasifican como ocupadas o vacías con un determinado nivel de confianza o probabilidad.

Estas soluciones parten de la hipótesis de que la posición del robot es conocida. En la práctica, se necesita de algún método de localización que estime la posición del robot en cada instante que, en este caso, no es considerada una variable estocástica. La precisión que alcanzan estos mapas en la descripción del entorno (tanto mayor cuanto más fina es la división del espacio), permite que el algoritmo de localización empleado acumule errores reducidos a lo largo de intervalos prolongados de tiempo. Así pues, la mayor desventaja de estos métodos es la pérdida de potencia que se deriva de no tener en cuenta la incertidumbre asociada a la posición del robot, lo cual origina que su capacidad para cerrar bucles correctamente se vea mermada.

Entre sus ventajas cabe destacar las siguientes:

- El algoritmo es robusto y su implementación sencilla.
- No hace suposiciones acerca de la naturaleza geométrica de los elementos presentes en el entorno.
- Distingue entre zonas ocupadas y vacías, consiguiendo una partición y descripción completa del espacio explorado. Por esto motivo es popular en tareas de navegación, al facilitar la planificación y generación de trayectorias empleando métodos convencionales.
- Permite descripciones arbitrariamente densas o precisas del mundo, simplemente aumentando la resolución de la rejilla que lo divide (es

decir, disminuyendo el tamaño de las celdillas individuales). Como es lógico, esto va en detrimento del rendimiento computacional del algoritmo.

- Permite una extensión conceptualmente simple al espacio tridimensional.

Una vez estimada la posición del robot y elegida la representación del mapa, podemos elegir la estrategia para conseguir elegir el camino que habrá de seguir el robot durante su exploración. Se conoce a este problema como "planificación de movimientos" o "problema de la navegación".

2.3. Navegación

Una forma básica de este problema es la producción de un movimiento continuo que conecte una posición inicial y una posición final, evitando cualquier tipo de colisiones con los obstáculos que existan en el entorno (paredes, muebles, escaleras, etc).

Cuando el número de dimensiones en las que puede desplazarse el robot es reducido, es posible resolver este problema con algoritmos que discretizan el espacio aplicando un patrón de celdillas de ocupación.

En casos con mayor complejidad espacial, el tratamiento exacto de la planificación de caminos puede considerarse computacionalmente indecidibles. En estos casos, es posible crear algoritmos basados en el muestreo que han demostrado ser bastante eficaces en muchos casos. No es posible detectar los casos en los que no existe un camino entre el inicio y el final, pero la probabilidad que tienen estos algoritmos de fallar decrece cuanto mayor sea el tiempo empleado en la navegación.

Los algoritmos de muestreo se consideran extremadamente buenos en planificación de movimientos en espacios multidimensionales, y se han aplicado a problemas con docenas e incluso cientos de dimensiones (manipuladores robóticos, moléculas biológicas, personajes animados digitales, robots humanoides).

2.3.1. Algoritmos de muestreo

Un algoritmo básico toma muestras de N diferentes configuraciones (por ejemplo, posiciones de las articulaciones en un brazo robótico) y las proyecta al espacio del problema, C. De todas ellas, elegirá únicamente aquellas que permanezcan dentro de las restricciones del sistema (colisiones con obstáculos, por ejemplos) y las utilizará como marcas o milestones.

Se trazará un camino entre dos milestones P y Q si el camino completo entre ambas está completamente dentro del espacio libre de restricciones. Para encontrar un camino que una la salida (S) y la meta (G), se incluyen como marcas en el camino. Si es posible encontrar un camino que una S y G, entonces el algoritmo devuelve ese camino y finaliza. En caso contrario, o bien no existe una solución, o el algoritmo no ha tomado un número suficientemente grande de muestras N.

Está demostrado que mientras mayor sea la facilidad de detectar qué posiciones cumplen las restricciones, el número de muestras N crece y la posibilidad de que el algoritmo encuentre una solución crece exponencialmente.

Hay muchas variaciones a esta versión básica del algoritmo:

- Típicamente, es mucho más rápido comprobar sólo segmentos cercanos entre marcas, en contraposición a comprobar todas las posibles uniones.
- Distribuciones de muestreo no uniformes intentan ubicar marcas en áreas que mejoran la conectividad del camino.
- Si el número de movimientos es limitado, es más eficiente no construir todos los posibles caminos del espacio alcanzable. Un árbol de búsqueda suele ofrecer buenos resultados.

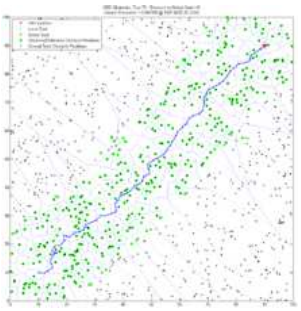

Figura 2.14: Estimación de la trayectoria óptima por un algoritmo de muestreo.

2.3.2. Trayectorias basadas en celdillas

Estos algoritmos discretizan el espacio en el que el robot puede moverse en una matriz de celdillas, donde cada celdilla representa un posible estado (posición, en el caso de un robot móvil) del robot. El robot puede desplazarse a cualquier celdilla adyacente, siempre y cuando la celdilla represente

un estado alcanzable (por ejemplo, un obstáculo representa un estado no alcanzable). Esto reduce el conjunto de acciones posibles en cada estado y es posible utilizar algoritmos de búsqueda, como los algoritmos A*, para encontrar la solución si el problema se modela corectamente en forma de grafo de decisión (ver 2.4.3).

Las implementaciones tradicionales producen caminos que están restringidos a desplazamientos en ciertos ángulos, múltiplos de un determinado ángulo base (figura 2.3.2). Ésto produce, generalmente, caminos sub-óptimos. Ciertos algoritmos ofrecen la posibilidad de realizar movimientos en cualquier ángulo, encontrando así caminos más cortos.

Los algoritmos de búsqueda por celdillas necesitan realizar repetidas búsquedas en los casos en que el entorno se actualice con el tiempo, o cuando el conocimiento del robot sobre el entorno cambia con la exploración (como en un problema de SLAM).

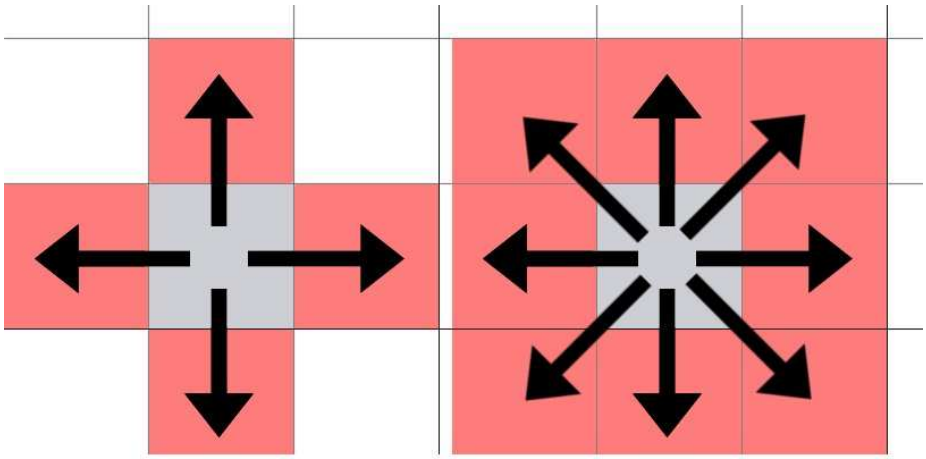

Figura 2.15: A la izquierda, casillas 4-conectadas. A la derecha, casillas 8-conectadas.

2.4. Algoritmos de búsqueda

Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento con ciertas propiedades dentro de una estructura de datos; por ejemplo, ubicar la celdilla de destino en una matriz de posiciones.

Aunque existe un inmenso número de algoritmos de búsqueda, nos centraremos en dos estilos, de los que pueden derivarse todos los demás:

2.4.1. Búsqueda primero en anchura (BFS)

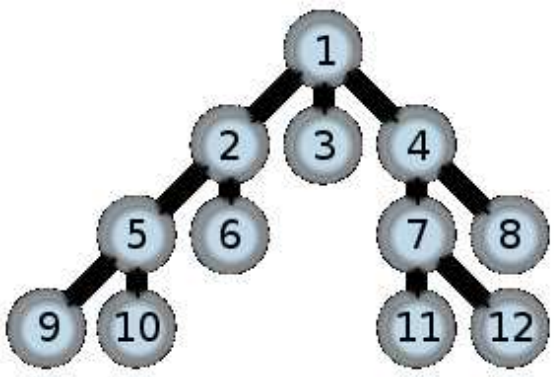

Figura 2.16: Órden en el que se despliegan los nodos (BFS)

BFS es una estrategia de búsqueda en grafos donde la búsqueda está limitada esencialmente a dos operaciones: visitar e inspeccionar un nodo del grafo y obtener acceso para visitar los nodos vecinos del nodo siendo visitado.

La búsqueda primero en anchura comienza en un nodo inicial, o raíz, y desde ahí inspecciona todos sus vecinos. Después, para cada uno de esos vecinos, inspecciona cada uno de sus correspondientes vecinos.

BFS es un algoritmo de búsqueda no informada que intenta expandir todos los nodos de un grafo, buscando sistemáticamente cada una de las posibles soluciones. Es decir, que recorre todos el grafo hasta dar con la solución, pero sin considerarla hasta que la encuentra.

Estos algoritmos pueden ser utilizados para resolver múltiples problemas en la teoría de grafos, pero el más importante en el campo de la robótica es que es capaz de encontrar siempre la distancia más corta entre dos nodos dados. Es decir, BFS siempre devuelve el camino óptimo, aunque no puede garantizarse que encuentre la solución en un tiempo finito.

2.4.2. Búsqueda primero en profundidad (DFS)

DFS es un algoritmo de búsqueda no informada que progresa al expandir el primer ”hijo” del nodo actual y repitiendo recursivamente esta operación hasta encontrar una solución o un nodo ”hoja”, es decir, un nodo sin ”hijos”. En este caso, retrocederá, también recursivamente, hasta encontrar el primer nodo anterior con ”hijos” que expandir.

En el ámbito de la robótica, los algoritmos de búsqueda en profundidad, así como sus variantes informadas, tienen multitud de aplicaciones como la

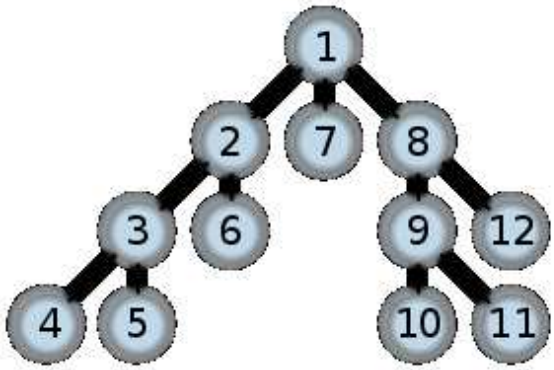

Figura 2.17: Órden en el que se despliegan los nodos (DFS)

búsqueda de nodos conectados (que nos permite determinar si un determinado problema tiene solución) y, fundamentalmente, encontrando el primer camino que une dos nodos.

2.4.3. Búsqueda informada. Algoritmos A*.

Si un algoritmo de búsqueda tiene un cierto conocimiento a priori del problema a resolver, y es capaz de usar esta información para otorgar una cierta prioridad a los nodos que ha de expandir, decimos que se trata de un algoritmo de búsqueda informada.

Por ejemplo, un algoritmo para la resolución de un juego de 3-en-raya dará prioridad a todas las jugadas posibles que consigan otorgarle una una ventaja sobre su adversario. Es decir, las jugadas que le permitan obtener la casilla central serán más prioritarias que las que le permitan obtener alguna casilla de los laterales. Y las jugadas que evitan que el oponente gane serán prioritarias sobre aquellas que le permitan unir dos fichas propias, pero no sobre aquellas que le permitan ganar.

A este conocimiento aplicado lo llamamos heurística, y si cuanto mejor sea, mejores serán los resultados que obtendremos de nuestro algoritmo.

Si podemos garantizar que nuestra estimación heurística del coste de una cierta elección es optimista, es decir, nunca calculamos un valor de la estimación que sea superior al coste real, nuestro algoritmo puede llamarse A* ("A estrella").

Un algoritmo A* es, por tanto, un algoritmo de búsqueda informado con una heurística optimista. Esto nos garantiza que una búsqueda A* expandirá el menor número de nodos de todos los algoritmos de búsqueda y siempre encontrará una solución, aunque esta no tiene por qué ser necesariamente la

solución óptima.

2.5. Historia de los controladores avanzados

Desde el comienzo de la industria del videojuego, la forma de interacción humano-máquina más común ha sido mediante "controladores" o "mandos". Desde sus formas más primitivas hasta las más complejas en la actualidad, el principio básico que todos comparten es la conversión de acciones físicas a señales electrónicas mediante diferentes tipos de sensores[6].

2.5.1. Primeros controladores

Los *"Joysticks"*, dispositivos en forma de palanca, permitían la detección de 8 diferentes movimientos -arriba, abajo, izquierda, derecha y los ángulos intermedios- de manera digital. Generalmente compuestos de una palanca y un único botón de acción, eran dispositivos grandes y pesados.

La Atari 2600 fue la primera consola en utilizar estos controladores. Gracias a que era posible desconectar éstos de la máquina, pronto surgieron otros aparatos que, mediante potenciómetros, eran capaces de producir señales analógicas y, por tanto, detectar la velocidad con la que se movía el *stick*, lo que otorgaba una mayor precisión al control.

Debido al gran tamaño y peso de las palancas, la fuerza ejercida por el jugador causaba roturas en los mandos. Atari decidió solucionar esto reduciendo el tamaño de sus palancas, de manera que pudieran ser accionadas con sólo dos dedos. Esta idea fue copiada por otras compañías, como Bally Astrocade, que sacó al mercado un pequeño mando que podía ser operado con una sóla mano, dada su forma de empuñadura de pistola, con el stick analógico encima y un botón digital a manera de gatillo.

Al mismo tiempo Mattel incorporaba un control en forma de rueda que permitía detectar 16 posiciones analógicas y añadía hasta cuatro botones de acción diferentes, diseño que se implantó en la mayor parte de los controladores de otras empresas, como la famosa Colecovision.

Como respuesta a la competencia, Atari decidió mejorar su tecnología basada en potenciómetros para el lanzamiento de su Atari 5200, permitiendo una detección de 360 grados y añadiendo un pequeño teclado numérico. Por primera vez, se incluye un botón de "pausa" que permite detener el progreso del juego cuando se desea. Lamentablemente, este controlador no tenía la posibilidad de detectar una posición "neutral" o de parada, lo que causaba que el personaje continuara moviéndose siempre en la última dirección introducida.

Algunas consolas posteriores, como la Veltrex, solucionaron este problema. El mando de esta presenta por primera vez la disposición de cuatro botones en una fila con una pequeña palanca a la izquierda, que sería tan común en el futuro.

Figura 2.18: Los primeros controladores.

2.5.2. Controladores durante los 80 y 90

Figura 2.19: Controladores a finales de los años 80.

A comienzos de los ochenta, un inventor japonés llamado Gunpei Yokoi creó una interfaz que más tarde sería conocida como *Digital-Pad*, o *D-Pad*, que consistía en un control digital direccional con forma de cruz que podía ser manipulado con el pulgar.

Nintendo popularizó el D-Pad con la salida al mercado del Sistema de Entretenimiento de Nintendo (NES, por sus siglas en inglés). Este control volvía a permitir únicamente 8 direcciones digitales, e incorporaba dos botones de acción. Le acompañaban dos pequeños botones que servían para pausar el juego y navegar por menús.

Sega, por su parte, decidió modificar ligeramente el diseño del D-Pad para un control más sensible. Ésta fue conocida como *JoyPad*, ya que era un híbrido entre los controles por palanca y el nuevo controlador digital de

Figura 2.20: Controladores a principios de los años 90.

Nintendo. También añadió botones de control extra, tres en un principio que luego se extendieron a seis.

En respuesta a la nueva competencia, Nintendo comenzó a experimentar con otros tipos de control. Sin embargo, para su siguiente consola (Super NES o SNES), el mando era muy similar al anterior. A pesar de ello, se introdujo el diseño que serviría de inspiración para todos los controladores que existirían en un futuro: botones traseros o "gatillo". La idea era utilizar los dedos índice para operar dos botones extra que estaban en la parte trasera del mando, perpendiculares al resto de botones.

2.5.3. Primeros intentos de control avanzado

Es durante estos años de rivalidad entre Nintendo y Sega en los que ambas compañías intentan diferenciarse mediante estilos de control novedosos, que utilizaban sensores y tecnologías más avanzados.

Power Glove

Figura 2.21: Power Glove, de Nintendo.

Por primera vez en la historia, se diseña un artefacto capaz de replicar

los movimientos de una mano, a tiempo real, en una televisión o monitor. El Power Glove es un guante de goma que incorpora un mando de NES en un lateral, así como un teclado numérico.

Usando una tecnología basada en la patente de VPL DataGlove, este controlador puede detectar la posición de la mano gracias a un conjunto de tres micrófonos (situados sobre y a los lados del televisor) y dos pequeños altavoces ultrasónicos incorporados en el mando. Los altavoces se turnan emitiendo pulsos cortos de sonido, que los micrófonos reciben. El tiempo que tarda el sonido en viajar a cada uno de estos micrófonos y un cálculo por triangulación nos permiten conocer la posición X, Y y Z de la mano.

Gracias a unos sensores recubiertos con tinta conductora situados en las articulaciones de los dedos, el guante también es capaz de distinguir con una precisión de 2 bits la posición de cuatro de los 5 dedos (por abaratar costes y complejidad hardware, el meñique se elimina ya que generalmente esta en la misma posición que el anular).

Lamentablemente, debido a las limitaciones tanto de la consola como del guante, este controlador no fue lo suficientemente preciso como para funcionar correctamente y fue un fracaso en ventas.

Sega Activator

Figura 2.22: Activator, de Sega.

Con el propósito de permitir a los jugadores participar con todo su cuerpo en el control del juego, Sega lanzó el Activator.

El Activator consistía de un octágono negro de plástico de aproximadamente un metro de diámetro. El dispositivo debía situarse en el suelo, conectarse tanto a la consola como a una fuente de alimentación externa, y calibrarse. Un techo recto, sin luces directamente encima del controlador, y sin relieves era necesario para poder jugar.

Cada lado del octágono contiene dos emisores de infrarrojos, así como dos receptores. Un haz de luz se proyecta hasta el techo, rebota y es captado por el receptor. El jugador participa situándose en el centro del mando e interrumpiendo los haces con sus extremidades.

Pese a que la idea del controlador era la de replicar el movimiento del jugador en la pantalla, en realidad, cada uno de los 16 haces representaba uno de los botones del mando al que reemplazaba. Ésto, añadido a la falta de juegos diseñados específicamente para el Activator lo convierte en una mala opción de reemplazo del mando original.

NES Zapper

Figura 2.23: Zapper, de Nintendo

La *Zapper* es una pistola de luz para la NES. Se conecta a un puerto para mandos de la consola, reemplazando el mando del segundo jugador.

La pistola consiste en un mecanismo muy simple: un botón hace las veces de gatillo, y dentro del ”cañón” hay un receptor de infrarrojos.

Cuando el gatillo se presiona, la pantalla entera se vuelve negra durante un breve instante. Inmediatamente después, la zona objetivo se vuelve blanca, mientras que el resto de la pantalla permanece negra. Si el receptor en la punta del arma detecta este cambio de luz, el juego registra un “acierto”. El proceso de disparo es lo suficientemente rápido como para que el ojo humano no sea capaz de detectarlo (aunque es posible notar que la pantalla ”parpadea”).

Debido a la naturaleza del efecto detectado por la pistola, sólo es posible utilizarla en pantallas CRT.

2.5.4. Finales de los 90

La era de las consolas de 32 bits comienza cuando Sega saca al mercado su consola Saturn. Ésta adopta la filosofía de Nintendo de usar mandos de reducido tamaño, con gatillos traseros.

Quien traerá las primeras innovaciones será Sony con su consola Play Station y su mando Dual Shock. Por primera vez los mandos tienen un diseño "con cuernos", que permiten al jugador sujetar con comodidad el controlador. El nuevo diseño incluye un D-Pad, 4 botones de acción y cuatro gatillos traseros, así como dos joysticks analógicos en el centro.

En respuesta a estos cambios, Nintendo pone a la venta su Nintendo 64 con su nuevo mando. Éste, como el Dual Shock, también tiene un diseño "con cuernos", tres, en este caso. Incluye un stick analógico para la dirección, un D-Pad, cuatro botones de acción, tres gatillos traseros y, por primera vez, un stick analógico como botón de acción.

También incluye, como novedad, la posibilidad de añadir un accesorio de vibración, también llamada *force-feedback*, idea que sería incluida en posteriores versiones del Dual Shock de Sony.

Sega decide sacar su propia versión de mando renovado, incluyendo dos joysticks (que ahora pasan a llamarse *ThumbSticks*, al poderse manejar con los pulgares) que, además, podían ser presionados como botones tradicionales. Este concepto será utilizado en muchos de los mandos posteriores.

Durante los años siguientes, a pesar de tener una nueva competidora en el mercado (Microsoft) y una generación nueva de consolas, no se producen grandes cambios en el diseño de los controladores.

Figura 2.24: Controladores a finales de los años 90.

Surgen, sin embargo, algunas ideas interesantes de control alternativo.

2.5.5. Nuevos intentos

Plataformas de baile

Figura 2.25: Plataforma de baile para el juego *Dance, Dance, revolution.*

Una plataforma de baile es un controlador plano electrónico utilizado en videojuego para la entrada en los juegos de baile. La mayoría de las almohadillas de baile se divide en una matriz de 3 x 3 de paneles cuadrados para el jugador los pise, con algunos o todos de los paneles correspondientes a las instrucciones o acciones dentro del juego.

Algunas también tienen botones adicionales fuera del área principal de paneles, como por ejemplo "Inicio" y "Seleccionar". Los pares de plataformas de baile a menudo se unen de lado a lado para ciertos modos de juego.

Muchas de estas plataformas de baile emplean sensores de presión, como galgas extensiométricas o sensores piezoeléctricos. Sin embargo, algunas plataformas de baile más avanzadas, llamadas plataformas de estado sólido, utilizan sensores de proximidad. De esta manera se elimina la necesidad de partes móviles, que pueden romperse con el uso.

Eye Toy

Eye Toy es una cámara a color, digital, similar a una webcam en prestaciones. La tecnología usa visión por ordenador y reconocimiento de gestos para procesar las imágenes adquiridas por la cámara. Esto permite a los jugadores interactuar con los juegos usando movimientos, detección de colores y, gracias al micrófono incluido, sonido.

La cámara requiere una habitación bien iluminada para poder jugar correctamente. De no disponer de suficiente luz, se informa al jugador mediante un LED rojo parpadeante situado en la parte frontal.

Figura 2.26: Eyetoy, de Sony

Pistolas de luz

Figura 2.27: Desert Eagle, de Trustmaster.

Dadas las restricciones de las primeras pistolas de luz al trabajar con otras tecnologías que no sean CRT, estos dispositivos han evolucionado para adaptarse a estas limitaciones.

Una de las soluciones empleadas requiere la utilización de uno (o más) emisores de infrarrojos, que la pistola recibe en unos sensores especiales dentro del "cañón" del arma. Al presionar el gatillo, la pistola envía a la consola la intensidad de los receptores que puede recibir. Al ser esta intensidad tanto función de la posición como de la inclinación, un conjunto de acelerómetros se utilizan para poder calcular la inclinación del arma y poder, por tanto, calcular la posición.

Es una evolución de esta tecnología la que se usará en la presente generación de consolas.

2.6. Generación actual

Las tres consolas de la generación actual (Sony Playstation 3, Nintendo Wii, Microsoft Xbox 360) utilizan distintos tipos de controladores avanzados.

Figura 2.28: Mandos a finales de la década pasada.

2.6.1. Nintendo Wii

El Wii Remote tiene la capacidad de detectar la aceleración a lo largo de tres ejes mediante la utilización de un acelerómetro ADXL330. El Wiimote también cuenta con un sensor óptico PixArt, lo que le permite determinar el lugar al que el Wiimote está apuntando; además de agregar una brújula electrónica en el WiiMotionPlus.

A diferencia de un mando que detecta la luz de una pantalla de televisión, el Wiimote detecta la luz de la Barra sensor de la consola, lo que permite el uso coherente, independientemente del tipo o tamaño de la televisión. La barra puede ser colocada por encima o por debajo de la televisión, y debe centrarse. Si está colocada por encima, el sensor debe estar alineado con la parte delantera de la televisión, y si coloca en la parte inferior, debe alinearse con la parte delantera de la superficie de la televisión en la que se coloca. No es necesario señalar directamente a la barra sensor, pero apuntar significativamente fuera de la barra de posición perturbará la capacidad de detección debido al limitado ángulo de visión del Wiimote.

El uso de la barra de sensores permite al Wiimote ser utilizado como un dispositivo de señalamiento preciso de hasta 5 metros de distancia de la barra. El sensor de imagen del Wiimote se utiliza para localizar los puntos de luz de la barra con respecto al campo de visión del Wiimote.

La barra de sensores es necesaria cuando el Wiimote está controlando movimientos arriba-abajo o izquierda-derecha de un cursor en la pantalla del televisor para apuntar a las opciones de menú u objetos como los enemigos en un juego.

2.6.2. Sony Playstation 3

En la Playstation 3 llegó una importante revolución en el aspecto de controladores para consolas de Sony. El mando añadió la función de detección del movimiento. A este primer mando se le llamó Sixaxis (seis ejes), haciendo referencia a los seis ejes de detección de movimiento (3 para movimientos posicionales en el espacio mediante acelerómetros, y 3 para la detección de rotación).

Más tarde, debido a las críticas de los usuarios hacia la falta de vibración, se hizo una revisión del mando con el nombre Dual Shock 3, que añade la función de vibración de nuevo al mando.

Al principio se diseñó un mando con forma de boomerang pero fue abandonado su diseño, volviendo al diseño tradicional del Dual Shock. La diferencia con el Wii Mote es que el mando de la Playstation 3 usa un chip para detectar los vuelcos del mismo, mientras que el mando de la Wii usa un sistema de chip giroscopio para sentir movimientos de desplazamiento, vuelco y posicionamiento al apuntar gracias a los infrarrojos emitidos por una barra que se puede colocar encima o debajo de la televisión.

2.6.3. Microsoft Xbox 360

Figura 2.29: Kinect, de Microsoft.

Finalmente, Kinect para Xbox 360, o simplemente Kinect es un controlador de juego libre y entretenimiento creado por Alex Kipman, desarrollado por Microsoft para la videoconsola Xbox 360, y desde junio del 2011 para PC a través de Windows 7 y Windows 8.

Kinect permite a los usuarios controlar e interactuar con la consola sin necesidad de tener contacto físico con un controlador de videojuegos tradicional, mediante una interfaz natural de usuario que reconoce gestos, comandos de voz, y objetos e imágenes. El dispositivo tiene como objetivo primordial

aumentar el uso de la Xbox 360, más allá de la base de jugadores que posee en la actualidad.

Kinect fue lanzado en Norteamérica el 4 de noviembre de 2010 y en Europa el 10 de noviembre de 2010. Fue lanzado en Australia, Nueva Zelanda y Singapur el 18 de noviembre de 2010, y en Japón el 20 de noviembre de ese mismo año.

Características

El sensor Kinect es una barra horizontal conectado a una base con un pivote motorizado con un rango de movimiento de aproximadamente 30º[7].

Cámara RGB

La captura de imágenes a color es realizada por Kinect mediante una pequeña cámara RGB situada en la parte central del artefacto.

Estas cámaras funcionan utilizando unos sensores capaces de convertir la señal recibida en forma de fotones a una señal electrónica digital que descompone la luz capturada en tres componentes: rojo, verde y azul (o *Red, Green, Blue*, de donde extraemos sus sílabas en inglés).

En el caso de Kinect, el sensor encargado de realizar esta conversión es un sensor dispositivo de carga acoplada o CCD. Estos sensores están compuestos por una matriz de condensadores. El número de condensadores contenidos en el sensor determina su capacidad de resolución, que mediremos en píxeles.

Un sensor CCD está basado en el efecto fotoeléctrico, un fenómeno físico por el cual la luz recibida es convertida en corriente eléctrica en algunos materiales. De esta forma, alterando la composición del material en el que se fabrican cada uno de los condensadores, es posible crear cedldas que reaccionan ante determinadas frecuencias de la luz (en este caso el espectro verde, el rojo y el azul).

De esta forma, para conseguir la conversión de la imagen física en imagen digital, la mayoría de cámaras CCD utilizan una máscara de Bayer que proporciona una trama para cada conjunto de cuatro píxeles de forma que un pixel registra luz roja, otro luz azul y dos píxeles se reservan para la luz verde (el ojo humano es más sensible a la luz verde que a los colores rojo o azul). El resultado final incluye información sobre la luminosidad en cada píxel pero con una resolución en color menor que la resolución de iluminación.

La información de color para cada píxel de la fotografía digital formada se calcula interpolando la señal de dos píxeles verdes, uno rojo y uno azul. De esta manera, aunque se pierde resolución de color, se gana resolución de iluminación, reduciendo el tiempo necesario de exposición del sensor a la luz

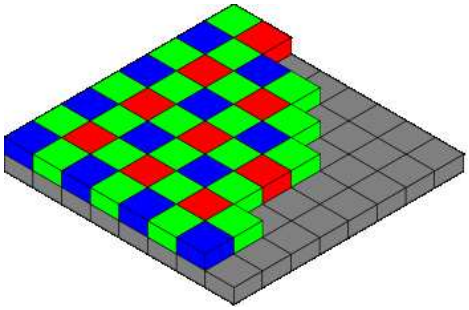

Figura 2.30: Distribucion de colores en un filtro de Bayer.

y, por tanto, el ruido generado por la sobrecarga de los propios condensadores por la temperatura generada por la corriente inducida que los atraviesa.

En particular, la cámara RGB equipada por Kinect tiene una resolución de 640x480 píxeles (VGA) y es capaz de procesar imágenes a una velocidad de 30 fotogramas por segundo, con una resolución de color de 32 bits (1 byte por cada color, 16.7 millones de colores).

Sensor de profundidad

El sensor de profundidad está formado por dos componentes: un generador de láser de infrarrojos y un sensor CMOS monocromo de luz infrarroja. Esta composición permite a la cámara capturar vídeo con información en tres dimensiones bajo cualquier condición de iluminación.

A diferencia de la cámara RGB, el sensor de profundidad emplea tecnología CMOS en su sensor de imagen. La principal diferencia radica en la velocidad de respuesta de un sensor CMOS, permitiendo un tiempo de exposición menor y, por tanto, una mayor sensibilidad a la luz.

La imagen en tres dimensiones es construida capturando la luz infrarroja proyectada por el láser, en forma de parrilla, y calculando la profundidad en base al tiempo que ha tardado la luz en volver al sensor (ver 2.1.3).

El sensor de distancia produce imágenes con una resolución de 320x240 (QVGA) con una información de profundidad de 11 bits, que es traducida a una imagen en escala de grises de 16 bits. Para facilitar la sincronización con la cámara RGB, también emite vídeo a 30 fotogramas por segundo.

En ambos casos, el ancho de campo horizontal está limitado a 57º y el ancho de campo vertical a 43º.

En la figura 2.6.3 se observan las distintas partes que componen el sensor Kinect. Señalado con ”1” se encuentran el emisor (izquierda) y el receptor (derecha) del sensor de infrarrojos. El número ”2” señala la cámara RGB. La

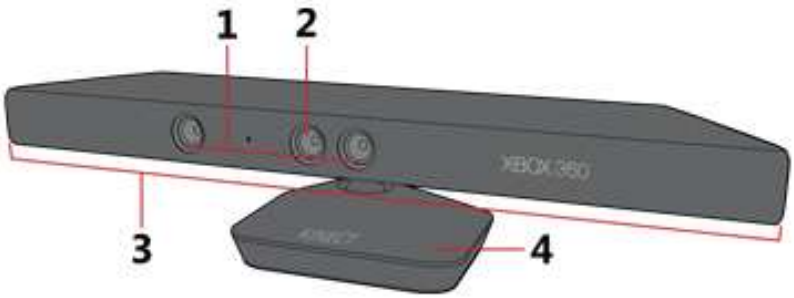

Figura 2.31: Componentes de Kinect

barra y el motor de la base están señalados por ”3” y ”4”, respectivamente.

2.7. Recursos y entornos de programación

Para la realización del proyecto se presentaban dos diferentes opciones en cuanto al *driver* a utilizar para poder interpretar los datos facilitados por el sensor Kinect:

2.7.1. LibFreenect. El conjunto de herramientas MRPT.

LibFreenect es el *driver* para Kinect realizado por el grupo OpenKinect. OpenKinect es una comunidad de gente interesada en hacer uso del hardware de Microsoft en Windows, OSX y Linux. El grupo desarrolla bibliotecas de código abierto y gratuitas que permiten trabajar con el sensor fuera de la consola Xbox 360[8].

La comunidad OpenKinect consta de más de 2000 miembros, que contribuyen al proyecto de la creación del mejor conjunto de aplicaciones para Kinect.

Mobile Robot Programming Toolkit (MRPT) ofrece a los desarrolladores un conjunto extenso, portable y probado de bibliotecas y aplicaciones que cubren las estructuras de datos y algoritmos más comunes empleados en diversas áreas de investigación en robótica.

El conjunto de herramientas incluye bibliotecas para trabajar con visión robótica, SLAM, mapeado, filtros de Kalman y de Montecarlo así como algu-

nos *drivers* para facilitar la interacción con un número de sensores utilizados en robótica. Esto incluye el *driver* freenect.

Todo el código está distribuido bajo la licencia GNU GPL versión 3, así que es posible descargar, modificar y utilizar todos los recursos de forma libre y gratuita.

El proyecto ha sido desarrollado por José Luis Blanco Claraco, entre otros, en el laboratorio MAPIR de la Universidad de Málaga.

Algunas ventajas e inconvenientes de MRPT:

- Completamente código abierto, con la posibilidad de crear y modificar contenido sin inconvenientes.
- Desarrollado en la Universidad de Málaga. Facilidad de contacto con los creadores y desarrolladores de las librerías.
- Documentación extensa y completa de todas las bibliotecas disponibles.
- Lamentablemente, se trata de un proyecto de tamaño reducido y no existen tantas opciones como en otras alternativas.

2.7.2. OpenNI. El conjunto de herramientas ROS.

OpenNI es una organización sin ánimo de lucro formada para certificar y promover la compatibilidad e interoperabilidad entre dispositivos y aplicaciónes que utilizan el paradigma de la interacción natural (NI, *Natural Interaction*)[9].

En el ámbito de la informática, una interfaz natural es el término utilizado por diseñadores y desarrolladores de interfaces humano-máquina para referirse a aquellas que son efectivamente invisibles para el usuario y están basadas en la naturaleza o los elementos naturales.

El término "interfaz natural" se usa en contraposición a las interfaces habituales, que requieren el uso de dispositivos externos artificiales cuya forma de funcionamiento ha de ser aprendida, como, por ejemplo, un ratón. Una interfaz de usuario natural (NUI) se basa en la capacidad del usuario de pasar rápidamente de novato a experto en su manejo. Aunque estas interfaces también requieren un cierto aprendizaje, éste se ve facilitado por el propio diseño, lo que da al usuario la sensación de estar instantánea y continuamente mejorando.

Aunque no forme parte del hardware ni de sus usos originales, Kinect es un dispositivo capaz de ser utilizado en el diseño de interfaces por de gestos, que se conocen como Kinect User Interfaces (KUI, interfaces de usuario con Kinect, en inglés), las cuales forman parte de las interfaces naturales.

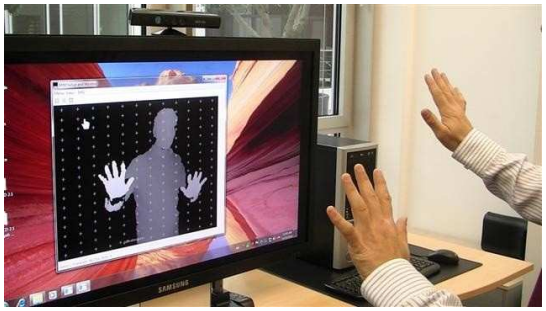

Figura 2.32: Interfaz de ordenador controlada por gestos con Kinect.

Dándose cuenta del enorme potencial del dispositivo de Microsoft en el desarrollo de de interfaces naturales, OpenNI, junto con PrimeSense, la compañia desarrolladora del hardware, y Willow Garage, un grupo de desarrolladores de software Open Source para robótica, crean un *driver* de código abierto para la utilización de Kinect en ordenadores.

La estructura de trabajo (o Framework) OpenNI permite, por un lado, comunicarse con los sensores de audio, video y sensor de profundidad de Kinect, mientras que proporciona una API que sirve de puente entre el hardware del equipo y las aplicaciones e interfaces del S.O. La idea es facilitar el desarrollo de aplicaciones que funcionen con interacción natural, gestos y movimientos corporales.

Actualmente OpenNI permite la captura de movimiento en tiempo real, el reconocimiento de gestos con las manos, el uso de comandos de voz y utiliza un analizador de escena que detecta y distingue las figuras en primer plano del fondo.

Willow Garage son los encargados, desde 2008, del desarrollo del Sistema Operativo para Robótica (ROS). Este proyecto cuenta con la colaboración de más de veinte instituciones.

ROS ofrece los servicios comunes a todos los sistemas operativos, como una capa de abstracción de hardware, control de dispositivos a nivel bajo, intercambio de mensajes entre procesos y manejo de paquetes.

Está basado en una estructura con forma de grafo, donde el procesamiento tiene lugar en cada uno de los nodos, que pueden recibir, enviar y manipular mensajes de otros procesos, sensores, actuadores, órdenes de control, de estado y de planificación, entre otros.

ROS está compuesto básicamente de dos partes: el núcleo del sistema ROS y los paquetes, bibliotecas y programas contribuidos por los usuarios, agrupados generalmente en pilas o stacks.

Los stacks incluidos en una distribución de ROS incluyen funciones como

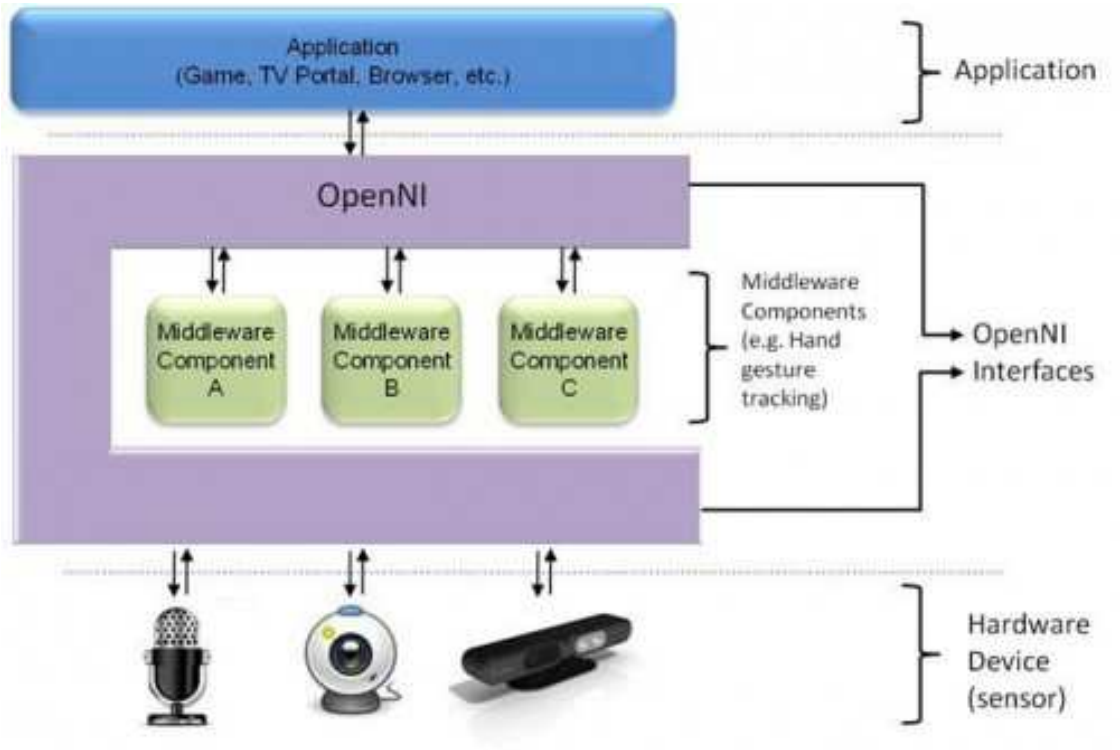

Figura 2.33: Estructura del *framework* OpenNI.

SLAM, planificación, percepción, simulación, etc.

Funcionamiento

Toda distribución de ROS incluye un cierto número de pilas, compuestas por paquetes. Estos paquetes son un conjunto de ficheros que son necesarios para la creación de programas individuales.

Podemos separar los tipos de ficheros en las siguientes categorías:

Paquetes: más formalmente, son las unidades principales de organización de software dentro de ROS. Un paquete puede contener procesos (nodos), bibliotecas dependientes de ROS, conjuntos de datos, ficheros de configuración y, en general, cualquier conjunto de información que sea útil mantener unido.

Manifiestos: Un manifiesto es un fichero en formato xml que contiene los metadatos correspondientes a cada paquete, como su información de licencia, sus dependencias o sus flags de compilación.

Pilas: Una pila es un conjunto de paquetes que ofrecen una funcionalidad conjunta, como podría ser la pila de navegación. Ésta es la forma común de distribución de software en ROS (similar a los paquetes en las distribuciones de Linux).

Tipos de mensajes: Definen la estructura de los datos contenidos en los mensajes que utilizará el paquete para comunicarse con el sistema operativo.

Tipos de servicios: Definen las estructuras de datos para las peticiones y respuestas de aquellos procesos que actúen como servicios en ROS. Un servicio es todo nodo que puede recibir una conexión de otro nodo, y que sólo actúa como respuesta a una petición de un nodo externo, de manera muy similar al paradigma cliente-servidor en telecomunicaciones.

A nivel de ejecución, el sistema operativo divide la información y los procesos en las siguientes categorías:

Nodos: los nodos son procesos que realizan algún tipo de cálculo. Gracias a su estructura granular, el sistema de control de un robot generalmente estará compuesto de diversos nodos individuales comunicándose entre sí, como hilos de ejecución.

Maestro: El nodo maestro de ROS se encarga de gestionar la comunicación entre nodos. Sin él, los procesos no serían capaces de encontrarse los unos a los otros, intercambiar mensajes o invocar servicios.

Servidor de parámetros: Permite almacenar de manera segura datos en una ubicación central, forma parte del nodo Maestro.

Mensajes: Los nodos se comunican entre sí pasándose mensajes. Un mensaje es simplemente una estructura de datos, como las de C, que se rellenan apropiadamente.

Temas (*topics*): La distribución de mensajes se realiza cuando un nodo publica un mensaje en un *topic*. Todos aquellos nodos que deseen recibir dichos mensajes se subscribirán a dicho *topic*. De esta manera, la comunicación emula la comunicación en bus, separando la generación de contenido de su consumición.

Servicios: Descritos más arriba. La comunicación con éstos se realiza mediante conexiones y el intercambio directo de mensajes.

De esta forma, el nodo maestro actúa como un servidor de nombres (como lo sería un servidor de DNS). Este nodo almacena los *topics* y servicios, así como la asociación entre nodos. Los nodos se registran con el nodo maestro antes de comenzar su ejecución. Como se comunican directamente con el nodo maestro, pueden recibir información sobre otros nodos registrados y crear conexiones si fuera necesario.

Los nodos se conectan los unos a los otros directamente, el maestro sólo otorga la información de contacto. Los nodos que se subscriben a un *topic* pedirán conectarse a los nodos que producen ese *topic*. Dicha conexión se establece mediante un protocolo de comunicación, en este caso TPCROS, una implementación del protocolo TCP/IP.

Bibliotecas destacadas

ROScpp Roscpp es, en esencia, el API que los desarrolladores de C++ deben utilizar para comunicarse con ROS. Es la biblioteca más utilizada en ROS y para la que existe la mayor colección de paquetes y pilas.

Las bibliotecas internas de ROScpp se dividen en cuatro paquetes principales:

- cpp_common: contiene código en C++ que no está directamente relacionado con ROS.
- roscpp_serialization: permite a los programadores implementar las funciones de serialización de mensajes para facilitar su almacenamiento en memoria.
- roscpp_traits: implementa las opciones de búsqueda internas de información dentro de los mensajes. Su principal función es la de convertir los tipos estándar de C++ en campos de mensaje de ROS.
- rostime: contiene todas las funciones para el manejo del tiempo en ROS.

std_msgs ROS usa un lenguaje simple para describir los valores de los datos almacenados en los mensajes que los nodos de ROS intercambian entre sí. Esta descripción facilita el trabajo automático de generación de tipos de datos en distintos lenguajes objetivo.

Std_msgs es la librería que permite la fácil conversión entre las primitivas básicas de ROS y los tipos de datos de los diferentes lenguajes de programación.

Los tipos de datos soportados por ROS incluyen: valores *booleanos*, *bytes*, caracteres, números enteros y en coma flotante de diversas longitudes, cadenas de caracteres y cualquier registro o combinación de los anteriores.

tf tf es un paquete que permite al usuario monitorizar y controlar múltiples ejes de coordenadas y su evolución en el tiempo. tf crea una estructura jerárquica en forma de árbol y ofrece la posibilidad de realizar transformaciones de puntos, vectores, etc, entre estos distintos marcos de referencia.

Generalmente, un sistema robotizado tiene muchos ejes de coordenadas que cambian con respecto al tiempo (un buen ejemplo serían las articulaciones de un brazo mecánico). Este paquete proporciona los medios al usuario para mantener de forma ordenada toda esta información, así como realizar diferentes consultas al sistema sobre ellas.

Los datos para cada nodo del árbol de transformaciones se almacenan en diferentes estructuras de datos definidas en la biblioteca tf. Estas estructuras incluyen: cuaterniones, vectores tri-dimensionales, puntos tri-dimensionales y transformadas.

Las transformadas contienen información para generar las matrices de transformación entre cada eje de coordenadas almacenado y el nodo raíz. Las transformaciones entre nodos se realizan transformando ambos sistemas de referencia al del nodo raíz y luego realizando la operación inicial.

tf permite, también, la visualización de los distintos ejes de referencia en programas como rviz o imprimir el árbol jerárquico en formato PDF.

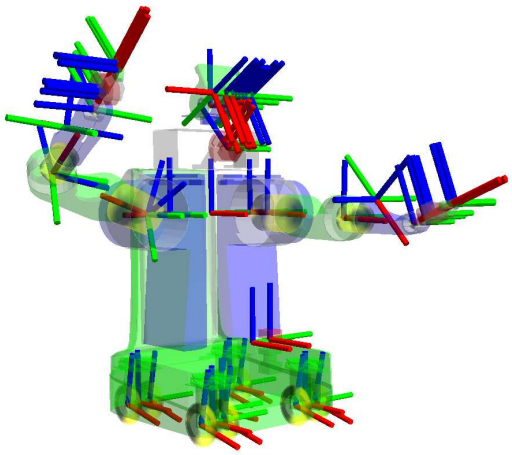

Figura 2.34: Múltiples ejes de coordenadas en un robot.

PCL *Point-Cloud Library*[10] es un proyecto independiente a ROS, que contiene numerosos algoritmos incluyendo el filtrado, cálculo de referencias, reconstrucción de superficies, reconocimiento de modelos y segmentación de imágenes. Esto algoritmos pueden usarse, por ejemplo, para extraer datos

anómalos en poblaciones con ruido, unir distintas nubes de puntos, segmentar partes relevantes de una imagen e identificar y extraer *landmarks* del mundo real para la navegación con robots. También es posible convertir esta información matemática en imágenes y se ofrecen diversos formatos y aplicaciones con los que visualizarlos.

Las nubes de puntos son estructuras de datos que representan una colección de puntos multidimensionales, y son usadas comúnmente en la representación de información en tres dimensiones.

En una nube de puntos 3D, los puntos generalmente representan las coordenadas geométricas X, Y y Z de una superficie muestreada.

Las nubes de puntos pueden adquirirse a través de sensores como las cámaras estereoscópicas, escáners 3D o generadas artificialmente en un programa de ordenador. PCL soporta nativamente las interfaces proporcionadas por OpenNI, como el *driver* de Kinect.

La información de un sensor se almacena en una estructura de datos definida en el paquete sensor_msg de ROS llamada PointCloud. Entre otros datos, se almacenan las coordenadas X, Y y Z y el color correspondiente a ese punto, siempre que sea posible. A este tipo de nubes se las llama nubes de puntos coloreadas.

Aunque la información de la profundidad extraída por el sensor de Microsoft no contiene información de color, es posible realizar una transformación entre el sistema de referencia del sensor de profundidad y la cámara RGB y asignar a cada punto un píxel de la imagen bidimensional, extrayendo así la información de color correcta.

Figura 2.35: Varias estructuras 3D formadas a partir de nubes de puntos.

OpenNI_Kinect OpenNI contiene todo el código necesario para realizar la comunicación con el dispositivo, así como la conversión de los datos obtenidos de los sensores a los distintos formatos ofrecidos por ROS, como nubes

de puntos o imágenes planas.

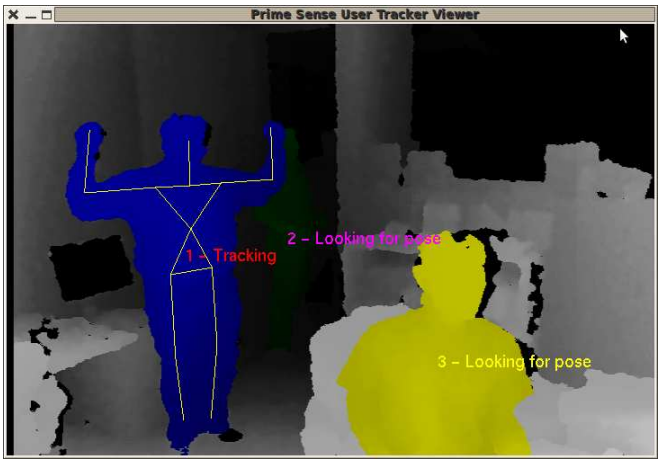

Figura 2.36: NITE en funcionamiento.

Además del *driver*, el paquete incluye una implementación en ROS de la biblioteca NITE, que incluye detección de poses, gestos y seguimiento de "esqueletos" humanos. OpenNI_tracker añade a esta funcionalidad ventajas propias de ROS como el uso de tf para publicar las posiciones de cada una de las articulaciones detectadas.

Figura 2.37: Dispositivos que cumplen el estándar OpenNI.

2.8. Caso de estudio: RGBDSLAM

RGBDSLAM es un programa desarrollado para ROS que es capaz de registrar las nubes de puntos generadas por sensores como Kinect o cámaras estereoscópicas y crear mapas en forma de nubes de puntos densas coloreadas.

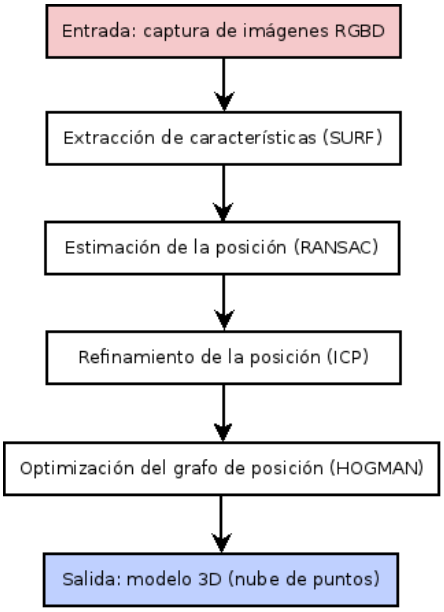

Figura 2.38: Esquema de ejecución de RGBDSLAM.

El algoritmo utilizado en esa solución el problema del SLAM consta de cuatro pasos. Primero, se extraen las características y *landmarks* de la información obtenida por el sensor utilizando un algoritmo llamado SURF (ver 2.10). Después, se comparan estas características contra las extraídas en iteraciones anteriores. Una vez obtenido el conjunto de correspondencias, se utiliza un algoritmo RANSAC (2.2.5) para estimar la transformación relativa entre los pares de imágenes obtenidas.

El tercer paso consiste en una refinación de esta estimación usando una implementación del algoritmo ICP (ver 2.11).

Estas estimaciones de la posición basadas en la comparación de parejas de fotogramas no es necesariamente consistente con la representación global del mapa, así que se realiza un cuarto paso en el que se optimiza el grafo global utilizando un algoritmo de resolución de grafos llamado HOGMAN (ver 2.9).

RGBDSLAM permite a un robot generar modelos en tres dimensiones del espacio en el que se desplaza. Aunque los autores también consideran que aplicaciones fuera del campo de la robótica son posibles, como el escaneado en tres dimensiones de viviendas para su posterior modelado y utilización por diseñadores de interiores.

2.9. HOGMAN

Como en nuestro caso de estudio, otros algoritmos utilizan una representación del problema del SLAM basada en grafos. En este tipo de formulaciones, la diferentes posiciones del robot se modelan como nodos en un grafo. La información de las restricciones espaciales calculadas a partir de la odometría y de las observaciones del entorno se codifican en las aristas que conectan los nodos.

En este tipo de soluciones, el problema del SLAM se divide en dos ramas independientes: la extracción e identificación de las restricciones mediante los sensores disponibles, lo que se conoce como *front-end*, y, una vez obtenidas estas restricciones, el cálculo de la posición más probable y de la incertidumbre de este cómputo se realiza en el motor de optimización o *back-end*.

La solución a la primera rama se explica en 2.2.3, HOGMAN es el algoritmo usado para la segunda.

Para garantizar un equilibrio razonable entre coste computacional y fiabilidad de la respuesta del algoritmo, HOGMAN utiliza una representación del espacio del problema (en este caso, el grafo de posiciones y sus restricciones) en diferentes niveles. Utilizando una representación no-euclídea del espacio tridimensional como simplificación del modelado del entorno, el algoritmo consigue un modelo susceptible a ser dividido en capas. La complejidad del modelo (y su similitud con el problema original) avanza a medida que profundizamos en esta estructura en niveles, como se puede observar en 2.9.

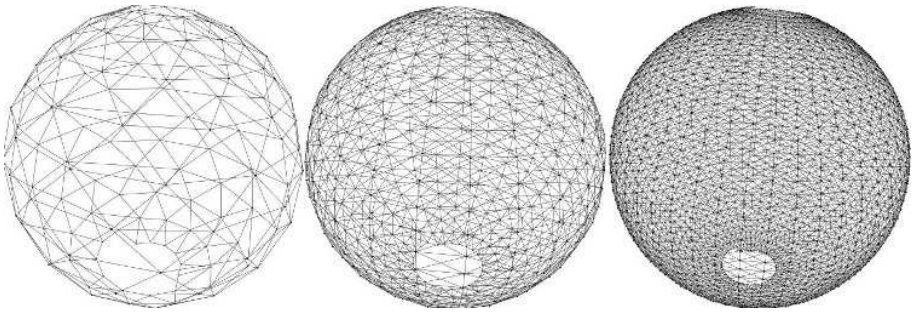

Figura 2.39: Estructura en capas de un grafo.

Este concepto permite estimar las rotaciones en tres dimensiones usando un modelo de Gauss-Newton, lo que simplifica el cálculo de las transformaciones en comparación a modelos más tradicionales, como el de ángulos de Euler.

El algoritmo se comporta de manera "perezosa" para optimizar el tiempo de cálculo, de manera que los cambios en cualquier nivel sólo se extienden

a las zonas del mapa que están directamente conectadas, sin re-optimizar todos los nodos del grafo en cada iteración. Así, cada vez que se obtiene una nueva observación, sólo el nivel superior -el más simplificado- necesita ser optimizado por completo. Cuando el mapa superior se cambia, sólo las regiones necesarias de los niveles inferiores reciben cambios. De esta manera, se conserva la consistencia global mientras que el coste computacional se reduce al mínimo posible.

El objetivo que persigue el algoritmo HOGMAN es el de obtener, de entre todas las posibles, la posición (nodo del grafo) que maximice la correspondencia con la información sensorial obtenida en el *front-end*. Para esto, los creadores del algoritmo[11] han adaptado algoritmos conocidos (Gauss-Newton y Levenberg-Marquardt ([12])) de minimización de errores para trabajar en su modelado del entorno como un espacio no-Euclideano.

2.10. SURF

SURF (*Speeded-Up Robust Features*, extracción acelerada de características) es un algoritmo de extracción y descripción de características, independiente de la escala y la rotación. SURF es capaz de aproximarse a (o, bajo ciertas condiciones, superar) las prestaciones de otros algoritmos similares en repetibilidad, diferenciación y robustez, aunque es posible computar sus cálculos mucho más rápido[13].

Esto se consigue mediante técnicas estadísticas de tratamiento de imágenes, en particular matrices Hessianas para las medidas del detector y un descriptor basado en muestreos aleatorios.

Para localizar puntos de interés en una imagen se utiliza un filtro basado en matriz Hessiana para identificar los sectores donde el determinante de este filtro es máximo (filtros hessianos). De esta forma se consigue, al eliminar las zonas de la imagen que no se maximizan por el filtro, extraer una máscara para la imagen original que contiene la información relativa a los puntos de interés almacenados en estructuras con forma de "burbuja".

Una vez extraída esta información, el siguiente paso consiste en asignar a cada punto de interés una "orientación", de manera que se facilite la identificación y correspondencia entre *landmarks* cuando las imágenes capturadas presentan una rotación, como ocurre durante el desplazamiento de un robot, por ejemplo.

Esta forma de computación permite obtener información robusta y fiable con una gran mejora en cuanto a velocidad con respecto a otras alternativas, aun sin tener en cuenta posibles optimizaciones para plataformas particulares. Esto hace de este algoritmo una elección perfecta para aplicaciones

Figura 2.40: Puntos de interés extraidos y sus orientaciones.

empotradas con extracción de características online, como un algoritmo de SLAM.

2.11. ICP

Iterative Closest Point (punto más cercano iterativo) es un algoritmo empleado para minimizar las diferencias entre dos nubes de puntos. ICP es comúnmente utilizado para reconstruir superficies tridimiensionales a partir de diferentes capturas de imagen, para la localización de robots, y para conseguir una planificación de trayectorias óptima.

La estructura del algoritmo, en todas sus versiones, es bastante simple:

- Se asocian los puntos siguiendo el criterio del vecino más cercano.
- Se estima la transformación entre nubes mediante una función de mínimos cuadrados.
- Se transforman los puntos utilizando los parámetros estimados en los puntos anteriores.
- Se repite el proceso, iterativamente.

Una de los problemas de este algoritmo es que es frecuente la acumulación de errores entre iteraciones, lo que puede provocar fallos en los algoritmos de mapeado.

En particular, RGBDSLAM utiliza la implementación proporcionada por la biblioteca PCL que combina las mejoras de dos algoritmos (*standard-ICP* y *point-to-plane ICP*) y que ha probado, mediante pruebas en entornos simulados y reales, que es capaz de ofrecer un rendimiento temporal superior

al de ambos algoritmos por separado y, a su vez, ser más robusto frente a los errores causados por las correspondencias incorrectas entre puntos.

Este nuevo enfoque recibe el nombre de ICP generalizado, y puede encontrarse una descripción detallada del algoritmo, así como las pruebas a las que ha sido sometido, en [14].

2.12. Entorno de desarrollo

Gracias a que ROS ofrece un alto grado de integración con el entorno de desarrollo Eclipse y dada la probada utilidad de éste, así como sus potentes herramientas de compilación y depuración, se ha elegido esta plataforma para realizar el desarrollo del proyecto.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Hoy en día, lo desarrolla la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios

Líneas de código fuente	2.063.083
Esfuerzo estimado de desarrollo (persona-año / persona-mes)	604,33 / 7.251,93
Estimación de tiempo (años-meses)	6,11 / 73,27
Estimación del nº de desarrolladores en paralelo	98,98
Estimación de coste	$ 81.636.459

Figura 2.41: Estado actual del proyecto.

El entorno de desarrollo integrado (IDE) de Eclipse emplea módulos (en inglés plug-in) para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este mecanismo de módulos es una plataforma ligera para componentes de software. Adicionalmente a permitirle a Eclipse extenderse usando otros lenguajes de programación como son C/C++ y Python, permite a Eclipse trabajar con lenguajes para procesado de texto como LaTeX, aplicaciones en red como Telnet y Sistema de gestión de base de datos. La arquitectura plugin permite escribir cualquier extensión deseada en el ambiente, como sería Gestión de la configuración. Se provee soporte para Java y CVS en el SDK de Eclipse.

Y no tiene por qué ser usado únicamente para soportar otros lenguajes de programación..

Eclipse dispone de un Editor de texto con resaltado de sintaxis. La compilación es en tiempo real.Cuenta con asistentes (wizards) para creación de proyectos, clases, tests, etc., y refactorización del código.

Es posible extender sus funciones (mediante plugins) para controlar versiones mediante SVN, así como trabajar con lenguajes de abstracción de bases de datos, como Hibernate.

A continuación se muestra una tabla resumen de la situación actual del software.

Lenguaje	Líneas de código	%
Java	1.911.693	92,66%
ANSI C	133.263	6,46%
C++	10.082	0,49%
JSP	3.613	0,18%
sh	2.066	0,10%
perl	1.468	0,07%
php	896	0,04%
sed	2	0,00%

Figura 2.42: Estimación del volumen de código desarrollado con Eclipse.

Un punto muy importante a notar son los diversos lenguajes de programación utilizados en el desarrollo del proyecto, de acuerdo al análisis realizado usando SLOCCount, el lenguaje más utilizado es Java, seguido de ANSI C.

Capítulo 3

Desarrollo práctico

Una vez comprendidos los principios teóricos en los que se basa el proyecto, se procede a exponer el trabajo desarrollado.

3.1. Preámbulos y configuraciones

3.1.1. Instalación del Sistema Operativo

Dado que se ha decidido que la plataforma de desarrollo de este proyecto será ROS, es necesario preparar el equipo disponible en el laboratorio para tal fin. Ya que el soporte completo para ROS sólo se ofrece para el sistema operativo Ubuntu, la instalación de éste es el primer paso que se debe dar.

Entre las múltiples opciones disponibles, el instalador para Windows Wubi (Windows-Based Ubuntu Installer) resulta la opción más cómoda. Wubi crea una instalación que no necesita una partición física en el disco duro, sino que reserva un espacio virtual dentro del mismo. De esta forma, la instalación del sistema operativo funciona de manera similar a como lo haría una máquina virtual, pero sin todas las complicaciones asociadas a la utilización de éstas.

El primer paso para la instalación es dirigirse a la página del instalador en la web de Ubuntu (`http://www.ubuntu.com/download/`) y descargar el programa. Una vez lanzado, el programa nos ofrece tres opciones:

Figura 3.1: Diálogo inicial de Wubi.

De ellas, elegimos ”instalar dentro de Windows”. A continuación, se nos ofrece una lista de opciones de instalación. Se debe elegir correctamente el nombre de la unidad de disco duro en la que está instalado el gestor de arranque de Windows (en una instalación normal, C:), el tamaño de la partición virtual, con suficiente espacio para poder trabajar con comodidad y el entorno de escritorio deseado de entre los disponibles: Ubuntu, Xubuntu o Kubuntu.

Una vez seleccionados estos parámetros, se escoge el lenguaje de instalación, el nombre de usuario para la instalación y su contraseña.

Figura 3.2: Diálogo principal de la instalación.

Si el instalador detecta una imagen ISO del sistema operativo en su mismo directorio, comienza el proceso de preinstalación que concluye cuando se muestra el siguiente diálogo:

Figura 3.3: Instalación finalizada.

Cuando se reinicia el PC, se muestra el gestor de arranque de Microsoft con las opciones Windows 7 (o el sistema operativo instalado anteriormente) y Ubuntu. Elegir esta segunda opción lanzará el gestor de arranque de Ubuntu (GRUB) que permite elegir qué versión de las disponibles se quiere lanzar.

Este programa permite una rápida desinstalación de Ubuntu si fuera necesario. Para Windows 7 el proceso es el siguiente:

Figura 3.4: Panel de control de Windows.

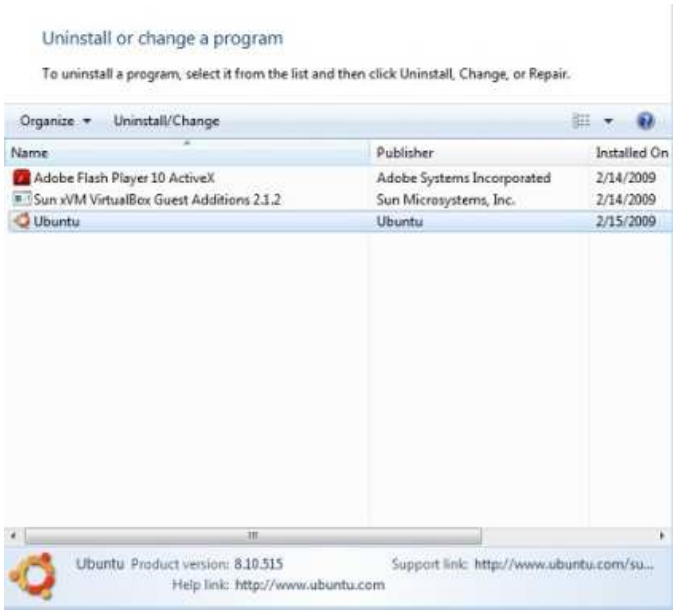

Figura 3.5: La instalación se selecciona como un programa normal.

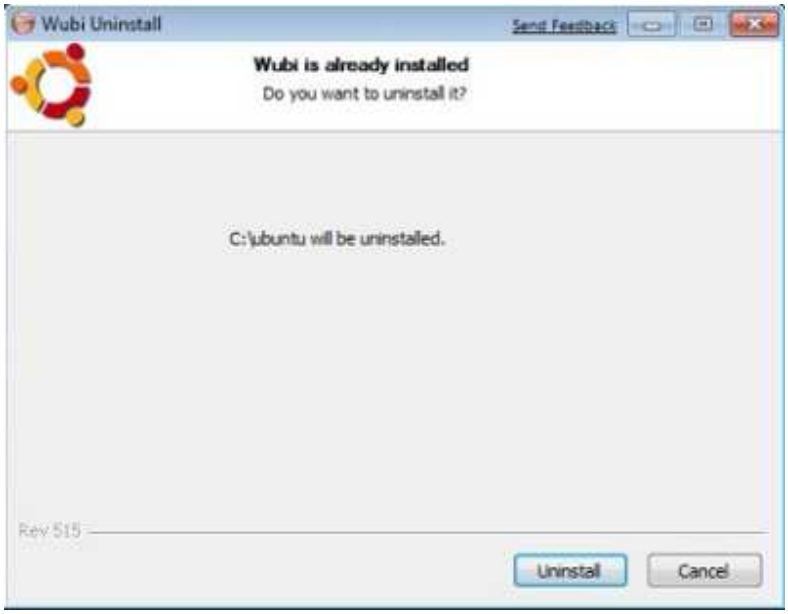

Figura 3.6: Diálogo de confirmación

Figura 3.7: La desinstalación ha sido completada.

3.1.2. Instalación del *framework* ROS

Antes de proceder con la instalación de ROS, es necesario preparar el gestor de software de Ubuntu para aceptar los paquetes de instalación de los repositorios adecuados. El proceso de instalación de ROS indica que es necesario activar los repositorios "main", "universe", "restricted" y "multiverse".

Figura 3.8: Selección de los repositorios.

También es necesario agregar el repositorio propio de ROS, que depende de la versión a instalar y de la versión del sistema operativo destino. Una vez escogido el correcto, puede agregarse en el siguiente diálogo:

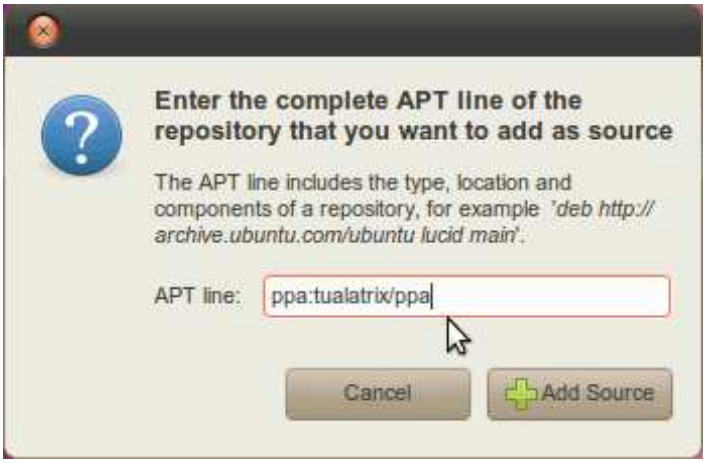

Figura 3.9: Agregar un nuevo repositorio.

En este punto, Ubuntu se encuentra preparado para comenzar la instalación de ROS. De las tres disponibles, la opción "desktop-full" es la recomendada, así que introducimos la siguiente orden en la línea de comandos:

```
sudo apt-get install ros-electric-desktop-full
```

Para facilitar la instalación de los demás paquetes que se necesitarán, es recomendable obtener las herramientas *rosinstall* y *rosdep*, utilizadas para automatizar la instalación de "*stacks*" de ROS y para resolver sus dependencias, respectivamente.

```
sudo apt-get install python-rosinstall python-rosdep
```

La instalación de ROS debería ser totalmente funcional una vez realizados estos pasos.

3.1.3. Instalación de las dependencias necesarias para el proyecto

Antes de proceder con la instalación de RGBDSLAM, es necesario instalar las dependencias del paquete, así como estudiar su comportamiento para la

posterior utilización en el proyecto a desarrollar.

Driver OpenNI_Kinect

La instalación del *driver* de Kinect en ROS se completa con dos simples operaciones en la línea de comandos:

```
sudo apt-get install ros-electric-openni-kinect rosmake openni_kinect
```

Esto descarga, compila e instala los tres componentes del *driver*: openni_kinect para conectar y comunicarse con la cámara, openni_camera para obtener la información de las cámaras del sensor y NITE, una biblioteca externa para el reconocimiento de "esqueletos" y la identificación de personas y gestos, que no son utilizados en este proyecto.

Si se lanza el *driver* a ejecución mediante la orden

```
roslaunch openni_camera openni_node
```

Y se comprueba la lista de *topics* publicados con

```
rosnode info /openni_node
```

Se observan los siguientes *topics* a los que es posible suscribirse:

- camera/rgb/camera_info
- camera/rgb/image_raw
- camera/depth/camera_info
- camera/depth/image_raw
- camera/depth/points

El último de estos *topics* es el de mayor interés para este proyecto ya que devuelve una conversión de los datos recibidos por el sensor de profundidad al espacio tridimensional, centrado en la cámara. El resultado es una nube de puntos coloreada en función de la profundidad detectada por el sensor, con un color de negro asignado a aquellos pixels de la imagen que no ofrecen información relevante, ya que se encuentran demasiado lejos de la cámara o demasiado cerca.

Biblioteca tf

Aunque tf es uno de los stacks que se instalan junto con ROS, es necesario construir el paquete antes de poder utilizarlo. Esto se consigue con la siguiente orden

```
rosmake tf
```

Esta biblioteca permite acceder a ciertas herramientas que serán de utilidad durante el desarrollo del proyecto, las más destacables son:

- *tf_monitor*. Permite conocer información sobre el árbol de transformaciones de los procesos en ejecución.
- *tf_echo*. Cuando se acompaña esta orden de dos nodos del árbol de transformadas, tf_echo vuelca la información relativa a la transformación entre ambos. En concreto, devuelve la traslación y la rotación tanto en raw/pitch/yaw como en cuaterniones.

Estudio de las opciones del framework para el simulado

Toda la información proporcionada por tf es susceptible de ser interpretada por rviz, un entorno de visualización 3D para robots usando ROS, y es posible utilizar una versión simplificada, llamada *turtlesim*, que representa un conjunto de robots moviéndose en el plano XY como tortugas desplazándose por un estanque.

Esta herramienta, en principio simple, permite comprender cuál es la correspondencia entre los ejes de movimiento y los movimientos reales de la cámara, por ejemplo.

Para instalar *turtlesim* se necesita tener previamente instalado tf e instalar rviz. Esto puede solucionarse con las siguientes líneas de comandos:

```
rosdep install turtle_tf rviz rosmake turtle_tf rviz
```

Se puede comprobar el funcionamiento correcto de la instalación ejecutando el programa completo que se desarrollará en los tutoriales (`ros.org/wiki/tf`). En éste, un "robot", representado por una tortuga, recorre una trayectoria aleatoria, mientras que un segundo robot lo sigue a una corta distancia. El algoritmo utilizado por la segunda tortuga es una simplificación del algoritmo de persecución pura, por lo que las trayectorias no coinciden exactamente.

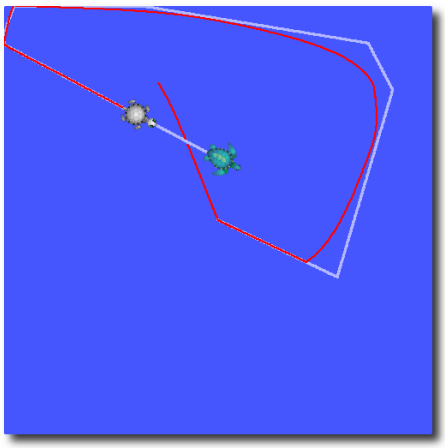

Figura 3.10: Ejecución ejemplo del programa *turtlesim*

En la figura 3.1.3 se puede observar la estructura en forma de árbol que forman los distintos nodos publicadores con respecto a las coordenadas comunes del ”mundo”, en este caso, el centro de la pantalla.

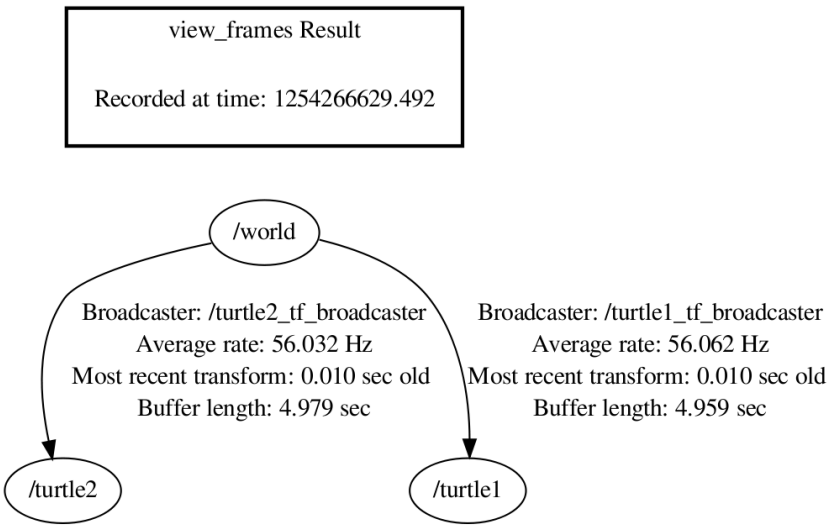

Figura 3.11: Estructura en árbol del conjunto de ejes que participan en *turtlesim*

3.1.4. Instalación del programa RGBDSLAM

Aunque existe una versión disponible de RGBDSLAM en `www.openslam.org`, el autor recomienda utilizar subversion para obtener la versión constantemente actualizada del proyecto. Una vez obtenido el código y añadido al PATH de ROS, es necesario descargar dos dependencias por separado HOGMAN-minimal y g2o[15].

Cuando los tres paquetes han sido añadidos al PATH de ROS, una orden en la consola de comandos permite instalar los tres paquetes y sus dependencias principales:

```
rosmake –rosdep-install hogman g2o rgbdslam
```

Una vez este proceso finaliza (puede tardar unos veinte minutos en completarse) y con el sensor Kinect conectado correctamente al PC y la alimentación, puede lanzarse a ejecución el programa:

```
roslaunch rgbdslam kinect+rgbdslam.launch
```

La orden utilizada se asegura de lanzar el núcleo de ROS, los *drivers* de Kinect y RGBDSLAM para el correcto funcionamiento del programa.

En la imagen 3.1.4 se observan cuatro secciones diferentes en pantalla principal:

- La imagen superior muestra el mapa generado. El mapeado toma la forma de una nube de puntos coloreada que combina la información de la cámara RGB y del sensor de distancias para asignar un color a cada punto con información tridimensional. La propia cámara puede encargarse de publicar la correlación entre ambas imágenes, corrigiendo el desplazamiento que existe entre ambos sensores, si se configura para publicar nubes de puntos registradas antes de lanzar el programa.

  Esta imagen contiene también el grafo que representa las distintas posiciones estimadas del robot, y su recorrido por dentro del mapa.

- La imagen izquierda de la zona inferior corresponde al stream de datos recibido de la cámara RGB.

- La imagen central corresponde al stream de datos de la cámara IR. Los colores en escala de grises codifican la distancia detectada.

- La imagen de la derecha muestra los puntos de interés extraídos, con la información del algoritmo RANSAC 2.2.5 superpuesta. El tamaño de los círculos representa el grado de confianza en la estimación (menor cuanto mayor es el grado) y la flecha muestra la orientación estimada en dicho algoritmo.

Figura 3.12: Captura de una ejecución del programa RGBDSLAM

El mapa tridimensional permite el control del punto de vista y del zoom y la posición de la cámara. También es posible, una vez detenida la captura de imágenes, exportar la nube de puntos final para su posterior visualizado y exploración en cualquier programa con soporte para PCL, como rviz.

3.1.5. Estudio de RGBDSLAM

Una vez comprobado el correcto funcionamiento de la instalación, se procede a analizar las posibilidades de la utilización de este algoritmo en la navegación de robots móviles.

Utilizando los métodos vistos en 3.1.3 y 3.1.3, así como la documentación que acompaña al programa y la información obtenida al contactar con el autor, se obtienen las siguientes conclusiones:

- No es posible obtener información sobre el mapa de manera online. El tamaño de la nube de puntos generada hace imposible el publicado de la misma durante la ejecución del algoritmo y por tanto no puede utilizarse como referencia para la navegación. La solución a este problema se encuentra en 3.2.5.

- Sí es posible conocer en tiempo real la posición actual del robot con respecto a una posición inicial, considerada (0, 0, 0). La relación entre ambas se publica mediante tf, y es posible suscribirse a los *topics* /map y /openni_camera para obtener la transformada entre ambas.

- Para poder obtener un equilibrio entre exactitud y rendimiento, el algoritmo basa sus cálculos no en toda la historia de capturas obtenidas (a un ritmo de 20/30 capturas por segundo) sino en las últimas diez. Esto, sumado al hecho de que el backend utilizado 2.2.5 no puede detectar rotaciones inferiores a 10º ni traslaciones inferiores a 5 centímetros de manera consistente, limita la velocidad de rotación a 20º/s y la de traslación a 0.4m/s.

3.1.6. Instalación del entorno de desarrollo Eclipse y adaptación del proyecto

Para poder comenzar con el desarrollo del proyecto es necesario instalar el entorno de desarrollo (IDE) Eclipse. Aunque Ubuntu incluye una instalación del entorno por defecto, las instrucciones de ROS especifican explícitamente que es necesario utilizar una versión separada descargada desde la web de Eclipse. La versión escogida será la última disponible para desarrolladores de C/C++.

Una vez descargada y descomprimida la versión, es necesario crear un paquete de ROS en el que comenzar el desarrollo y conseguir importarlo correctamente en Eclipse. Hay varias formas de realizar este trabajo (Cmake es la más utilizada), pero la forma más simple resulta ser la creación de un nuevo paquete vacío en ROS mediante el comando

```
roscreate-pkg [nombre del proyecto]
```

Si, como en el caso de este proyecto, las dependencias se conocen de antemano también puede modificarse la orden para incluirlas, aunque siempre es posible modificar el manifiesto en la carpeta del proyecto para agregarlas a mano.

Por ejemplo, para la creación de este proyecto la orden completa sería:

```
roscreate-pkg PFC_ROS rospy roscpp tf pcl pcl_ros sensor_msgs std_msgs
```

Y la lista de dependencias en el manifiesto sería

```
<depend package="rospy"/>
<depend package="roscpp"/>
<depend package="tf"/>
<depend package="pcl"/>
<depend package="pcl_ros"/>
<depend package="sensor_msgs"/>
<depend package="std_msgs"/>
```

Esto crea la estructura de carpetas adecuada para que ROS pueda indexar correctamente el paquete. Una vez hecho esto, se comprueba el correcto funcionamiento del paquete intentando cambiar al directorio que lo contiene usando roscd en la consola de comandos.

Si todo ha funcionado correctamente, en este punto es posible crear un archivo vacío (llamado main.cpp, por ejemplo) e introducirlo en la carpeta "src". Luego, debido a ciertas incompatibilidades entre el IDE y ROS, es necesario modificar manualmente el archivo "CmakeLists.txt" que se habrá creado automáticamente.

En concreto, la línea

```
#rosbuild_add_executable(example examples/example.cpp)
```

Debe quedar de la siguiente manera

```
rosbuild_add_executable(main src/main.cpp src/nodo.cpp)
```

Ahora el proyecto está preparado para ser importado a Eclipse directamente. Para ello, en Eclipse es necesario clickar en "Archivo->importar..." y seleccionar "Proyectos existentes al Espacio de Trabajo". Esto ofrece todas las comodidades del desarrollo en un IDE, como autocompletar texto, indentación automática y resaltado de sintaxis.

Para poder compilar y ejecutar el proyecto es necesario establecer las variables del entorno necesarias. Esto se consigue haciendo click con el botón derecho sobre el proyecto y eligiendo "Propiedades ->C/C++ Make Project" y comprobar que las variables ROS_ROOT, ROS_PACKAGE_PATH, PYTHONPATH y PATH están asignadas a los valores correctos (que pueden comprobarse escribiendo echo $[nombre de la variable] en consola).

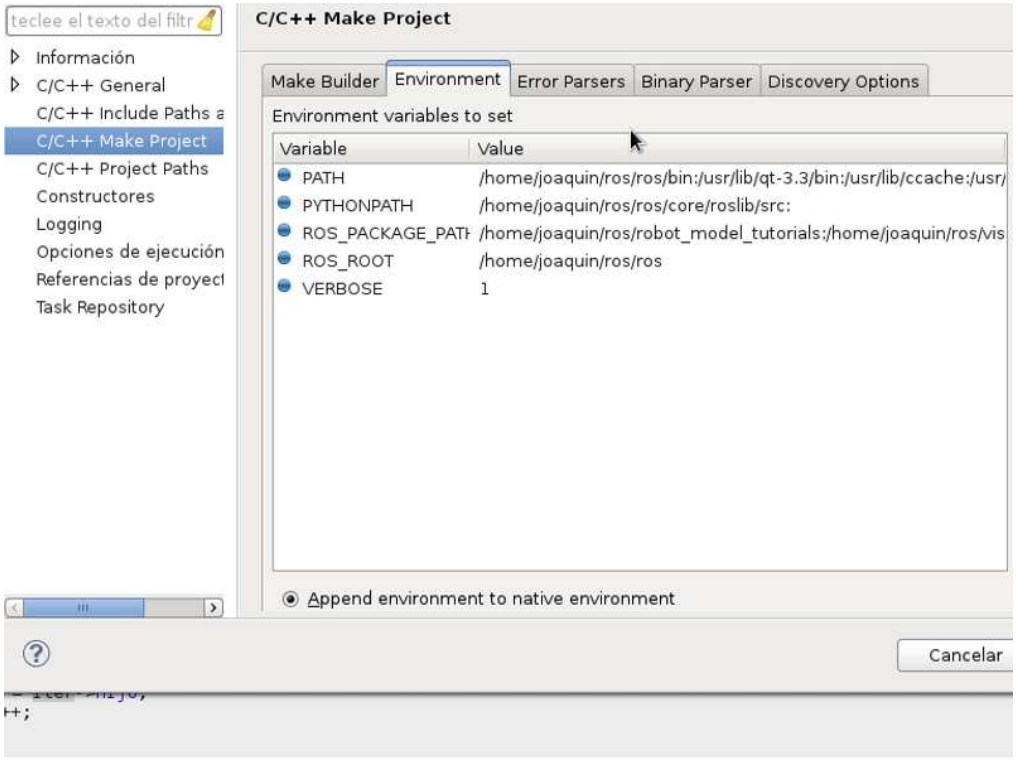

Figura 3.13: Configuración de las variables del entorno.

También es necesario informar a Eclipse de dónde establecer las conexiones para poder realizar la conexión con ROS cambiando los valores de las va-

riables ROS_ROOT y ROS_MASTER_URI en ”Ejecutar ->Configuración de la ejecución ->Proyecto C/C++.”

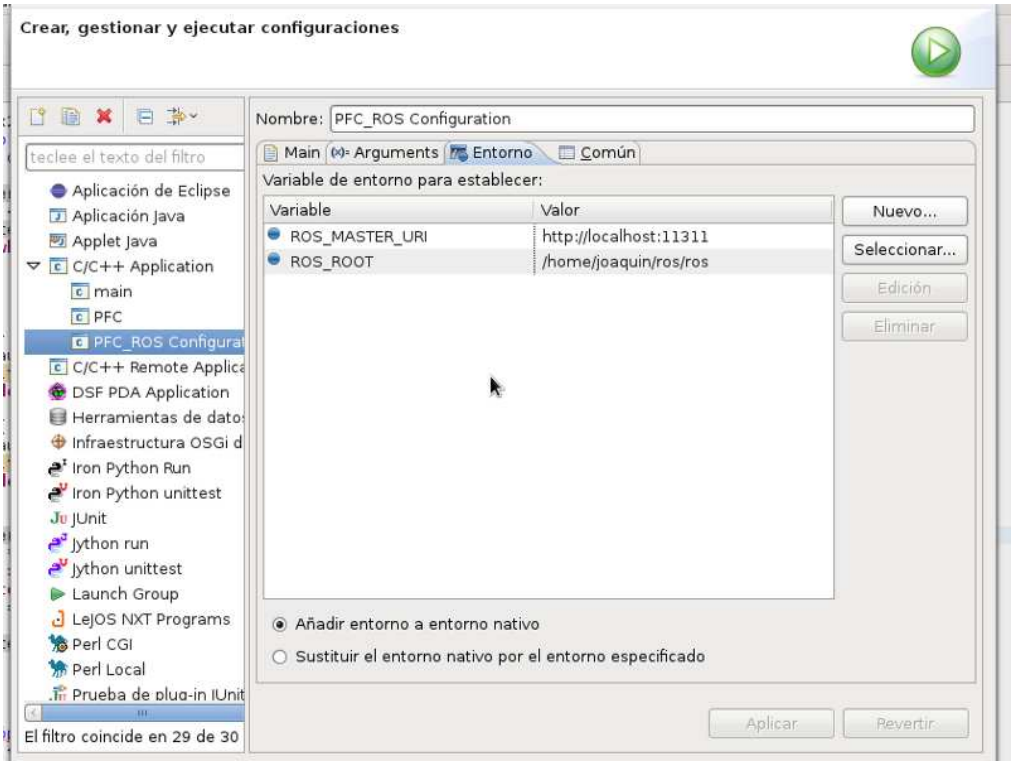

Figura 3.14: Configuración de las variables de ejecución.

Si un ejecutable existe previamente (si hemos hecho ”make” en la carpeta del proyecto antes de importar) es necesario seleccionar el ejecutable correcto en la pestaña ”principal” del diálogo anterior.

3.2. Desarrollo del proyecto

3.2.1. Introducción

Una vez preparado el entorno, es necesario establecer unos objetivos antes de comenzar el desarrollo.

En primer lugar, se necesita que el robot que incorpore el sensor Kinect sea capaz de recibir e interpretar los datos proporcionados por la cámara, así como utilizarlos para desplazarse por el entorno.

En segundo lugar, el robot ha de ser capaz de comunicarse con el algoritmo RGBDSLAM y utilizar la salida del mismo para localizarse en el entorno y ser capaz de cumplir una serie de objetivos, como alcanzar ciertas destinaciones, realizar un mapa completo, etc.

Por último, es necesario adaptar la funcionalidad del algoritmo de SLAM a la navegación de robots móviles, ya que no está diseñado para este fin. No es posible conocer variables como la certidumbre de una determinada estimación de la posición, ni las acciones de control necesarias para alcanzar un destino.

3.2.2. Modelado

Con estos objetivos en mente, se procede a escoger la forma de representar la información necesaria para la localización y la navegación.

La estrategia elegida para gestionar el mapeado se corresponde con la versión basada en celdillas de navegación vista en 2.3.2 por su equilibrio entre simplicidad de implementación y la potencia de los posibles algoritmos de búsqueda de caminos que pueden crearse para este diseño.

3.2.3. Estructuras de datos

Mapa

Ya que se va a tratar con un diseño basado en mapas de ocupación[16], el problema del diseño de la estructura de datos para almacenar la información del mapeado tiene una solución, en principio, trivial: podemos representar el mapa dentro de una matriz de las dimensiones que nos permita el hardware almacenar. Esta matriz estará compuesta de elementos binarios, que indicarán si la porción del mundo representada por cada casilla se encuentra libre (con un valor de true) u ocupada (false).

A esta información es necesario añadir una variable de incertidumbre, que nos diga qué tan fiable es la información que poseemos sobre este fragmento del entorno. En nuestro diseño, esta variable adquiere dos posibles valores: observado y no observado. El mapa, por tanto, se almacenará en memoria con la forma de un registro compuesto por dos valores binarios, que se corresponden con variables de tipo *Bool* en C++.

La estructura en forma en forma de matriz bidimensional permite una conversión directa desde el espacio cartesiano, asignando columnas al eje de las X y filas al de las Y. Esta representación es trivialmente ampliable al espacio tridimensional, pero el robot que se utilizará en este proyecto

se trata de un robot móvil con ruedas, limitando las trayectorias al plano correspondiente al suelo.

La resolución de la matriz, definida como la dimensión en metros cuadrados a los que corresponde cada celdilla de la matriz, está almacenada en una variable de tipo *float* que puede variar su tamaño según las posibilidades del hardware utilizado, pero que debe permanecer constante durante toda la ejecución del programa.

Trayectorias

Se define una trayectoria en este modelado como la lista de casillas que forman el camino desde una posición inicial hasta una posición final. Así pues, cada elemento de dicha lista debe contener un identificador único correspondiente a su ubicación en el espacio cartesiano. En concreto, cada elemento contiene una variable de tipo entero para su coordenada X y una para su coordenada Y.

Se ha decidido que, para las aplicaciones de este proyecto, puede definirse la contigüidad de cada casilla como el conjunto de los casillas 4-conectadas, requiriendo numerosas modificaciones del algoritmo principal cualquier cambio que se haga en este respecto. La razón por la que se ha escogido 4-conexión frente a 8-conexión (u otras opciones más potentes) se explica en 3.2.5.

Para la implementación de los algoritmos de búsqueda, y por la naturaleza dinámica del problema a resolver, se escoge una estructura de datos en forma de lista enlazada para almacenar el grafo de búsqueda. En dicho grafo, cada nodo corresponde con una casilla de la matriz y las aristas representan contigüidad. Al no tratarse de un mapa topológico 3.2.3 no es necesario almacenar el peso de las aristas, ya que es constante para todas ellas por la propiedad que representan. Cabe destacar que, aunque la contigüidad sea una cualidad inherentemente simétrica, la estructura en forma de lista y la forma en la que los punteros se comportan en C++ hacen de éste un grafo dirigido.

Para facilitar el diseño del algoritmo de búsqueda de caminos, así como otras funciones del programa principal, se han realizado algunas modificaciones a la implementación común de la lista enlazada.

La primera de ellas es la creación de una súper-clase que agrupa todas las funciones de acceso a la estructura de datos y que consiste en un simple puntero al primer nodo, que se corresponde con la casilla de inicio de la búsqueda. De esta forma, junto a la manera de introducir los nodos dentro de la estructura (como se explicará en 3.2.3) se puede crear una cierta jerarquía interna en la lista. Esto será útil en la gestión del árbol de búsqueda.

La siguiente modificación realizada es la conversión de la lista de simple a

doblemente enlazada: cada nodo contiene un puntero al nodo siguiente, pero también al nodo anterior. Ésto facilita el recorrido en ambas direcciones de la misma, creando así una estructura que representa mejor las propiedades de un grafo no dirigido.

Por último, se añade a cada nodo la posibilidad de ser, a su vez, miembro de una cola dinámica simple con la inclusión de un nuevo puntero.

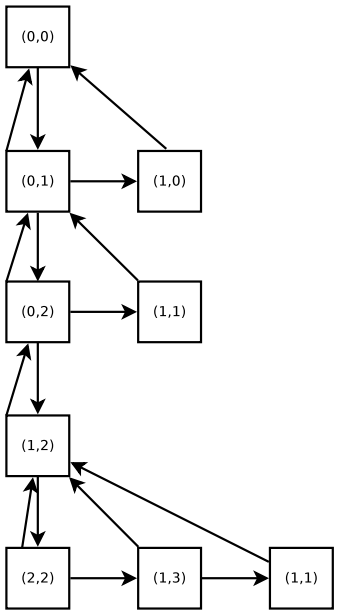

Figura 3.15: Representación de un posible árbol de búsqueda.

Podemos observar una implementación de esta estructura de datos en la figura 3.2.3, la columna de nodos de la izquierda representa la trayectoria recorrida hasta el momento. Cada fila representa el conjunto ordenado de posibles caminos a considerar de llegar hasta una ”hoja”. Las aristas representan la existencia de un puntero en la clase conectando los nodos.

Controlando qué nodos se incluyen en cada iteración del algoritmo de búsqueda, esta estructura cumple las propiedades necesarias de un árbol: es un grafo dirigido, sin ciclos y con un nodo sin aristas de entrada que puede identificarse fácilmente como raíz del árbol.

Para construir correctamente un árbol, consideraremos raíz al nodo correspondiente a la casilla inicial, en la que el robot se encuentra cuando lanza una orden de búsqueda. Para cada nodo que decidamos expandir, consideraremos ”hijo” al *primer* nodo que se incluya que sea contiguo a éste. Los demás posibles ”hijos” generados se considerarán ”hermanos” del primero y formarán parte de una lista simple.

Con esta jerarquía se consigue tener, en todo momento, acceso a una lista doblemente enlazada que contiene la trayectoria seguida a la que puede accederse recorriendo la lista de vectores "hijo" o "padre" según el sentido en que se desee conocer el camino realizado. Además, se tiene un acceso a una cola de prioridad de los nodos "hermanos" del nodo actual en caso de necesitar hacer una búsqueda hacia atrás (ver 2.4.2).

Esta estructura garantiza un funcionamiento óptimo en algoritmos de búsqueda basados en DFS, mientras que ofrece facilidades para el recorrido en anchura. La utilidad de este último punto será explicada más adelante.

Estrategia de búsqueda

El algoritmo de búsqueda de caminos requiere tres datos para poder funcionar: una posición de inicio, una posición objetivo e información sobre el mapa en el que buscar.

La posición de inicio, que debe estar necesariamente desocupada, se convierte en la raíz del árbol de búsqueda. Desde aquí, y para cada nueva posición que se desee explorar, se procede a expandir el nodo (siempre y cuando éste no sea el nodo destino).

El proceso de expansión consiste en identificar cuáles de las posibles posiciones 4-conectadas son candidatas viables a ser incluidas en el árbol. El criterio que sigue el algoritmo de búsqueda para comprobar la viabilidad de cada casilla es el siguiente:

- La casilla evaluada ha de estar contenida dentro de los límites de la matriz que contiene el mapa. No debe ser posible acceder a posiciones de memoria sin asignar.
- La casilla no debe estar ocupada. De esta forma garantizamos que sólo se incluyen aquellos nodos que tienen posibilidades de formar parte del camino final, reduciendo el tiempo y el espacio que se desperdicia en llamadas a reserva dinámica de memoria.
- La casilla no debe formar parte del recorrido seguido hasta el momento. Al no añadir nodos que ya forman parte del trayecto recorrido aseguramos la eliminación dinámica de bucles en la navegación, condición necesaria para la formación de un árbol de búsqueda.
- Como extensión del criterio anterior, el nodo "padre" del actual se descarta antes de realizar la evaluación por los otros tres criterios, ahorrando una comprobación por paso. Dado el coste de la búsqueda en el camino actual (búsqueda recursiva de coste lineal con el número de nodos en el camino), es una optimización notable para caminos largos.

Si al finalizar el proceso de selección de nodos no queda ninguna posible casilla destino, hemos alcanzado una ”hoja”, un nodo sin ”hijos”. Llegados a este punto, el algoritmo de búsqueda seleccionará una nueva ”rama” del árbol, de entre las insertadas anteriormente. Para ello, se recorrerá primero la lista de ”hermanos” del nodo actual. De estar vacía, se procederá a repetir el proceso recursivamente con el ”padre” del nodo actual. Si durante este proceso de búsqueda por backtracking alcanzamos el nodo raíz, habremos recorrido todos los caminos posibles y el destino no será alcanzable.

En cualquier otro caso (se ha seleccionado una casilla como candidata a formar parte de la selección), se crea un nodo que pasa a una cola ”de espera” donde se realiza el cálculo de la estimación del coste (en este algoritmo, distancia al nodo final) de elegir ese nodo.

Una vez rellenada la cola de espera, los nodos candidatos son ordenados mediante un algoritmo de ordenación simple, como el de la burbuja (`http://en.wikipedia.org/wiki/Bubble_sort`), dando una mayor prioridad a los nodos con una menor distancia estimada total.

Este proceso se repite hasta que se encuentra el nodo destino o un camino imposible. De encontrar un camino viable, se procede a realizar una optimización del mismo.

La reducción del coste total del camino consiste en realizar un recorrido en anchura, de manera iterativa desde la raíz, del árbol. Por cada uno de los nodos ”hermano” de los incluidos en el camino, se realiza una comprobación para averiguar si ya forman parte del camino en un nivel inferior, es decir, si el camino podría haber alcanzado ese nodo por un camino más corto. De detectarse una posible optimización, el camino completo se ”cuelga” de la ”rama” más corta, eliminando todos los nodos sobrantes (ver figura 3.2.3).

Ejemplo de optimización

Si se aplica el algoritmo descrito en un en un entorno como el mostrado en las siguientes imágenes, donde la casilla de inicio está marcada con un color azul y la de destino, con un color rojo y los obstáculos están señalados con casillas negras, el camino que se recorre es el siguiente:

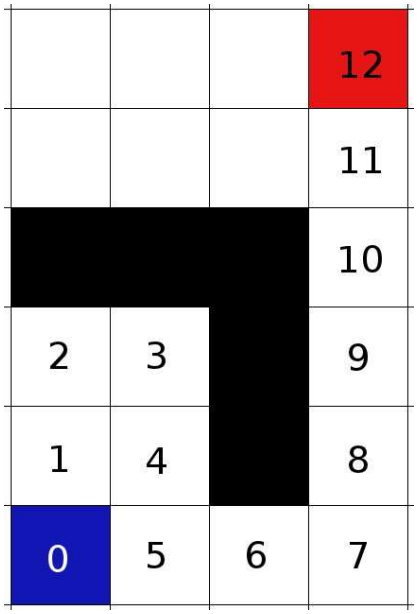

Figura 3.16: Camino estimado por el algoritmo.

Sin embargo, se puede apreciar que existen casillas que, a posteriori, se sabe que no era necesario recorrer (señaladas en verde).

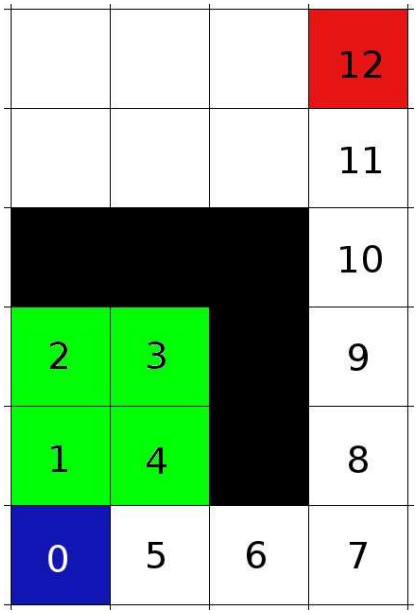

Figura 3.17: Casillas sobrantes.

El algoritmo de optimización descubrirá que el nodo marcado como 5 podía haberse explorado desde el nodo inicial, pero la heurística indicaba que el nodo superior era una mejor elección (de no existir el obstáculo).

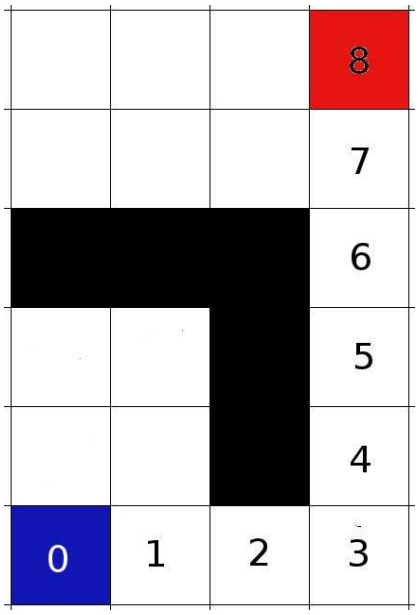

Figura 3.18: Casillas sobrantes.

El camino resultante, por tanto, no contiene el recorrido innecesario, acortando significativamente el largo del trayecto total.

3.2.4. Sistema de visión

Como se ha comentado en la sección 3.1.3, existen ciertas limitaciones en cuanto al ancho de campo de la cámara y las profundidades mínimas y máximas del sensor de distancias que es necesario evaluar. Los valores obtenidos, facilitados por el fabricante, hacen referencia al uso de la cámara como controlador de videojuegos, así que es necesario comprobar los límites del sensor en el entorno de la robótica.

En la figura están representados los datos obtenidos mediante experimentación con el sensor y el driver OpenNI. La zona roja, de unos 40 centímetros de profundidad, representa la zona en la que es posible obtener imagen RGB pero no imagen de profundidad. La zona azul es donde se pueden obtener datos fiables de ambos sensores.

Se puede observar que el ancho de campo limita también la cantidad de casillas que pueden observarse en función de la profundidad considerada. En

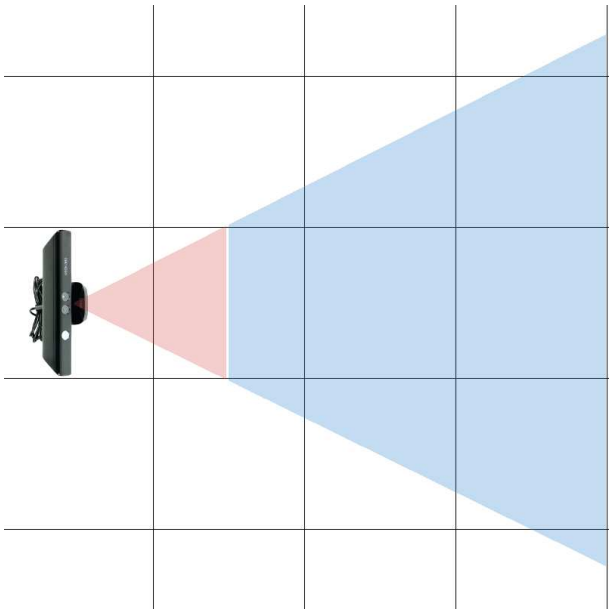

Figura 3.19: Datos obtenidos de manera empírica. Las casillas tienen una dimensión de 40x40 centímetros.

la siguiente figura, las casillas blancas son aquellas de las que no se dispone ningún tipo de información, mientras que de las verdes (y también de la roja, ver 3.2.6) es posible obtener datos.

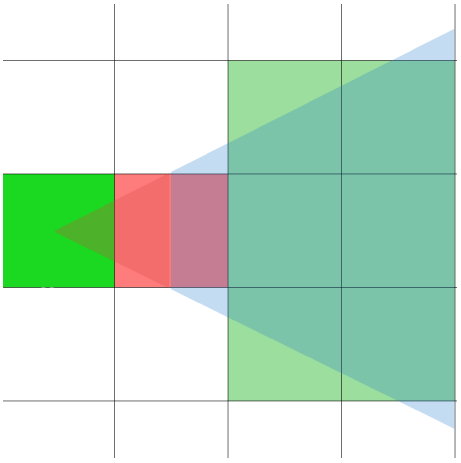

Figura 3.20: Información disponible.

3.2.5. Modelado del sistema de visión

Una vez investigadas las posibilidades de Kinect como sensor de distancias, se puede proceder al diseño de un modelo simplificado del cono de la cámara que funcione en el contexto de un mapeado basado en celdillas de ocupación.

Debido al límite inferior de la distancia, es posible establecer una resolución de matriz de 0.4x0.4 metros. Usando esta información y la relativa al ancho de campo de la cámara, se puede establecer un modelado del cono del sensor similar al representado en la figura 3.2.5.

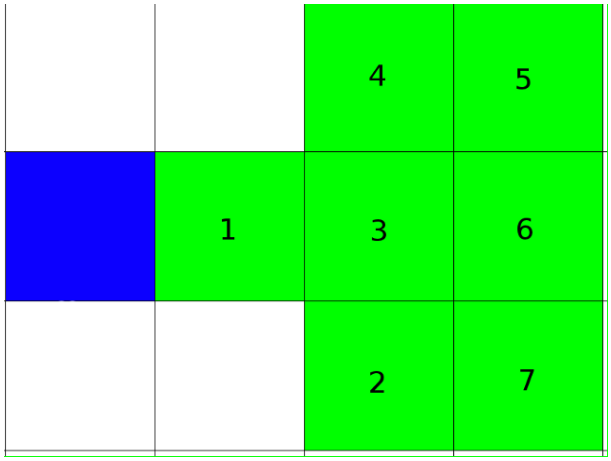

Figura 3.21: Distribución de casillas en el cono de visión.

Debido a las dimensiones del entorno en el que se desarrolló este proyecto, se ha limitado la profundidad del modelo en tres niveles, aunque es posible extenderlo al menos en dos niveles dadas las características de la cámara.

El algoritmo de búsqueda debe ser modificado para adaptarse a la nueva manera de obtener la información de la ocupación de las celdillas.

En principio, se considera el mapa completo como libre de obstáculos. En cada iteración, se comprueban todas las casillas posibles alrededor de la actual, actualizando todos los cambios detectados en el mapa. El algoritmo de búsqueda siempre tendrá, de esta manera, información fiable de todas las casillas 4-conectadas y podrá calcular el siguiente destino de la misma forma que lo hacía con anterioridad sí, y sólo sí, se obtiene información alrededor de la posición inicial antes de realizar el primer cálculo.

Para reducir el número de giros que habría que realizar y, por tanto, el tiempo en que el algoritmo está esperando la información relativa al entorno los giros pasan por un proceso de selección similar al de los nodos (ver 3.2.3).

El algoritmo de minimización no incluirá un giro en la cola de movimientos a realizar si:

- El giro llevaría a la cámara a intentar obtener información de posiciones inalcanzables (fuera de los límites del mapa), siempre y cuando no se observe al menos una posición relevante en el proceso.
- El giro no aportaría nueva información. Es decir, el giro nos llevaría a mirar hacia el camino recorrido.
- Las siete casillas susceptibles de ser observadas ya han sido examinadas.
- Una casilla que se encuentra en un nivel superior ("detrás") a una casilla con obstáculo se considera imposible de examinar.

Una vez finalizado este proceso de selección se obtiene la información del entorno mediante el algoritmo de visión (ver 3.2.6), codificada en un vector de siete posiciones que se corresponden con la numeración vista en la figura 3.2.5. Con éste, se rellenan las celdillas correspondientes en el mapa y puede comenzar una nueva iteración del bucle de control: planificación de ruta, desplazamiento y obtención de información.

3.2.6. Implementación del sistema de visión

Haciendo uso de las herramientas ofrecidas por la biblioteca PCL y la información obtenida de la cámara (2.7.2 y 3.1.3, respectivamente) es posible diseñar un filtro que cumpla dos objetivos: realizar correctamente la segmentación de la nube de puntos generada por el sensor y reducir la cantidad de datos que son procesados, optimizando los parámetros espaciales y temporales del programa.

Partiendo de una nube de puntos que abarca todo el espacio cubierto por la cámara, se eliminan todos aquellos puntos que quedan fuera del área relevante para el robot, es decir, todos aquellos puntos que no representan información relativa a posibles obstáculos a considerar.

Para esto, hacemos uso de un filtro pasa-banda proporcionado por la biblioteca PCL. El filtro funciona guardando en una nube de puntos auxiliar sólo aquellos puntos cuyas características se encuentren dentro de unos márgenes determinados. Estas características pueden ser tanto las coordenadas X, Y o Z como otros parámetros, como el color RGB del punto, por ejemplo.

El primer filtrado de la nube de puntos elimina todos los puntos que se encuentran por encima del nivel de la cámara, punto más elevado del robot empleado. Se mantiene un pequeño margen (10 cms) para evitar posibles

errores de lectura causados por el ruido del sensor. Asimismo, se eliminan los puntos correspondientes al suelo, que forman parte de la nube de puntos, pero no deben considerarse obstáculos. De esta forma se elimina un conjunto de puntos significativamente grande, ya que el suelo se almacena de forma densa en la nube, y se elimina toda la información respectiva a la parte superior de la imagen.

Para obtener la información relativa a la profundidad inferior a la mínima detectada por el sensor – 40 centímetros – es necesario conocer una propiedad de la cámara: si el objeto es lo suficientemente ancho con respecto al campo de visión, la "sombra" proyectada por éste bloqueará casi por completo el sensor.

Aprovechando esta información, el siguiente filtrado elimina el conjunto de puntos que se encuentra fuera de la celdilla ubicada frente a la cámara, ya que no pueden formar parte de un obstáculo localizado delante del robot. Si el tamaño de la nube de puntos después del filtrado es mayor que cero, podemos asegurar que no hay ningún objeto obstruyendo el sensor y, por tanto, no hay obstáculos en la casilla observada.

Partiendo de la nube filtrada original, los filtros para las siguientes profundidades funcionan de manera idéntica entre sí. Se eliminan todos los puntos que no corresponden a la profundidad deseada (de 40 a 80 centímetros en un caso y de 80 a 120 en el otro), y la nube de puntos resultante se segmenta en tres partes: nube de puntos izquierda, derecha y central. Si el tamaño de la nube de puntos de alguna de estas partes es mayor que cero, se marca como ocupada la posición correspondiente en el vector que se devuelve al algoritmo encargado de volcar esta información en el mapa.

Control de acceso

Para obtener acceso completo a la nube de puntos, es necesario subscribirse al *topic* en el que ésta publica su información. Para esto, necesitamos crear una función que ROS conoce como *callback*. A efectos prácticos, estas funciones se comportan como hilos separados de ejecución, que entran en acción cada vez que el bucle principal del programa termina una iteración y realiza una llamada a la orden *ros::Spin()*.

Como no es necesario, ni deseable, que el algoritmo de filtrado se ejecute constantemente, es necesario controlar la ejecución con semáforos. Los semáforos son variables que se marcan a determinados valores para permitir el avance de un programa. Un sistema con dos (o más) semáforos se utiliza cuando alguno de los hilos de ejecución necesita de la información proporcionada por otros, pero debe controlar cuándo y cómo se obtiene ésta.

En este proyecto, el acceso al algoritmo de filtrado debe estar cerrado

hasta que la detección de obstáculos necesite obtener datos nuevamente sobre su entorno y, a su vez, no puede continuar con su ejecución hasta que los datos no hayan acabado de obtenerse correctamente, por lo que son necesarios dos semáforos.

Así, cuando se alcanza la orientación correcta de la cámara y es necesaria una captura de datos nueva, se activa la variable que impide al filtro ejecutarse en cada iteración del algoritmo. En una implementación normal, el filtro obtendría el acceso de forma inmediata y podría continuar su ejecución, pero en ROS es necesario forzar una iteración del nodo principal, lo que se logra mediante la llamada a función *ros::SpinOnce()*.

Una vez el filtro ha terminado de rellenar los datos necesarios, desactiva su variable de acceso y activa la variable que deja saber al algoritmo principal que ha terminado su ejecución. Éste puede entonces continuar.

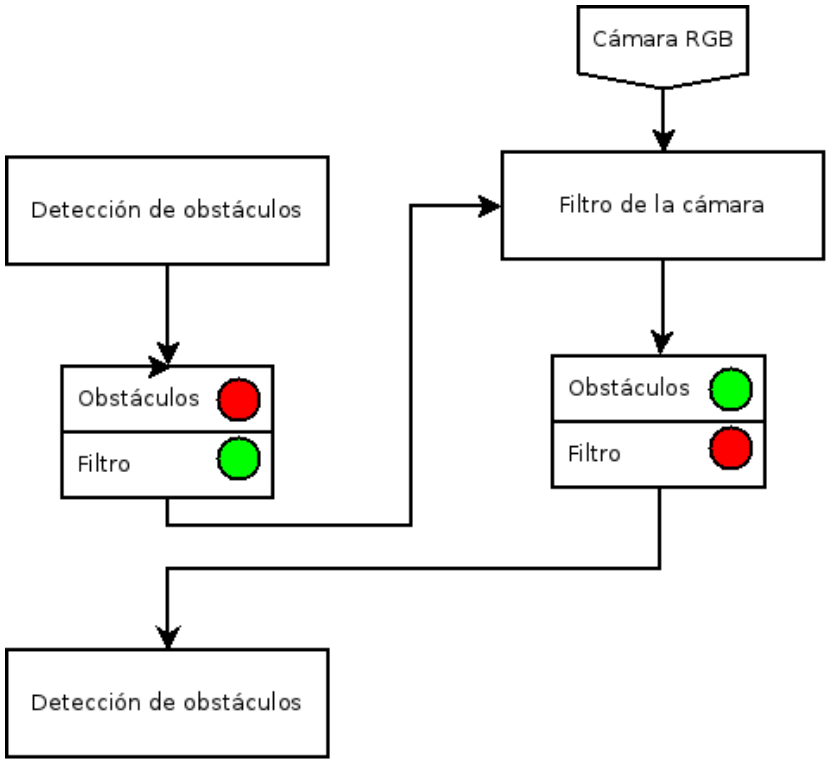

Figura 3.22: Esquema del funcionamiento del control de accesos.

3.2.7. Integración con RGBDSLAM

Llegado a este punto, lo único que resta al algoritmo es desplazar físicamente al robot dentro del entorno.

Con este fin se utilizan dos herramientas proporcionadas por la biblioteca tf (3.1.3): una transformada "estampada", es decir, que posee información sobre el tiempo en que ha sido capturada, y un escuchador para realizar la suscripción a la salida del programa RGBDSLAM.

La salida del programa no comienza hasta que la cámara ha captado su primer movimiento, algo que suele suceder incluso con la cámara estática debido al ruido del sensor, así que la primera transformada de la posición recibida tendrá un pequeño desplazamiento con respecto a la posición asumida por el algoritmo (0, 0, 0). Por tanto, el programa principal debe esperar a recibir la primera información proporcionada por RGBDSLAM y almacenar ese valor como *offset*, que se aplicará a todos los valores obtenidos a partir de ese momento.

Cuando se tiene un flujo constante de datos y un desplazamiento fiable, el bucle principal procede con los cálculos correspondientes a rutas, obstáculos y mapeado deteniéndose únicamente cuando necesite de los datos de la posición y orientación actuales.

En ambos casos, el procedimiento es similar:

- Se espera a que el escuchador tenga primera la transformada.
- Se calcula la resta de la transformada actual y el desplazamiento inicial.
- Se envía la orden de movimiento (rotación, traslación) correspondiente al robot (ver 3.3).
- Se compara la posición actualizada calculada a partir de la transformada con la posición destino y si no coinciden el bucle detiene su ejecución durante 50 milisegundos, para sincronizar con el bucle de control del robot.
- Una vez alcanzada la posición deseada se devuelve el control al bucle principal.

Las posiciones destino corresponden con el centro de las "casillas" en las que se ha dividido el entorno. Las rotaciones de la cámara se calculan en incrementos de noventa grados para observar las casillas 4-conectadas y las posiciones observables tras éstas. El bucle principal continúa hasta que se ha alcanzado la casilla de destino o se detecta una casilla de destino inalcanzable.

3.3. Plataforma final y adaptaciones

La información sobre la plataforma física en la que finalmente se ha implementado el proyecto, así como toda la información relacionada con los algoritmos de comunicación, control y movimiento del robot se encuentra detallada en [17].

El robot empleado fue construido usando la tecnología LEGO NXT MINDSTORM. Esta plataforma incluye: el brick NXT, múltiples sensores (luz/color, ultrasonidos, contacto, sonido) motores y cableado eléctrico, y el set de construcción habitual de la línea Technic, formado por miles de piezas de construcción, engranajes, ejes correas y poleas.

El robot LEGO Mindstorms NXT es la segunda versión del sistema desarrollado en colaboración entre LEGO y el MIT, presentado en enero de 2006 en el International Consumer Electronics Show. La nueva versión además de otros cambios menores en los sensores electrónicos y las piezas de construcción, incorpora una unidad de control nueva: el NXT.

Combinando los bloques de construcción, la fácil programación del NXT y su interfaz de entrada y salida se puede obtener un sistema de prototipado rápido para el desarrollo de una gran variedad de actividades, lo que ha permitido que este sistema haya sido ampliamente aceptado como una herramienta para la investigación y la educación universitaria como se demuestra con la publicación de ediciones especiales de revistas como IEEE Robotics and Automation Magazine o IEEE Control Systems Magazine.

La unidad de control, definida como ladrillo inteligente NXT, está basada en el ARM7, un potente microcontrolador de 32 bits, con 256 Kbytes FLASH y 64 Kbytes de memoria RAM. Para la programación y las comunicaciones, el NXT está equipado con un puerto USB 2.0 y con un dispositivo inalámbrico Bluetooth clase II, V2.0. Así mismo, el NXT dispone de 4 entradas (una de ellas incluye una expansión IEC 61158 Type 4/EN 50 170 para usos futuros) y 3 salidas analógicas.

La nueva versión del LEGO proporciona 4 tipos de sensores electrónicos: de contacto, de luz, de sonido y de distancia. Además, existen varias compañías que han desarrollado numerosos sensores compatibles con el NXT, como giróscopos, acelerómetros, buscadores de infrarrojos, cámaras de visión artificial, etc.

De entre los distintos disponibles, en este trabajo se ha optado por utilizar el firmware LeJOS, el cual permite la programación de los NXT en lenguaje Java, ofreciendo librerías orientadas a la navegación, a la comunicación Bluetooth, al uso de trigonometría y funciones matemáticas complejas, a la programación multithreading, al soporte de programación orientada a objetos... Además proporciona distintos niveles de abstracción a la hora de

programar los movimientos de los NXT, desde funciones de alto nivel que definen rectas y curvas, al tratamiento del movimiento de los servos de corriente continua a nivel del voltaje que se les aplica.

A continuación una lista de las características electrónicas del brick NXT, extraídas del manual de LEGO:

Figura 3.23: Especificaciones técnicas.

Aunque estas características son lo suficientemente potentes para ejecutar gran parte del algoritmo principal, no cumplen con los requisitos para el tratamiento de imágenes online, ni para la ejecución del programa RGBDSLAM. Por tanto, la parte software se ejecutará en un PC, y toda la comunicación se realizará por medio de USB. Será el propio robot quien decida la mejor forma de generar la trayectoria, mientras que el PC decidirá los distintos destinos a los que éste deberá desplazarse.

La información de la cámara es tratada directamente por el PC, mientras que el control de la posición de la cámara se realiza en el robot.

A este fin, es necesario realizar ciertas modificaciones sobre el código original. La primera de ellas es definir qué tipo de órdenes van a intercambiarse entre el programa y el robot:

```
//Definiciones de las órdenes posibles para el robot
#define DESPLAZAMIENTO 0
#define ROTACION 1
#define FRENAR 2
#define FIN 3
#define POSICION 4
#define FRENAR_ROTACION 5
#define DESPLAZAMIENTO_GIRO 6
```

Gracias a la configuración del robot, donde la cámara está sujeta a una plataforma independiente que puede rotar mientras el robot permanece estático o desplazándose en una línea recta, es posible separar las órdenes de rotación y de traslación. Por este motivo existen, a su vez, dos órdenes de frenado independientes.

Para suavizar las trayectorias en las que el robot, además de avanzar, debe girar, se crea una orden que avisa al robot para que éste decida cuál es la curva más adecuada para alcanzar su destino.

La orden llamada "FIN" es la que avisa al robot que el programa ha terminado, para que comience el protocolo de desconexión y finalización de ejecución internamente.

Los segmentos del programa que tratan con el desplazamiento y la rotación ahora deben esperar confirmación del robot antes de continuar su ejecución, y deben actualizar, en cada iteración, la información del robot sobre su posición y orientación actuales, obtenidas mediante RGBDSLAM.

La rutina que sufre un cambio más drástico con respecto a su implementación original es la de backtracking, ya que debe informar al robot de todos los pasos que debe dar para deshacer el camino recorrido, porque éste no almacena la trayectoria seguida.

Para realizar correctamente las comunicaciones, se han desarrollado dos funciones: *enviar* que gestiona las comunicaciones PC-robot y *recibir* que se encarga de las comunicaciones robot-PC.

La comunicación entre ambos dispositivos se realiza usando una implementación de la biblioteca libusb y ha de ser inicializada al comienzo del bucle principal:

```
dev = nxt_find_nth(cnt, name); hdl = nxt_open(dev);
```

Las variables *cnt* y *name* contienen información específica del *brick* que se utiliza para ejecutar el código y pueden obtenerse sus valores mediante

la órden *lsusb* en la línea de comandos. Las llamadas a ambas funciones son bloqueantes, es decir, impiden el avance del programa hasta que han acabado su ejecución y ambas deben terminar la ejecución del programa si no consiguen conectar exitósamente con el *brick*.

Una vez inicializada la comunicación, *enviar* codifica la información que se ha de enviar al robot utilizando los *defines* mostrados anteriormente. Además, el robot recibe los valores de X e Y correspondientes a su casilla actual en la matriz de ocupación, y la orientación deseada al final del movimiento.

Recibir es una función bloqueante que detiene la ejecución del bucle principal hasta que recibe una señal del robot, confirmando que ha finalizado correctamente de ejecutar la orden recibida.

3.4. Análisis de una ejecución del programa.

Figura 3.24: Laberinto en el que se realizan las pruebas del programa.

La figura 3.4 presenta un ”laberinto” de ejemplo en el que el robot habrá de alcanzar un cierto destino desde una casilla inicial, de manera completamente autónoma. El modelado del problema puede observarse en la siguiente figura:

En este modelado, la casilla inicial del robot (parámetro del programa) está señalada en color azul y la casilla final en rojo.

La primera tarea que debe realizar el programa es elegir la cantidad de giros necesarios para conocer toda la información relativa a la posición inicial del robot. Dada la posición inicial del robot y el tamaño de la matriz, se

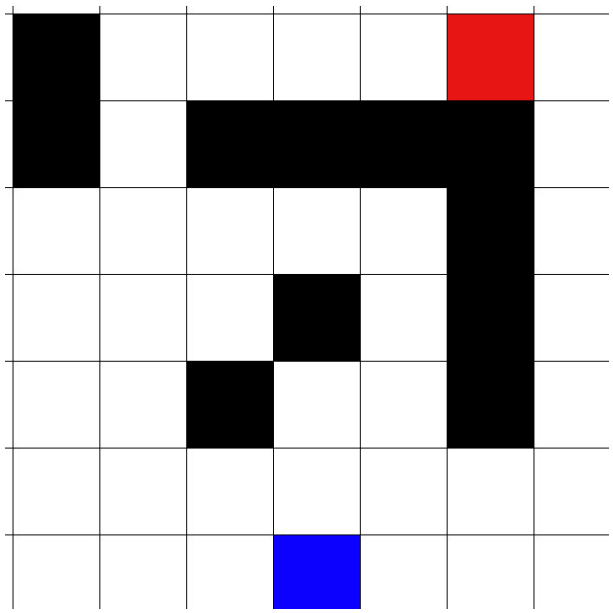

Figura 3.25: Modelado correcto del laberinto.

realizarán 3 giros: delante (o ”arriba”), izquierda y derecha. Para realizar esto, el robot recibe tres órdenes separadas de ”ROTACIÓN” y el programa espera en la orden ”recibir” hasta que consigue una confirmación de que el giro ha sido realizado.

Cuando se ha obtenido la información relativa a las 3 casillas adyacentes del robot (todas las casillas de las que se disponga información serán señaladas con el color gris en las siguientes figuras) el algoritmo puede comenzar con el proceso de selección de destinos parciales y la obtención de información del entorno.

En este caso, la heurística dice que la casilla que ofrece mejores posibilidades para alcanzar el destino de forma óptima es la casilla de ”arriba” de la actual, ya que el objetivo se encuentra a 6 casillas de distancia hacia arriba y sólo 2 hacia la derecha. Se envía al robot una orden de desplazamiento de 40 centímetros en la misma dirección en la que se encuentra el robot actualmente.

A partir de la segunda iteración, se aplica una optimización extra: si no se tiene información de una casilla adyacente a la actual, pero dicha casilla no forma parte del (posible) camino ésta *no* se explorará, para reducir el número de giros necesarios.

Así pues, una vez elegido el primer destino y realizado el desplazamiento, el proceso se repite. El algoritmo selecciona los giros necesarios para conocer

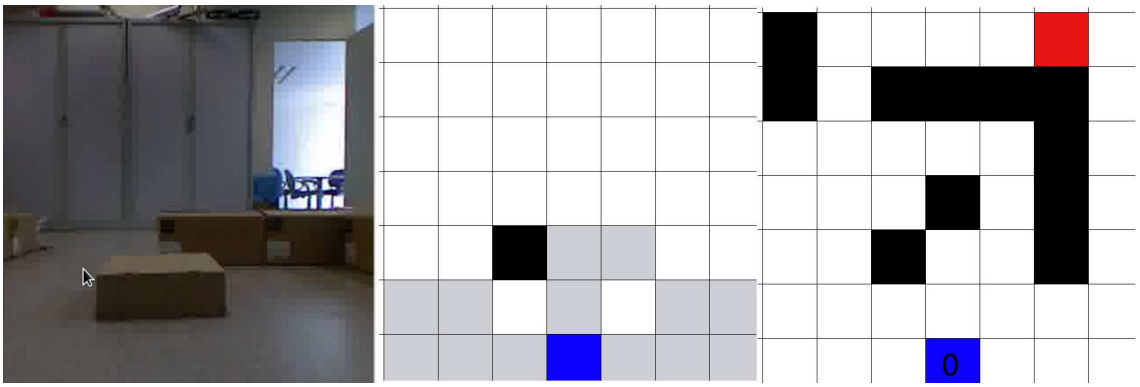

Figura 3.26: Iteración inicial. De izquierda a derecha: captura de la cámara, información del robot y desplazamiento numerado en el mapa.

toda la información necesaria para estimar el mejor movimiento posible, que vuelve a ser un avance de 40 centímetros. Puede observarse que, aunque no se dispone de información de ninguna de las casillas laterales no se analizará su contenido, debido a la optimización adicional, hasta que no se elija alguna de ellas como posible destino. En caso de que fueran elegidas y al realizar la comprobación se descubriera que están ocupadas, el algoritmo escogerá otra de las casillas adyacentes a la actual.

El obstáculo localizado directamente enfrente de la cámara no fue detectado en la primera iteración debido a la imposibilidad de distinguir la "sombra" proyectada por el obstáculo de la izquierda de un área libre de obstáculos. El algoritmo de detección de obstáculos detiene su ejecución en dicho nivel, al no poderse obtener más información fiable de los posteriores.

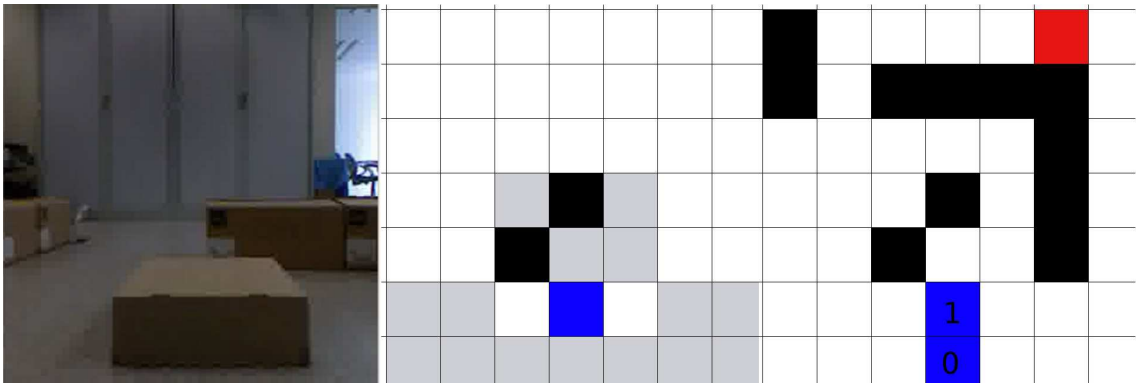

Figura 3.27: Segunda iteración.

Al encontrarse el robot en un ”rincón”, es necesario realizar el primer desplazamiento que requiere un giro. La orden ”DESPLAZAMIENTO_GIRO” avisa al robot que tendrá que cambiar su orientación para que éste elija la trayectoria que considere más apropiada, realizando una curva suave en vez de una recta.

Para maximizar la información recibida, el robot realiza una exploración adicional del entorno después de realizar un giro ya se pueden observar casillas que se encontraban fuera de su alcance. Esta comprobación tiene lugar después de un ajuste de la cámara, ya que la orientación final del robot no tiene que ser necesariamente la correcta, por errores en su propia odometría o por la trayectoria realizada. Se utiliza la información de RGBDSLAM para realizar la reorientación del sensor.

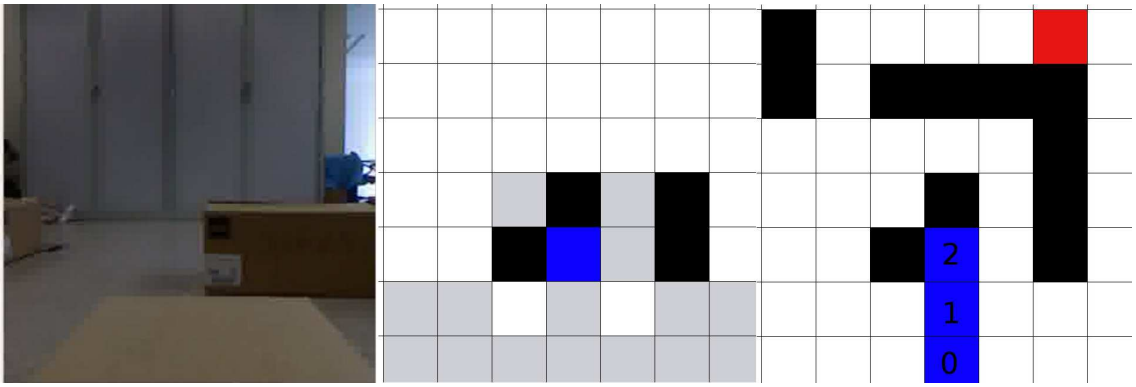

Figura 3.28: Tercera iteración.

Debido al diseño del ”laberinto”, las siguientes iteraciones ocurren sin particularidades hasta alcanzar la novena iteración. En ella, el robot ha encontrado un camino que lo ha acercado al borde de la matriz de ocupación. Dicho borde funcionará de manera muy similar a una frontera o pared virtual. No se observará nada que se encuentre fuera de ese límite ni se considerarán giros que lleven al robot más allá del mismo. De esta manera se evitan giros innecesarios, que consumen una gran cantidad de tiempo.

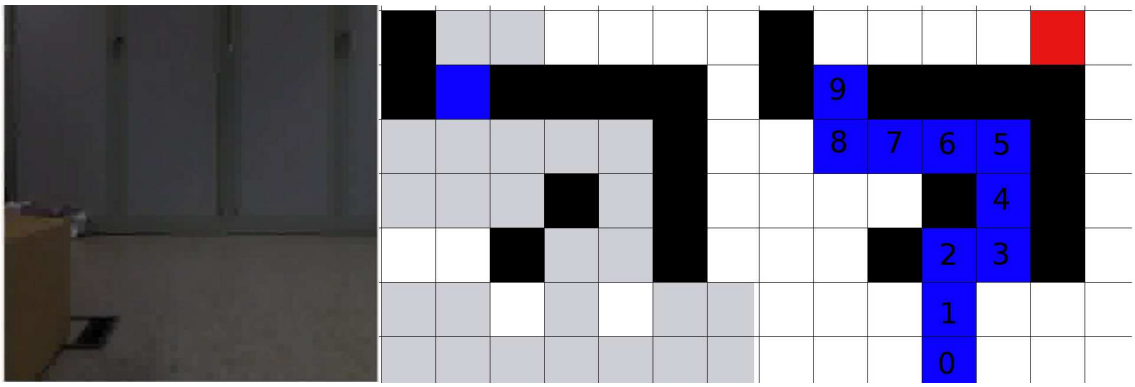

Figura 3.29: Novena iteración.

Finalmente, el robot alcanza la posición final. El programa informa al mismo de esta situación mediante el código de orden "FIN". Así, el robot puede comenzar las rutinas de desconexión y liberación del puerto USB que está utilizando.

El programa notifica de la resolución exitosa al usuario mediante un mensaje y procede a optimizar la trayectoria realizada (si es posible), notificando al usuario de ello.

> ''Solución encontrada en 14 desplazamientos!"
> El camino calculado es: (3, 0), (3,1), (3,2), (4,2), (4,3), (4,4), (3,4), (2,4), (1,4), (1,5), (1,6), (2,6), (3,6), (4,6), (5,6)
> El camino optimizado es: (3, 0), (3,1), (3,2), (4,2), (4,3), (4,4), (3,4), (2,4), (1,4), (1,5), (1,6), (2,6), (3,6), (4,6), (5,6)
> No se han producido modificaciones.

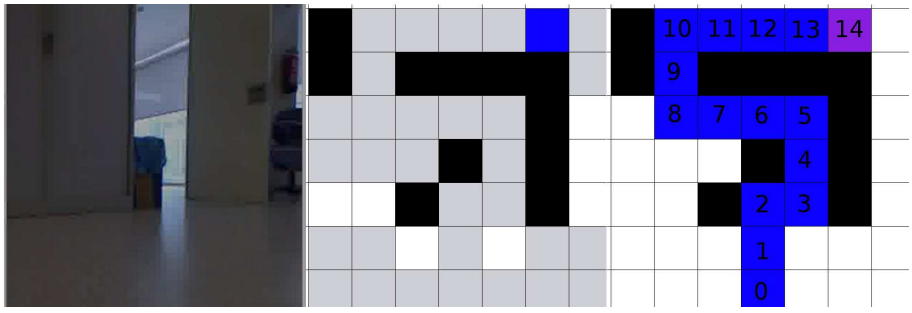

Figura 3.30: Laberinto completado.

Capítulo 4

Conclusiones y futuros proyectos

Se han cumplido exitosamente los objetivos que se proponían al comienzo de este proyecto. Se ha adquirido un conocimiento amplio sobre el funcionamiento del sensor Kinect, de las tecnologías que usa y de sus limitaciones.

También se han estudiado las diferentes opciones de software disponible para esta plataforma, así como el conjunto de herramientas disponibles para tratar la información proporcionada por la misma. Se han realizado diferentes pruebas para conocer las estructuras de datos en las que se ha codificado dicha información.

Asimismo, se han estudiado diferentes enfoques en la resolución del problema del SLAM, con sus distintas ventajas e inconvenientes. Se ha estudiado un caso práctico de implementación de uno de estos algoritmos, y se ha comprendido tanto su funcionamiento como sus limitaciones a la hora de adaptarlo a otras aplicaciones.

Finalmente, los conocimientos adquiridos durante esta tarea de investigación de los distintos campos propuestos y las herramientas disponibles se han utilizado para crear una aplicación que combina todo lo aprendido. Se han utilizado los datos extraídos de la cámara para realizar un filtro detector de obstáculos, se ha utilizado la salida del algoritmo de SLAM para la navegación y se ha desarrollado un algoritmo que permite combinar ambas partes para generar una trayectoria que permite cumplir determinados objetivos.

Para futuros proyectos se propone una serie de mejoras sobre el código actual. Por ejemplo, sería deseable conseguir ampliar el conjunto de posibilidades a la hora de elegir un camino de las actuales casillas 4-conectadas a 8-conectadas o incluso conseguir un algoritmo con 360 grados de libertad.

También sería deseable aumentar la resolución de la matriz de ocupación, consiguiendo de esta manera trayectorias mejores y con más adaptabilidad

al entorno. Para ello sería necesario encontrar una manera más eficiente de segmentar la nube de puntos del sensor de distancias para hallar la correspondencia con las celdillas.

Aunque el algoritmo actual genera las trayectorias mientras el robot explora el terreno y, una vez alcanzado el destino, ésta es optimizada, sería interesante adaptar el programa para que la trayectoria fuera generada y optimizada con el conocimiento parcial que tiene el robot del mapa y, una vez se encuentre una inconsistencia entre el camino planeado y los objetos encontrados en el mapa, se cree una una nueva trayectoria óptima basada en los nuevos datos adquiridos sobre el entorno. Esto permitiría dotar al algoritmo de nueva funcionalidad con modificaciones mínimas sobre el código actual.

Sería deseable, también, la implementación de un algoritmo de SLAM que requiera un coste computacional menor que RGBDSLAM, permitiendo incorporar toda la funcionalidad expuesta en este proyecto a un robot completamente autónomo, sin la necesidad de un potente PC para realizar los cálculos correspondientes al tratamiento de la información.

Bibliografía

[1] Soren Riisgaard and Morten Rufus. Slam for dummies.a tutorial approach to simultaneous localization and mapping.

[2] Ángel Soriano Vigueras. Diseño y programación de algoritmos para robot móviles. aplicación al robot lego-nxt. Master's thesis, Universitat Polit'ecnica de Val'encia, 2009.

[3] Juan Reyero. A graphic explanation of the bayes theorem, November 2011.

[4] Jörg Stückler and Sven Behnke. robust real-time registration of rgbd images using multi-resolution surfel representations".

[5] Hugh Durran-Whyte and Tim Bailey. Simultaneous localisation and mapping (slam): Part i the essential algorithms.

[6] Dark Watcher. History of the game controller.

[7] Microsoft. *Manual del sensor Kinect*.

[8] OpenKinect.org. Open kinect documentation.

[9] Daniel Wigdor and Dennis Wixon. *A brave NUI world*. Morgan Kaufman, Reading, Massachusetts, 2011.

[10] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[11] Giorgio Grisetti and Rainer Kürmele. Hierarchical optimization on manifolds for online 2d and 3d mapping.

[12] Hao Yu and Bogdan Wilamowski. Levenberg–marquardt training.

[13] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf).

[14] Aleksandr V. Segal, Dirk Haehnel, and Sebastian Thrun. Generalized icp.

[15] Rainer Kürmele, Giorgio Grisetti, and Hauke Strasdata. "g2o: A general framework for graph optimization".

[16] Sebastian Thrun and Arno Bücken. Integrating grid-based and topological maps for mobile robot navigation.

[17] Vicent Mayans Roca. Desarrollo multiplataforma de aplicaciones de control y comunicación para robots móviles. Master's thesis, Universitat Polit'ecnica de Val'encia, 2012.

Índice de figuras

2.1. Robot rodante con cuatro ruedas. 7
2.2. Mini robot espía con seis ruedas. 8
2.3. Robot dotado de orugas. 8
2.4. Robot bípedo humanoide. 9
2.5. Robot reptador. 9
2.6. Robot nadador con forma de pez. 10
2.7. Quadracoptero. 10
2.8. Avión espía no tripulado o *drone* 11
2.9. Sensor láser Hokuyo. 13
2.10. Sensor de distancias por sonar. 14
2.11. Cámara estereoscópica. 15
2.12. Filtrado de datos anómalos y extracción de línea por RANSAC. 20
2.13. Esquema de un algoritmo de SLAM. 22
2.14. Estimación de la trayectoria óptima por un algoritmo de muestreo. 25
2.15. A la izquierda, casillas 4-conectadas. A la derecha, casillas 8-conectadas. 26
2.16. Órden en el que se despliegan los nodos (BFS) 27
2.17. Órden en el que se despliegan los nodos (DFS) 28
2.18. Los primeros controladores. 30
2.19. Controladores a finales de los años 80. 30
2.20. Controladores a principios de los años 90. 31
2.21. Power Glove, de Nintendo. 31
2.22. Activator, de Sega. 32
2.23. Zapper, de Nintendo 33
2.24. Controladores a finales de los años 90. 34
2.25. Plataforma de baile para el juego *Dance, Dance, revolution.* . 35
2.26. Eyetoy, de Sony 36
2.27. Desert Eagle, de Trustmaster. 36
2.28. Mandos a finales de la década pasada. 37
2.29. Kinect, de Microsoft. 38
2.30. Distribucion de colores en un filtro de Bayer. 40
2.31. Componentes de Kinect 41

2.32. Interfaz de ordenador controlada por gestos con Kinect. 43
2.33. Estructura del *framework* OpenNI. 44
2.34. Múltiples ejes de coordenadas en un robot. 47
2.35. Varias estructuras 3D formadas a partir de nubes de puntos. . 48
2.36. NITE en funcionamiento. 49
2.37. Dispositivos que cumplen el estándar OpenNI. 49
2.38. Esquema de ejecución de RGBDSLAM. 50
2.39. Estructura en capas de un grafo. 51
2.40. Puntos de interés extraidos y sus orientaciones. 53
2.41. Estado actual del proyecto. 54
2.42. Estimación del volumen de código desarrollado con Eclipse. . . 55

3.1. Diálogo inicial de Wubi. 57
3.2. Diálogo principal de la instalación. 57
3.3. Instalación finalizada. 58
3.4. Panel de control de Windows. 58
3.5. La instalación se selecciona como un programa normal. 59
3.6. Diálogo de confirmación 59
3.7. La desinstalación ha sido completada. 60
3.8. Selección de los repositorios. 60
3.9. Agregar un nuevo repositorio. 61
3.10. Ejecución ejemplo del programa *turtlesim* 64
3.11. Estructura en árbol del conjunto de ejes que participan en *turtlesim* 64
3.12. Captura de una ejecución del programa RGBDSLAM 66
3.13. Configuración de las variables del entorno. 69
3.14. Configuración de las variables de ejecución. 70
3.15. Representación de un posible árbol de búsqueda. 73
3.16. Camino estimado por el algoritmo. 76
3.17. Casillas sobrantes. 76
3.18. Casillas sobrantes. 77
3.19. Datos obtenidos de manera empírica. Las casillas tienen una dimensión de 40x40 centímetros.
3.20. Información disponible. 78
3.21. Distribución de casillas en el cono de visión. 79
3.22. Esquema del funcionamiento del control de accesos. 82
3.23. Especificaciones técnicas. 85
3.24. Laberinto en el que se realizan las pruebas del programa. . . . 87
3.25. Modelado correcto del laberinto. 88
3.26. Iteración inicial. De izquierda a derecha: captura de la cámara, información del robot y desplaz
3.27. Segunda iteración. 89
3.28. Tercera iteración. 90
3.29. Novena iteración. 91

3.30. Laberinto completado. 91