



Escuela
Politécnica
Superior

SLAM para robots domésticos



Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:

Valderico Carratalá Rizzo

Tutor/es:

Miguel A. Cazorla Quevedo

Félix Escalona Moncholí

Junio 2021



Universitat d'Alacant
Universidad de Alicante

SLAM para robots domésticos

Autor

Valderico Carratalá Rizzo

Tutor/es

Miguel A. Cazorla Quevedo

Departamento de Ciencia de la Computación e Inteligencia Artificial

Félix Escalona Moncholí

Departamento de Ciencia de la Computación e Inteligencia Artificial



Grado en Ingeniería Robótica



Escuela
Politécnica
Superior



Universitat d'Alacant
Universidad de Alicante

ALICANTE, Junio 2021

Preámbulo

“La razón por la que se ha llevado a cabo este proyecto, es el interés personal hacia los robots móviles y a las nuevas tecnologías. La finalidad es poder teleoperar de manera autónoma un robot, con únicamente la información de la odometría, de unos sensores láser y de una cámara RGBD para la creación de un mapa de la zona, con reconocimiento de los objetos presentes en la escena.”

Agradecimientos

Este trabajo no se habría podido llevar a cabo de no haber sido por la entera disposición de mis tutores Miguel Cazorla Quevedo y Félix Escalona Moncholí, bajo cuya supervisión escogí el tema de este trabajo de fin de grado, y gracias a los cuales pude ir avanzando a lo largo de las diferentes etapas de este proyecto.

También me gustaría agradecer la ayuda de mis compañeros de clase, los cuales sirvieron para resolver diversas dudas que me fueron surgiendo a lo largo del proyecto, así como para aconsejarme sobre el avance del mismo.

Por último, no puedo terminar sin agradecer a mi familia, cuyo estímulo constante y amor me ha servido como apoyo incondicional a lo largo de mis años en la Universidad. Estoy agradecido también a mis profesores de instituto y de la universidad, los cuales me motivaron a estudiar, y a llegar a donde me encuentro ahora mismo.

Es a ellos a quien dedico este trabajo.

*A mis abuelos Valderico Rizzo Aguirre y M^a del Carmen de Pazos Romero,
apoyos constantes sin los cuales no habría llegado hasta aquí.*

*"Nuestras virtudes y nuestros defectos son inseparables, como la fuerza y la materia.
Cuando se separan, el hombre ya no existe".*

Nikola Tesla.

Índice general

1	Introducción	1
2	Estado del arte	3
3	Objetivos y motivación	5
4	Metodología	7
4.1	Gmapping	7
4.2	YOLOv3	9
4.3	ROS	10
4.4	Robot Turtlebot	11
4.4.1	Kinect	13
4.4.2	Láser	14
5	Desarrollo	15
5.1	Aproximación	15
5.2	Turtlebot3	16
5.3	Modificación de Gmapping	16
5.4	YOLO	17
5.5	ROS	18
5.5.1	Nodo Gmapping	18
5.5.2	Nodo principal 1 - Darknet_ROS y sincronización de variables	18
5.5.3	Nodo principal 2 - Procesamiento de nubes y creación del mapa 3D	19
5.5.4	Nodo principal 3 - Generación de los bounding boxes 3D	20
6	Experimentación	23
6.1	Gmapping	23
6.2	You Only Look Once (YOLO)	23
6.3	Procesamiento de las nubes de puntos	24
6.4	Creación del mapa 3D	25
6.5	Resultado final	26
7	Conclusiones	31
8	Trabajos Futuros	33
	Bibliografía	35

Índice de figuras

1.1	Mapa de ocupación obtenido mediante el paquete Gmapping de ROS	1
4.1	Mapa de ocupación map.pgm	7
4.2	Imagen devuelta por YOLO con las predicciones y bounding boxes	10
4.3	Estructura Publisher-Subscriber de ROS	11
4.4	Imagen del robot Turtlebot3 Burger. Obtenida de ROBOTIS (2021)	12
4.5	TF del robot Turtlebot3.	13
4.6	Imagen del sensor Kinect de Microsoft. Obtenida de Xu y Lee (2012)	13
4.7	Imagen del área detectable del URG-04LX de Hokuyo. Obtenida de <i>Hokuyo URG-04LX Datasheet</i> (s.f.)	14
4.8	Timing de la señal de sincronización del URG-04LX de Hokuyo. Obtenida de <i>Hokuyo URG-04LX Datasheet</i> (s.f.)	14
5.1	Diagrama de funcionamiento del sistema.	15
5.2	Imagen devuelta por YOLO.	17
5.3	Datos de los bounding boxes devueltos por YOLO.	18
5.4	Matriz de transformaciones utilizada para rotar y trasladar las nubes de puntos.	19
5.5	Generación de los bounding boxes 3D, funcionando de manera síncrona al resto de nodos.	21
6.1	Distintas pruebas realizadas con la configuración de parámetros de gmapping	23
6.2	Pruebas del voxelizado con tamaño de hoja de 0.1 , 0.05 y 0.015 respectivamente	24
6.3	Pruebas de RANSAC con thresholds de 0.09 y 0.05 respectivamente	25
6.4	Mapeado 3D de la escena, con fallos en el posicionamiento de las nubes de los objetos debido a giros o movimientos a velocidades altas	26
6.5	Escenario con diversos objetos, para comprobar el funcionamiento del sistema	26
6.6	Mapas 2D y 3D junto a los bounding boxes detectados en instantes determinados de tiempo.	27
6.7	Mapa de ocupación 2d generado por Gmapping	28
6.8	Mapa 3D generado tras dar una vuelta al recinto con velocidades bajas/medias	28
6.9	Mapa generado combinando la información 2D y 3D, visto desde perspectivas diferentes	29

1 Introducción

Hoy en día, la obtención y/o generación de mapas, es un proceso muy importante para el ámbito de la robótica móvil, sobre todo cuando se desea obtener un robot operado de manera autónoma sin la necesidad de un operario que controle constantemente el movimiento del robot. Una vez se ha obtenido el mapa de la zona, ya sea estático o dinámico, se podría proceder a realizar con relativa facilidad la navegación automática del robot. Para ello, existen tres tipos básicos de mapas: mapas de representación espacial, de representación geométrica o de representación topológica. Cada uno de ellos tiene sus pros y sus contras, pero los más utilizados actualmente son los mapas de representación espacial basados en rejillas de ocupación. Estos mapas se basan en que dividen el entorno que rodea al robot en un conjunto de celdas, donde cada una de ellas se rellena con un valor determinado, para indicar si está ocupada o no, o si, en su defecto, no se conoce la información suficiente de dicha casilla.

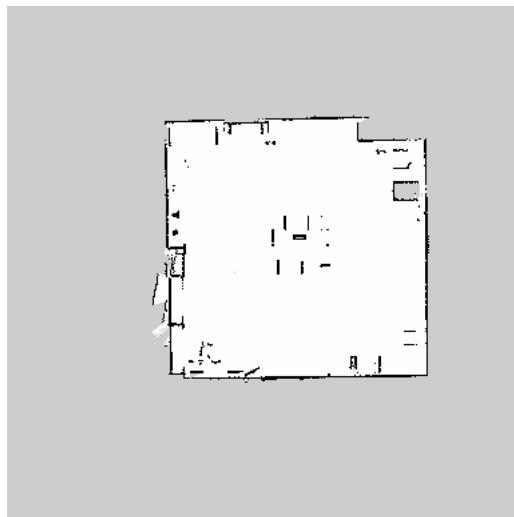


Figura 1.1: Mapa de ocupación obtenido mediante el paquete Gmapping de ROS

El problema de los mapas de ocupación, es que son capaces de discretizar el entorno, pero no de representar los objetos incluidos en él. Es por ello que deseamos conseguir, aplicando técnicas de inteligencia artificial para la detección de objetos, un sistema capaz de generar un mapa del entorno y guardar además, la información de los objetos de la escena, ya sea la posición de los objetos en 2D, como una nube de puntos con la representación de los mismos. Para ello, utilizaremos la red neuronal YOLO (You Only Look Once), la cual, una vez entrenada, es capaz de recibir una imagen y devolvernos el número de objetos detectados en la misma, así como las coordenadas de los objetos dentro de la imagen y la clase a la que pertenecen. Se usará esta red para localizar los objetos en las imágenes detectadas por el

robot móvil, y generar a partir de ellos un mapa 3D con los objetos de la escena, previamente segmentados. Con todo esto, se desea crear un sistema capaz de crear mapas con una mayor cantidad de información, combinando la información de un mapa de ocupación, con el mapa 3D generado a partir de los objetos detectados. La motivación de dicho proyecto, es que al obtener un mapa con mayor información, se podría facilitar la navegación automática de un robot móvil, o la manipulación de objetos, así como a la vez obtener una mayor cantidad de información del entorno del robot, de manera que podamos conocer en todo momento los objetos que hay presentes en una escena.

Es por ello que la finalidad inicial de este proyecto es crear un sistema de generación de mapas 2D y 3D, el cual sea capaz de incluir información de los objetos de la escena, mediante la modificación para ello del paquete de ROS Gmapping (Grisetti y cols., 2007), y la inclusión de sistemas de inteligencia artificial para la detección de objetos. Más adelante se implementará también un sistema de auto navegación en un robot móvil, en este caso el robot Pepper de Aldebaran and Softbank Robotics. Sin embargo, debido a las restricciones a raíz del COVID, se procedió a realizar el proyecto enteramente en simulación, mediante el robot Turtlebot3 de Open Source Robotics Foundation, y a basarse más en la implementación del sistema generador del mapa.

Así pues, en la Sección 2, se habla del status quo de la generación de mapas, explicando varios ejemplos de proyectos cuyo objetivo es similar al de este. En la Sección 3, se proponen todos y cada uno de los objetivos que se pretenden alcanzar en la realización de este trabajo, tanto personales, como el aprendizaje y la mejora de lenguajes de programación, como del trabajo en sí a la hora de hacer funcionar el sistema. En la Sección 4, hablamos sobre la metodología empleada, es decir, los elementos que utilizamos para llevar a cabo este trabajo. En la siguiente, Sección 5, exponemos cómo hemos utilizado esos elementos y herramientas aplicados al proyecto. Después, pasamos a la Sección 6, donde haremos diferentes pruebas del sistema en funcionamiento para comprobar sus límites y cuándo funciona mejor. Una vez explicado todo, en la Sección 7 realizaremos una serie de conclusiones, haciendo referencia a todo el transcurso del proyecto. Para terminar, en la Sección 8, expondremos las ideas de futuro que se han pensado llevar a cabo teniendo en cuenta el estado final del proyecto.

2 Estado del arte

Con la creciente necesidad actual de usar robots de servicio en interiores, hay una gran cantidad de trabajos que estudian la percepción del entorno en el que opera el robot. Es por ello que la generación de mapas es un tema muy importante para la robótica, en especial para los robots móviles, y un aspecto muy importante es el reducir todo lo posible los errores a la hora de generar dicho mapa.

Un ejemplo básico y conocido de esto, sería el del algoritmo SLAM, el cual es capaz de generar un mapa del entorno, y a la vez estimar la pose del robot dentro del mismo, todo esto en tiempo real.

Hay una gran cantidad de trabajos en este campo, así como métodos basados en láser, como el algoritmo de gmapping (Grisetti y cols., 2007) que utilizamos en este tfg, métodos basados en visión mediante cámaras, como por ejemplo ORB-SLAM (Mur-Artal y Tardós, 2016), o ElasticFusion (Whelan y cols., 2016), etc.

Los métodos anteriormente comentados, se basan más en la discretización del espacio en el que trabaja el robot, es decir, en las propiedades geométricas del mismo, pero no tienen en cuenta para ello los objetos que se encuentran dentro del espacio, por lo que para tareas que requieran un nivel superior de interacción entre el entorno y el robot, como puede ser la manipulación de objetos por ejemplo, estos métodos pueden ser insuficientes. Esto se debe a que además de tener una discretización del espacio donde opera el robot, también sería necesario obtener información de los objetos del entorno, ya sea la capacidad a secas de detectar su posición, como la de identificar el tipo de objeto que se trata, así como lógicamente su posición en el espacio 3D.

Para ello, hablaremos en primer lugar de algunos métodos que han estudiado la segmentación semántica de los objetos, basado en imágenes RGB y de profundidad, pero el propósito de la mayoría de dichos trabajos es únicamente el de comprender los elementos que componen las imágenes 2D, no el de identificar la posición de dichos objetos en el espacio 3D para mapear un entorno.

Los investigadores Lin y cols. (2013) fueron los primeros que propusieron un método para generar bounding boxes 3D (cuadros delimitadores) alrededor de los objetos. Dicho método utiliza para ello el algoritmo de Constrained Parametric Min-Cuts (CPMC) con características de apariencia 2D, y lo consiguieron expandir a un framework 3D para generar cuboides de los objetos. Para ello, las relaciones entre los objetos, y la escena y los objetos se incorporan en un modelo Conditional Random Field (CRF), el cual se encargaba de realizar la inferencia de las etiquetas semánticas, clasificando así los cuboides. De esta manera, la formulación, clasificación de escenas, y el reconocimiento 3D están acoplados en un mismo sistema, de manera que pueden resolverse de manera conjunta mediante inferencia probabilística.

Otra gran cantidad de trabajos de detección de objetos en 3D, se inspira cada vez más en el uso de redes neuronales convolucionales para detectar objetos en 2D. Los investigadores Song y Xiao (2016) diseñaron una red convolucional, las cuales mediante imágenes RGB y de

profundidad como entradas, eran capaces de generar el bounding box 3D del objeto. Para ello, los valores iniciales que había que proporcionar a la red en la categoría detectada del objeto mediante la imagen RGB, y el volumen del objeto obtenido de la imagen de profundidad. Sin embargo, este enfoque tiene varios problemas para la configuración de una gran cantidad de parámetros de la estructura de la red, además de necesitar una gran cantidad de recursos para funcionar.

Otros investigadores como Z. Ren y Sudderth (2016) presentaron un descriptor de nubes de gradientes, el cual estaba orientado para clasificar bounding boxes 3D. Este método era capaz de fusionar las características contextuales de la imagen, pero debido a el conjunto de datos que utilizaba (SUN-RGBD), el tiempo que tardaba en calcular las características es muy alto, aunque consigue una alta precisión.

El último trabajo que mencionaremos es el de los investigadores Song y Xiao (2017), los cuales propusieron un método de detección de objetos 3D, el cual servía para orientar, colocar, y establecer una valoración a bounding boxes alrededor de objetos, utilizando para ello la información 2D de una imagen para reducir el espacio de búsqueda 3D. En este método, la información de los objetos obtenida de la imagen 2D se utiliza para detectar los objetos, sus ubicaciones y tamaños, y mediante un perceptrón multicapa (MLP) se aprenden dichas características. Más adelante refinan las detecciones realizadas basándose en las relaciones de las categorías detectadas de los objetos en una escena. Sin embargo, dicho método introduce algo de error, al no ser capaz de eliminar correctamente algunos puntos de fondo de la nube 2D, lo cual puede resultar en la determinación inexacta de la posición u orientación del objeto.

En la mayoría de estos métodos se utilizan como se puede ver en su mayoría únicamente imágenes RGB o RGB-D para realizar la segmentación de los objetos y crear el mapeado. Sin embargo, el enfoque que seguiremos para nuestro método propuesto es el de obtener primero los bounding boxes de los objetos a partir de una imagen RGB-D mediante una red neuronal de detección de objetos, y de posteriormente combinar esta información con la del algoritmo de gmapping, para que funcionen conjuntamente. De esta manera, al hacer en tiempo real tanto el mapeado del escenario, como la obtención de los bounding boxes de los objetos, se puede permitir una mejor estimación de la posición de dichos objetos en la escena. Por lo tanto se permitirá crear tanto un mapa 2D de rejilla de ocupación para realizar la navegación por el entorno, como un mapa 3D, el cual incluya tanto la información de la posición y orientación de los objetos de la escena, como de la clase a la que pertenecen, permitiendo así obtener un mapa con mayor información, el cual puede resultar útil para tareas más complejas.

3 Objetivos y motivación

La motivación personal para este trabajo se basa en el aprendizaje de nuevas técnicas de programación, y la combinación de algoritmos conocidos de mapeado con técnicas de inteligencia artificial y sensores de captación de nubes de puntos. De esta forma, la motivación para el desarrollo de este trabajo ha sido:

- Mejorar el nivel del lenguaje de programación Python y C++.
- Emplear técnicas de inteligencia artificial en el proyecto.
- Profundizar en el uso de ROS.

Los objetivos de este trabajo se pueden resumir en la siguiente lista:

- Integrar y programar diferentes subsistemas para lograr un sistema capaz de generar un mapa 2D con la integración de información de objetos 3D previamente detectados por una IA.

4 Metodología

En esta sección hablaremos de cada uno de los componentes y herramientas que hacen funcionar nuestro sistema.

4.1 Gmapping

Para la generación de mapas, una técnica muy común utilizada para obtener un mapa cuando no conocemos el entorno es Simultaneous Localization And Mapping (SLAM).

Esta técnica es muy utilizada en la robótica para la construcción de mapas del entorno, ya que tiene la ventaja de que permite la construcción del mapa y el localizamiento del robot de manera simultánea en tiempo real, mientras el robot se está moviendo por el escenario. Es por ello que el objetivo principal de gmapping consigue en generar un mapa del entorno, reduciendo en gran medida el error del mapa generado, haciendo que sea muy útil para tareas de navegación en espacios sin cambios de elevación.

La construcción de dicho mapa puede resultar muy compleja, pero gracias a los paquetes que contiene ROS, podemos utilizar esta técnica de manera muy sencilla.

Para ello, ROS nos proporciona el paquete gmapping¹, el cual nos permite utilizar el algoritmo de SLAM utilizando como sensor un láser (u otro sensor equivalente al láser, como, por ejemplo, el Kinect) montado a la base del robot. Para ello, gmapping utiliza un nodo de ROS llamado slam_gmapping, el cual teniendo los datos del láser y la odometría del robot, es capaz de construir un mapa de ocupación 2D (OGM). En la Figura 4.1 se puede ver el mapa construido por Gmapping tras dar una vuelta a una habitación.

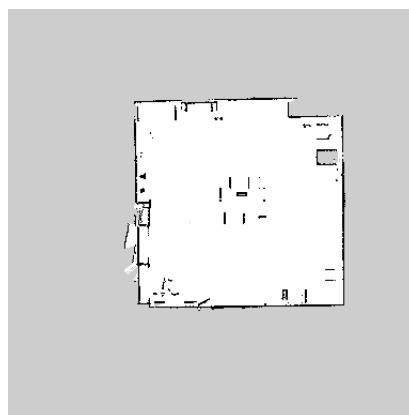


Figura 4.1: Mapa de ocupación map.pgm

¹En <http://wiki.ros.org/gmapping> se puede encontrar información de dicho algoritmo

Para ello, utilizaremos una variante del problema llamada Full SLAM, adaptada para el robot Turtlebot3, lo cual comentaremos más adelante. Esta variante se diferencia en que guarda los valores anteriores de la trayectoria para obtener unos mejores resultados (smoothing). De esta manera, se podría conseguir que durante una primera pasada del robot por el entorno, este obtuviese el mapa, y una vez hecho esto, el robot ya podría comenzar a navegar de manera autónoma esquivando obstáculos por el entorno.

Esta técnica también tiene ciertos inconvenientes, como por ejemplo, que para la construcción del mapa, es necesario tener una gran precisión sobre la posición del robot (odometría), o que si el mapa es dinámico, habría que estar constantemente actualizando el mapa en busca de cambios, o utilizar otras técnicas de control para complementar el mapa inicial y ser así capaz de esquivar los obstáculos dinámicos.

El mapa por su parte, se publica en el topic /map, y utiliza el mensaje nav_msgs/OccupancyGrid. Dicho mapa representa la ocupación del entorno, de manera que divide el entorno un conjunto de celdas, donde cada una de ellas puede tomar un valor dentro de un rango de 0 a 100. Siendo 0 una casilla completamente libre, y 100 una casilla completamente ocupada. Además, existe un valor especial -1, el cual indicaría que no se conoce nada de una celda.

El comando para lanzar Gmapping desde una terminal es el siguiente:

Código 4.1: ejecutar gmapping

```
1 $ rosrun gmapping slam_gmapping scan:=base_scan
```

Una vez ejecutado el comando este procederá a leer los datos del láser (base_scan) y a generar el mapa automáticamente. En este punto los ideal es mover robot a través del entorno para ir generando el mapa y después guardarlo. Para guardar el mapa generado debemos utilizar el siguiente comando:

Código 4.2: guardar el mapa

```
1 $ rosrun map_server map_saver -f <map_name>
```

Este comando generará dos ficheros con la información del mapa: map.pgm y map.yaml. El primero es una imagen con la información del mapa generado 4.1, mientras que el segundo es un fichero donde se incluye toda la información del mapa generada por Gmapping.

La imagen representaría los espacios libres (en blanco) y los espacios ocupados (en tono grises) detectados por la aplicación.

El segundo fichero map.yaml, representa la información del mapa, su contenido debería ser similar a este:

Código 4.3: código dentro del fichero map.yaml

```
1 image: /home/ludo/map.pgm
2 resolution: 0.050000
3 origin: [-10.000000, -10.000000, 0.000000]
4 negate: 0
5 occupied_thresh: 0.65
6 free_thresh: 0.196
```

Donde:

- resolution: Indica la resolución del mapa, metros/pixel.
- origin: Las posición del píxel en 2D de la esquina inferior izquierda.

- occupied_thresh: Grado de probabilidad para considerar que celda esta totalmente ocupada.
- free_thresh: Píxeles con probabilidad mejor que el umbral para ser considerados completamente libres.

En nuestro caso, como bien hemos comentado anteriormente, utilizaremos una variante del algoritmo Full Slam adaptada para el robot Turtlebot3. Para ello, se ha descargado el paquete de turtlebot3_slam², el cual funciona de manera análoga al metodo de Full Slam comentado, pero ya está adaptado para funcionar en simulación con el robot Turtlebot3.

4.2 YOLOv3

YOLO es un sistema de detección de objetos en tiempo real de última generación. YOLOv3 es extremadamente rápido y preciso. Aplica una única red neuronal a la imagen completa. Esta red divide la imagen en regiones y predice cuadros delimitadores (bounding boxes) y probabilidades para cada región. Estos cuadros delimitadores están ponderados por las probabilidades predichas.

Este modelo tiene varias ventajas sobre los sistemas basados en clasificadores. Mira la imagen completa en el momento de la prueba, por lo que sus predicciones se basan en el contexto global de la imagen. También hace predicciones con una única evaluación de red, a diferencia de sistemas como R-CNN, que requieren miles para una sola imagen. Esto lo hace extremadamente rápido, más de 1000 veces más rápido que R-CNN y 100 veces más rápido que Fast R-CNN.³

Para ejecutar YOLO con la información que devuelva la cámara del robot en ROS, bastaría con utilizar el paquete de Darknet adaptado a ROS por LegedRobotics⁴, el cual al añadirlo a nuestro espacio de trabajo de ROS nos permitirá ejecutar YOLO con la información del topic /camera/rgb. Para ello, únicamente habrá que ejecutar el siguiente comando desde terminal, habiendo lanzado previamente el robot en simulación:

Código 4.4: lanzar YOLOv3 en ROS

```
1 $ roslaunch darknet_ros yolo_v3.launch
```

Una vez hecho esto, se lanzara YOLO, y tras cargar los pesos generados en el entrenamiento de la red, procederá a imprimir los objetos que detectó, su confianza y cuánto tiempo tardó en encontrarlos. En la Figura 4.2 se muestra la imagen devuelta por YOLO con las predicciones y bounding boxes de los objetos detectados.

Código 4.5: Información devuelta por la terminal

```
1 FPS:0.2
2 Objects:
3
4 person: 35%
5 chair: 52%
```

²En <https://github.com/ROBOTIS-GIT/turtlebot3> se pueden encontrar los diferentes paquetes para lanzar correctamente el robot Turtlebot3, como para el algoritmo de Slam que utilizaremos

³Cita sacada de <https://pjreddie.com/darknet/yolo/>

⁴En https://github.com/leggedrobotics/darknet_ros se puede encontrar el repositorio de dicho paquete



Figura 4.2: Imagen devuelta por YOLO con las predicciones y bounding boxes

4.3 ROS

La comunicación entre los elementos es un factor clave del sistema, por lo que vamos a explicar la forma que hemos elegido para establecer esta comunicación. A continuación, vamos a hablar de ROS (Robot Operating System). ROS es un framework que nos permite trabajar con robots a través de lo que se denominan mensajes. Estos mensajes son datos, y se envían a través de canales entre dos nodos.

La estructura general de ROS es la siguiente: un nodo publica un mensaje en un canal y otro nodo se suscribe a ese mismo canal para recibir dicho mensaje. No obstante, hay variaciones en el concepto de "canal", concretamente, tres, Topics, Servicios y Acciones. Los Topics implementan un mecanismo de comunicación publicación-suscripción constante, es decir, el nodo publicador publica todo el rato los datos que queremos enviar y el nodo suscriptor recoge esos datos y los utiliza en su código. Por otro lado, los Servicios, pese a que el funcionamiento es muy similar al de los Topics, tienen una diferencia clave, y es que, cuando envía los datos al suscriptor, necesita que este le devuelva un resultado como feedback para poder continuar el programa, en otras palabras, es una estructura síncrona. Por último, las Acciones, que son iguales a los Servicios, pero estas son asíncronas, lo que nos permite enviar los datos y seguir con el programa del publicador mientras que el suscriptor ejecuta el suyo y, una vez acabe, enviarle la respuesta al publicador. Teniendo presente estas tres opciones, nos decantamos por los Topics. Esta elección se debe básicamente a que no necesitamos respuesta del suscriptor, simplemente queremos enviar datos y que el suscriptor los procese y ejecute según su programa. Otro motivo es que, los Servicios, al tener que esperar la respuesta para seguir ejecutando son poco prácticos, ya que si quisieramos enviar la acción de parar porque hemos visto un obstáculo, hasta que no se termine la orden anterior, no nos haría

caso. Así pues, mostramos en la Figura 4.3 la estructura que sigue un sistema basado en un publicador-suscriptor mediante Topics de ROS:

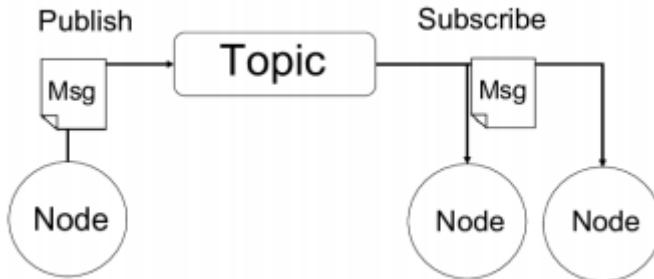


Figura 4.3: Estructura Publisher-Subscriber de ROS

En la Figura 4.3 podemos diferenciar 5 elementos clave:

- Node: Un nodo es un ejecutable que usa ROS para comunicarse con otros nodos.
- Msg: Tipo de dato de ROS que es utilizado durante la suscripción y publicación de un topic.
- Topic: Es el canal por donde circulan los mensajes. Los nodos pueden publicar mensajes a través de un topic. Se puede suscribir a ellos para recibir mensajes.
- Publisher: Es un tipo de nodo que publica los mensajes en ese Topic.
- Subscriber: Es un tipo de nodo que recibe los mensajes a través de ese Topic.

Como hemos visto, mediante ROS, podemos enviar mensajes de un nodo a otro con la información que queramos. Por eso mismo hemos elegido esta manera de establecer la comunicación entre los elementos del sistema.

4.4 Robot Turtlebot

El Turtlebot es un robot diseñado por Open Source Robotics Foundation, diseñado como un kit de robot personal de muy bajo costo, y de código abierto. Este robot está dotado de un sistema de movilidad diferencial, y de diversos sensores, los cuales permiten realizar de manera sencilla un robot móvil, con la capacidad de obtener datos mediante diferentes sensores y tener la suficiente potencia para crear aplicaciones interesantes. Además, al ser la estructura del robot modular, permite que implementar nuevos sensores de manera sencilla.

El TurtleBot3 se puede personalizar de varias formas dependiendo de cómo reconstruya las partes mecánicas y use partes opcionales como la computadora y el sensor. Además, TurtleBot3 se ha desarrollado con SBC rentable y de tamaño pequeño que es adecuado para un sistema integrado robusto, sensor de distancia de 360 grados y tecnología de impresión 3D.

La tecnología central de TurtleBot3⁵ es SLAM, navegación y manipulación, lo que lo hace adecuado para nuestro proyecto. El TurtleBot puede ejecutar algoritmos SLAM (localización y mapeo simultáneos) para construir un mapa y puede conducir por su habitación. Además, si fuera necesario, el TurtleBot3 se puede utilizar como un manipulador móvil capaz de manipular un objeto adjuntando un manipulador como OpenMANIPULATOR. A continuación, se muestra en la Figura 4.4 la estructura del robot utilizado.

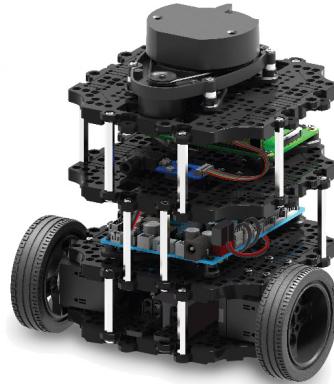


Figura 4.4: Imagen del robot Turtlebot3 Burger. Obtenida de ROBOTIS (2021)

Así pues, en nuestro caso usaremos el modelo Burger, el cual dispone de un sensor láser en su base para realizar el mapeado del entorno, así como una cámara Kinect RGBD, la cual nos permite ejecutar la red neuronal YOLO para detectar objetos, así como también obtener las nubes de puntos del entorno, como la de los objetos deseados.

Para ser capaz de ejecutar el turtlebot desde ROS, utilizamos el paquete creado por Robotis-Git, <https://github.com/ROBOTIS-GIT/turtlebot3>, el cual nos permite lanzar el robot en simulación de manera sencilla, y aparte incluye paquetes como turtlebot3_slam, los cuales nos facilitarán parte del proyecto.

Para lanzar el turtlebot3 en simulación en gazebo, es tan sencillo como escribir el siguiente comando:

Código 4.6: lanzar turtlebot3 en Gazebo

```
1 $ roslaunch turtlebot3_gazebo [name_of_launch_file].launch
```

Una vez hecho esto, si deseáramos ejecutar el algoritmo de SLAM, podríamos lanzarlo con el siguiente comando:

Código 4.7: lanzar OpenSlam en ROS

```
1 $ roslaunch turtlebot3_slam turtlebot3_slam.launch
```

Este último comando lanzará también el robot en Rviz, para que podamos observar en tiempo real la creación del mapa.

Una vez lanzado correctamente el turtlebot, su TF debería ser similar al de la Figura 4.5.

⁵En <https://github.com/ROBOTIS-GIT/turtlebot3> se pueden encontrar los diferentes paquetes para lanzar correctamente el robot Turtlebot3, así como algoritmos de SLAM, de navegación, teleoperación, etc

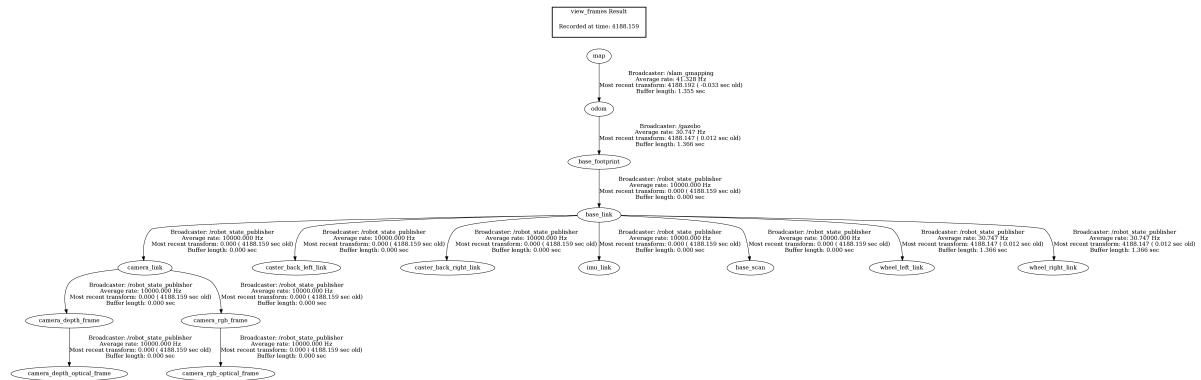


Figura 4.5: TF del robot Turtlebot3.

4.4.1 Kinect

El sensor Kinect, es un sensor diseñado por Microsoft, el cual es una cámara RGB-D, y dispone de 3 sensores, RGB, audio y profundidad. Esto sensor salio originalmente como complemento para la consola Xbox fabricada por Microsoft, pero debido a su funcionalidad, ha sido ampliamente usado en otras aplicaciones útiles en el área de la visión por computadora, como la robótica y el reconocimiento de acciones. El sensor Microsoft Kinect Sensor y sus componentes se muestran en la Figura 4.6.

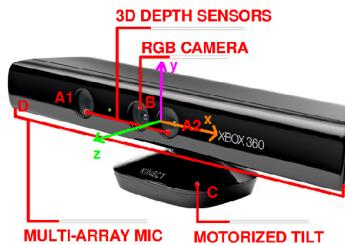


Figura 4.6: Imagen del sensor Kinect de Microsoft. Obtenida de Xu y Lee (2012)

La parte A es un sensor de profundidad, también llamado sensor 3D. Es una combinación de un láser infrarrojo con un sensor CMOS para permitir que el sensor Kinect procese escenas 3D en cualquier condición de luz ambiental. La distancia de las superficies de los objetos desde el punto de visibilidad de la cámara se especifica mediante el mapa de profundidad. Este sistema se denomina ToF (Time of Flight) porque establece el mapa de profundidad de la escena considerando el tiempo que tarda la luz en volver a la fuente después de que se emita desde el sensor y rebote en un objeto. El rango de profundidad óptimo del sensor es de 1,2 a 2,5 metros. La parte B es una cámara RGB que tiene una resolución de color de 32 bits. Puede usar vídeo en color bidimensional de la escena. La parte C es una articulación motorizada, la cual influye al campo de visión.

Por último, la parte D contiene una matriz de 4 micrófonos que se encuentra a lo largo de

la barra horizontal. Es útil para el reconocimiento de voz, la supresión de ruido ambiental y la cancelación de eco.

4.4.2 Láser

El sensor de rango láser utilizado por el turtlebot3 es el URG-04LX, de Hokuyo Automatic co. ltd, el cual es un sensor láser para escaneo de áreas. La fuente de luz del sensor es un láser infrarrojo de 785 nm de longitud de onda con láser de clase 1 de seguridad. El área de escaneo del láser es un semicírculo de 240° con un radio máximo de 4000 mm. El ángulo de inclinación es de 0,36° y el sensor genera la distancia medida en cada punto (683 pasos). La distancia de detección puede variar con tamaño y objeto. El diámetro del rayo láser es inferior a 20 mm a 2000 mm con una divergencia máxima de 40 mm a 4000 mm. El principio de la medición de la distancia se basa en el cálculo de la diferencia de fase, por lo que es posible obtener una medición estable con una influencia mínima del color del objeto y reflectancia. En la Figura 4.7 se puede observar el área detectable por dicho sensor láser.

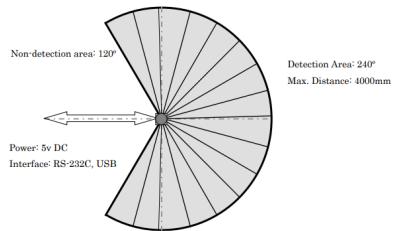


Figura 4.7: Imagen del área detectable del URG-04LX de Hokuyo. Obtenida de *Hokuyo URG-04LX Datasheet* (s.f.)

Además, se especifica en el datasheet del propio URGLX-04LX que el sensor está diseñado para uso en interiores únicamente, que no es un dispositivo o herramienta de seguridad y que no está diseñado para uso en aplicaciones militares.

A continuación, se muestra en la Figura 4.8 el timing de la señal de sincronización.

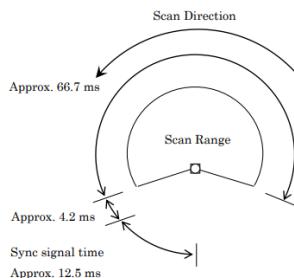


Figura 4.8: Timing de la señal de sincronización del URG-04LX de Hokuyo. Obtenida de *Hokuyo URG-04LX Datasheet* (s.f.)

Dicho sensor tiene un paso de datos máximo de 683 pasos. Dado que la resolución angular es 0.3515625° (360° / 1024 pasos) y el rango angular es 239.765625° ((683-1)×360/1024).

5 Desarrollo

En este capítulo vamos a explicar cómo utilizamos los elementos y herramientas explicadas anteriormente y de qué forma los integramos para lograr el funcionamiento de todo el sistema.

5.1 Aproximación

En la Figura 5.1 se muestra el funcionamiento completo de este proyecto de forma estructural, separando cada uno de los procesos más determinantes para su funcionamiento.

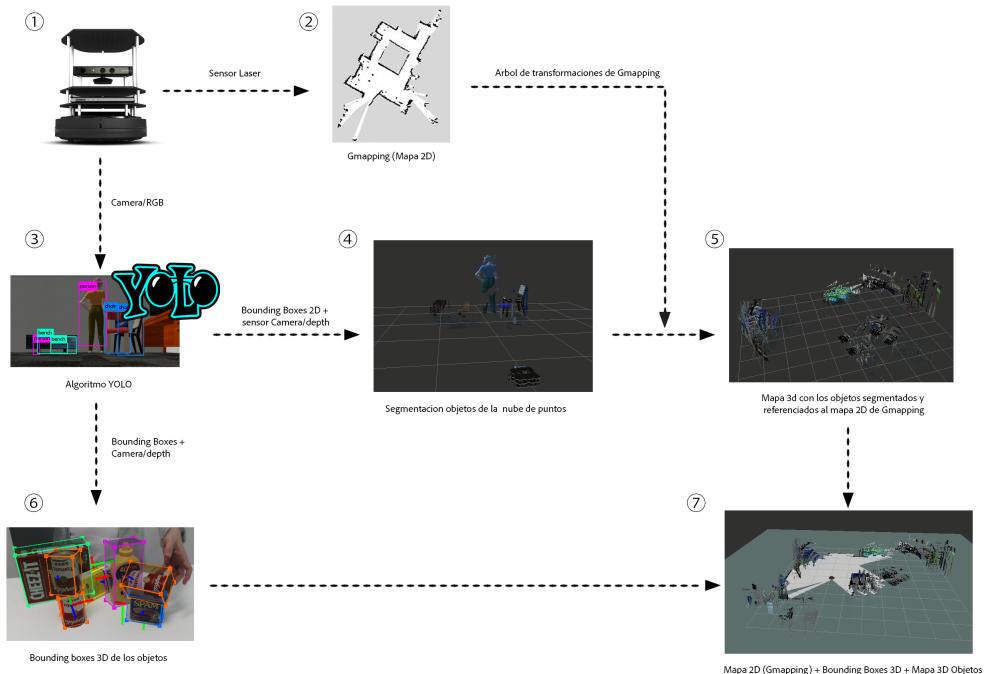


Figura 5.1: Diagrama de funcionamiento del sistema.

Como se aprecia en la imagen anterior, disponemos de diversos elementos, los cuales componen este proyecto. Además, el bucle de control sigue el orden de estos elementos. En primer lugar, el turtlebot (1) utiliza el sensor láser para ir realizando el mapeo 2D (2) del espacio de trabajo mediante el algoritmo de Gmapping. A su vez, se utiliza la información de la cámara RGBD para alimentar la red neuronal (3) YOLO (You Only Look Once), la cual es capaz de detectar en una imagen 2D los diferentes objetos que se encuentran en ella. Una vez obtenidos los cuadros delimitadores 2D (bounding boxes) de los objetos en la imagen, así como la clase

a la que pertenecen, estos se utilizan tanto para (4) segmentar los objetos de la escena de la nube de puntos que toma la kinect, como para (6) generar a partir de ahí los cuadros delimitadores 3D de los objetos. Además, una vez se tienen las nubes de puntos correctamente segmentadas y procesadas, (5) se transforman en relación al mapa de Gmapping, utilizando para ello el árbol de transformaciones que Gmapping genera. Con toda esta información, se busca (7) generar un mapa 3D de la escena con la información del mapa de ocupación 2D de gmapping, así como con la información de los objetos 3D.

5.2 Turtlebot3

El uso que se le da al Turtlebot es el siguiente: obtener las mediciones del láser para realizar el mapeado mediante el módulo de Gmapping, y a su vez utilizar la cámara kinect RGBD, para obtener tanto las imágenes 2D que capta la cámara, como las nubes de puntos del entorno.

Para realizar dicho comportamiento, se ha modificado el paquete original de Gmapping, así como se han desarrollado dos programas principales, dentro de los cuales se realizan las tareas de procesamiento y de comunicación de los diferentes datos necesarios, para lograr así el comportamiento final.

Lo primero que necesitamos para poder realizar el comportamiento, es el envío del árbol de transformaciones desde el nodo de Gmapping, al primer programa principal.

5.3 Modificación de Gmapping

En cuanto a la modificación realizada del algoritmo Gmapping, consiste en la generación previa de un tipo de mensaje custom de ROS, dentro del cual se procederá a guardar tanto el árbol de transformaciones que Gmapping va generando al actualizar el mapa, así como un identificador, el cual utilizaremos más adelante para ver cuando cambia el árbol de transformaciones, y generar así el mapa 3D con las nubes de puntos.

Así pues, se crearon dos mensajes de tipo custom de ROS, el primero (TreeVariables.msg) almacena las coordenadas (x,y,theta) de los diferentes tNodes de Gmapping, los cuales son un tipo de datos, que entre otras cosas, almacenan las coordenadas por las que ha ido pasando el robot en referencia al frame del mapa, que es lo que a nosotros nos interesa. El segundo mensaje custom (TreeInfo.msg) almacena en un vector las diferentes coordenadas de tipo TreeVariables, y también el identificador comentado anteriormente, para el cual en nuestro caso utilizaremos la dirección de memoria de los tNode.

Una vez creados los mensajes, se procede desde el nodo de Gmapping donde se realiza la actualización del mapa, a crear una variable de cada tipo de dichos mensajes, a inicializarlas a los valores adecuados, y posteriormente a publicarlos en un topic determinado, el cual utilizaremos más adelante para transformar las nubes de puntos respecto al mapa, así como para mediante el identificador, ver cuando es necesario actualizar dicho mapa.

5.4 YOLO

YOLO es un sistema utilizado para detección de objetos en tiempo real. Para ello como bien se ha comentado antes, YOLO aplica una única red neuronal a la imagen completa. Esta red divide la imagen en regiones y predice cuadros delimitadores (bounding boxes) y probabilidades para cada región. Estos cuadros delimitadores están ponderados por las probabilidades predichas. En primer lugar necesitamos lo que se conoce como dataset, es decir, una base de datos de imágenes. En nuestro caso utilizaremos el dataset COCO, el cual es un dataset conocido y está ya construido, para ahorrarnos el tiempo de la creación del dataset. Dicho dataset incluye imágenes de hasta 80 objetos diferentes, dentro de los cuales se encuentran objetos comúnmente utilizados en el día a día, como pueden ser sillas, sofás, mesas, pantallas de ordenador, personas, etc. Para hacer que funcione entonces YOLO, descargamos dicho dataset, modificaremos las rutas de YOLO para que reconozca la ubicación del dataset, y procederemos a entrenar la red neuronal con dichos datos. Una vez hecho esto, podemos ya lanzar el algoritmo de YOLO, para lo cual utilizamos el paquete Yolo for Ros (Darknet_Ros), el cual hace lo hace funcionar en conjunto con el framework de ROS. Para ello dentro del paquete Darknet_Ros, sustituimos la carpeta donde se encuentra YOLO, por nuestra versión entrenada con el dataset de COCO, y únicamente faltaría indicar el topic del cual leerá las imágenes, generandonos al ejecutarlo los topics con la información de los bounding boxes de los objetos detectados, cambiando para ello lo que sea necesario del fichero darknet_ros/config/ros.yaml. Una vez hecho esto, si ejecutamos el paquete Darknet_Ros, se abrirá una pestaña donde podremos visualizar lo que ve el robot, así como los bounding boxes 2D que se generan, lo cual se puede ver en la siguiente Figura 5.2

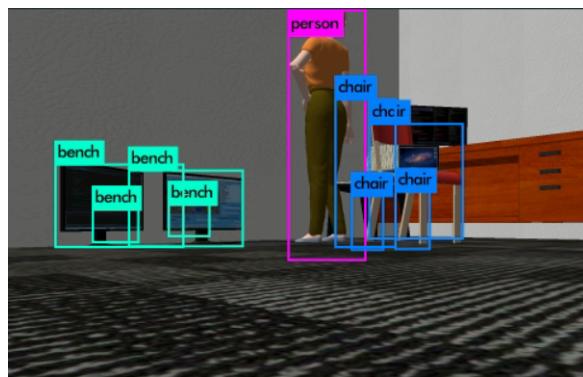


Figura 5.2: Imagen devuelta por YOLO.

A su vez, como se puede ver en la Figura 5.3, haciendo un rostopic echo del topic /darknet_ros/bounding_boxes, se pueden visualizar las coordenadas de dichos bounding boxes dentro de la imagen, así como la clase del objeto, y la probabilidad de que dicha suposición sea correcta.

```

    probability: 0.308493316174
    xmin: 1073
    ymin: 464
    xmax: 1155
    ymax: 598
    id: 56
    Class: "chair"

    probability: 0.325039327145
    xmin: 1183
    ymin: 454
    xmax: 1278
    ymax: 595
    id: 56
    Class: "chair"

```

Figura 5.3: Datos de los bounding boxes devueltos por YOLO.

5.5 ROS

En este apartado vamos a explicar cómo se utiliza el framework de ROS para este trabajo, así como los diferentes nodos utilizados para el procesamiento de los datos, y la creación de los diferentes mapas. Recordamos que ROS nos permite comunicar dos nodos mediante un canal, por el cual enviamos un mensaje. Para realizar dicha comunicación entre los diferentes nodos y topics, seguiremos el procedimiento descrito en la Figura 5.1, donde se pueden ver cómo se comunican los diferentes nodos, así como los datos que se transmiten entre ellos.

ROS en este proyecto, es una herramienta fundamental, ya que se utiliza para comunicar los diferentes elementos de este proyecto, para mediante cada uno de los nodos creados, obtener la información necesaria. Para ello, vamos a intentar explicar cómo funcionan los diferentes nodos y mensajes creados a continuación, por el orden en el que se usan en el proyecto. Para ello dividiremos el proyecto en tres nodos principales, el nodo de Gmapping, donde se construirá el mapa y se generará el árbol de transformaciones del robot, el segundo nodo principal (Python), en el cual se juntará la información del árbol de transformaciones de Gmapping, con la nube de puntos, y el resultado de YOLO, el segundo nodo principal (C++), dentro del cual se realizará el procesamiento de la nube de puntos, así como la creación del mapa final 3D, y por último el nodo donde se generarán los bounding boxes 3D de los objetos.

5.5.1 Nodo Gmapping

En primer lugar, desde el nodo de Gmapping, para conseguir crear el mapa 2D, es necesario suscribirse a los topics del sensor láser, así como declarar los diferentes publishers donde se publicará el mapa, o en nuestro caso, también se enviarán los datos del árbol de transformaciones junto a un identificador, como bien hemos comentado anteriormente. Esto último se realizará mediante un tipo de mensaje custom de ROS, el cual se publica en el topic/tree_vector, mientras que el mapa 2D generado por Gmapping se publicará en el topic /map .

5.5.2 Nodo principal 1 - Darknet_ROS y sincronización de variables

Una vez se ha iniciado el proceso de creación del mapa 2D , y se está publicando constantemente la información del árbol de transformaciones, dicha información se recibe desde un nodo python. Dicho nodo de python se encarga de procesar las imágenes que recibe el robot de la cámara RGBD desde el topic /camera_reading mediante YOLO . Una vez hecho esto, se

obtendrán los bounding boxes de los objetos detectados en dicha imagen, así como la clase de dichos objetos, y la probabilidad de que la detección sea correcta. Acto seguido, se procederá a obtener la nube de puntos que toma el robot mediante el sensor Kinect, obteniéndola del topic /camera/depth/points, y de posteriormente sincronizar los datos de dicha nube con los bounding boxes, y los datos proporcionados por Gmapping del árbol de transformaciones, almacenandolos para ello en un mensaje de ROS de tipo custom.

Una vez se tiene esta información almacenada en un mensaje, teniendo en cuenta que tanto la nube de puntos actual, como las coordenadas del robot en el momento que se tomó la nube (a partir del árbol de transformaciones), como las coordenadas de los diferentes objetos detectados en YOLO, así como la clase de los objetos, dicho mensaje se publica en el topic vector_nube y desde otro nodo de tipo C++, se realizará el procesamiento y segmentación de las nubes de puntos, así como de la creación del mapa 3D con los objetos del entorno.

5.5.3 Nodo principal 2 - Procesamiento de nubes y creación del mapa 3D

Acto seguido, desde un nuevo nodo de tipo C++, procederemos a suscribirnos al topic donde se publica el mensaje custom con las coordenadas del robot, y las nubes de puntos con las diferentes coordenadas de los objetos dentro de la misma, obtenido gracias a YOLO. Una vez suscritos, y gracias a la librería Point Cloud Library (PCL), procederemos a realizar el procesado y la transformación de las diferentes nubes de puntos.

Para ello, lo primero que hacemos es extraer la nube de puntos del mensaje custom, y la convertimos a un tipo de nubes de pcl::pointcloud2, para ser así capaces de procesarla y aplicarle los filtros necesarios. Una vez hecho esto, procedemos a extraer del mensaje custom las coordenadas de los diferentes objetos en la nube de puntos, proporcionadas por los bounding boxes devueltos por YOLO, y a segmentarlos de la nube de puntos global.

Para realizar dicha segmentación, por cada uno de los objetos detectados crearemos una nube de puntos de tipo pcl::pointcloud2, y copiaremos en ella los puntos que pertenezcan al interior de las coordenadas del bounding box correspondiente. Una vez extraído cada uno de los objetos, es necesario aplicarles una transformación de rotación y translación, con el fin de colocarlos en referencia al frame del mapa de Gmapping, y que por lo tanto las coordenadas de los objetos estén sincronizadas entre el mapa 2D y el 3D. Para realizar dicha transformación, utilizaremos las coordenadas del robot, obtenidas del árbol de transformación de Gmapping, y conociendo la relación entre el frame de la cámara Kinect y el del mapa, aplicaremos una matriz de traslación y rotación a las nubes, siendo así capaces de transformarlas correctamente. A continuación, en la Figura 5.4 se muestra la ecuación utilizada para relacionar los dos sistemas de coordenadas, de manera que se relacione el sistema de coordenadas del mapa y de la cámara mediante la R (rotación) y t (traslación):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \end{bmatrix} \begin{bmatrix} U \\ V \\ W \\ 1 \end{bmatrix}$$

Figura 5.4: Matriz de transformaciones utilizada para rotar y trasladar las nubes de puntos.

Antes de transformar dichos objetos respecto al frame de referencia del mapa, aplicamos un filtro RANSAC, así como una voxelización a las nubes de puntos, lo cual permitirá eliminar los planos dominantes de la escena, como los puntos de la pared o del suelo, y además reducirá el número de puntos de las nubes, haciendo que el procesado sea más rápido.

Una vez hecho esto, se aplica la matriz de transformación, y después de haber segmentado y transformado los diferentes objetos de la escena, solo queda juntar las diferentes nubes de los objetos ya segmentados en un mapa 3D global, volver a convertir la nube al tipo sensor_msgs::pointcloud2, y publicar dicha nube global en un tópico, en mi caso /nube_objeto para poder visualizarlo desde rviz.

5.5.4 Nodo principal 3 - Generación de los bounding boxes 3D

Por último, lo único que faltaría por hacer, sería el generar los bounding boxes 3D de los objetos a partir de la información de los bounding boxes 2D de YOLO, y de la información de la nube de puntos. Para ello, utilizamos el paquete Darknet Ros 3D¹, el cual teniendo la información comentada anteriormente, es capaz de generar dichos bounding boxes 3D. Por lo tanto, clonamos el repositorio y descargamos las dependencias necesarias. Acto seguido, modificamos la información necesaria del nodo, tanto de los archivos yaml, como de los ficheros launch, para que el nodo sea capaz de leer la información los topics requeridos, que en este caso son la nube de puntos, la imagen captada por la cámara del robot, y los bounding boxes 2D devueltos por YOLO.

Por ello, como bien hemos comentado, para crear los bounding boxes 3D, el programa parte de la nube de puntos sin segmentar, así como de la imagen_raw, y de los bounding boxes 2D devueltos por YOLO. Primero convierte la nube de puntos de tipo sensor_msgs::Pointcloud a pcl::pointcloud2, y genera una nube de puntos y un visualization_msgs::Marker por cada objeto detectado. Una vez hecho esto, itera sobre los puntos de cada una de las nubes, y se los asigna al punto3D correspondiente. Acto seguido, verifica si la proyección de dichos puntos se encuentra dentro de cualquier bounding box, y si es así, los asigna al grupo correspondiente. Después, se obtienen los subgrupos de cada nube de puntos del bounding box, y se obtiene el índice de clúster correspondiente al clúster más cercano , para agruparlos si fuera necesario. Por último, si hasta ahora todo ha funcionado, se orientan los bounding boxes respecto a la posición del robot, se crean los mensajes de tipo visualization_msgs::Marker con los bounding boxes 3D de cada uno de los objetos detectados, y se publican los bounding boxes en el topic /darknet_ros_3d_markers, como se puede comprobar en la Figura 5.5. Además, a la hora de publicar los bounding boxes, se tendrá en cuenta la clase detectada para cada uno de los objetos, y en caso de dicha clase no se encuentre dentro de una lista definida en el archivo de configuración yaml de 5.5, no se añadirá al array de markers. Una vez publicados los bounding boxes 3D, se podrá proceder a su visualización en tiempo real desde rviz, y aunque desde dicho simulador no se impriman las clases de los objetos, accediendo al topic donde se publican, se puede acceder a un campo dentro del cual se encuentra la clase de cada uno de los objetos.

Por último, comentar que la generación de los bounding boxes 3D se hace de manera dinámica, de modo que en cada instante de tiempo, únicamente se publica un marker array

¹https://github.com/IntelligentRoboticsLabs/gb_visual_detection_3d

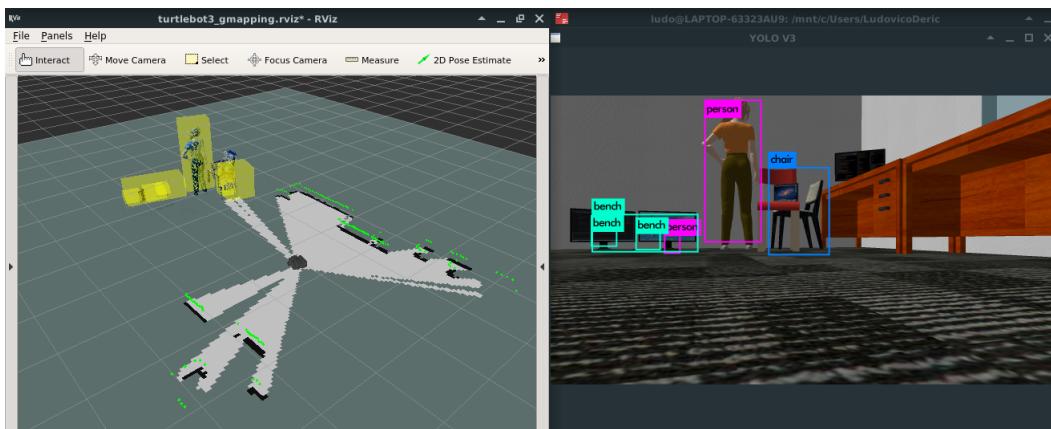


Figura 5.5: Generación de los bounding boxes 3D, funcionando de manera síncrona al resto de nodos.

con los bounding boxes de los objetos que están siendo detectados por YOLO, en lugar de almacenar en un array global todos los bounding boxes detectados a lo largo de la creación del mapa. El motivo por el que se ha implementado de esta manera el comportamiento, es con el fin de evitar que se vayan generando continuamente bounding boxes de los mismos objetos y se acaben solapando, dificultando por lo tanto la distinción de los mismos, y además, porque a la hora de que el robot realice un proceso más complejo, como podría ser la navegación automática, o la manipulación de alguno de los objetos de la escena, únicamente será necesario para el, el obtener los bounding boxes 3D de los objetos que está viendo, y que a su vez estos tengan la mayor precisión posible.

Una vez hecho todo lo anterior, se pueden ejecutar los nodos deseados, y proceder a la visualización en tiempo real tanto de la creación del mapa 2D de Gmapping, como a la del mapa 3D, o a los bounding boxes 3D de los objetos.

6 Experimentación

En este apartado vamos a enseñar el resultado final, así como las diferentes pruebas que se han ido realizando en el sistema, hasta obtener el resultado final:

6.1 Gmapping

En esta primera sección, vamos a realizar pruebas con el algoritmo de gmapping. Para ello probaremos a modificar los diferentes parámetros de configuración del algoritmo de gmapping, y mediante un rosbag de ROS comprobaremos cuando el funcionamiento del sistema es mejor.

Para probar los parámetros, se realizaron diferentes pruebas, modificando para ellos los parámetros de número de partículas, la actualización lineal, y la actualización angular. A continuación se muestran en la Figura 6.1, 3 de las pruebas realizadas, dentro de las cuales se puede ver el mapa ideal (color azul) comparándolo con el modelo obtenido por el robot (color rojo):

Pruebas	1	2	3
Particles	150	100	30
LinearUpdate	0,2	0,1	0,1
AngularUpdate	0,2	0,1	0,1

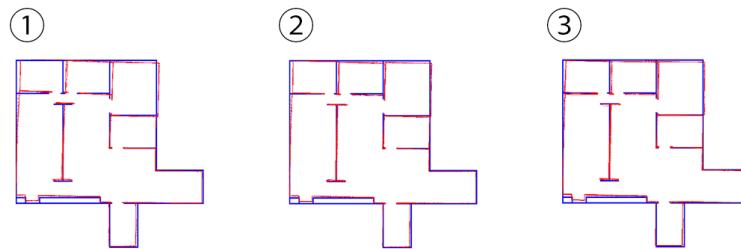


Figura 6.1: Distintas pruebas realizadas con la configuración de parámetros de gmapping

Tras realizar estas pruebas, se concluyó que los parámetros que mejor salida producían, eran los siguientes: `[_particleNumber, _linearUpdate, _angularUpdate] = [100, 0.1, 0.1]`

6.2 YOLO

Las pruebas realizadas sobre YOLO se han basado únicamente en la configuración del threshold de la probabilidad de detección de objetos. Mientras más bajo sea dicho parámetro, más detecciones se obtendrán, pero a su vez mayor probabilidad habrá de que dichas detecciones no sean correctas.

El threshold de detección de YOLO permite un rango de valores de 0 a 1, de manera que si se establece en 0, se devolverán todos las detecciones calculadas para la imagen actual, y conforme el valor se incrementa (siendo 0.25 el valor predeterminado determinado por YOLO), menor es el número de detecciones, pero a su vez, más precisas son.

Es por ello, que teniendo en cuenta que con el valor predeterminado de YOLO se seguían obteniendo detecciones imprecisas, se decidió aumentar el valor del threshold hasta un valor de 0.35, permitiendo así filtrar con algo más de precisión las detecciones incorrectas.

6.3 Procesamiento de las nubes de puntos

Para el procesamiento de la nube de puntos, se procedió a modificar los diferentes valores de configuración para la voxelización de la nube, y posteriormente para el filtrado de la misma mediante el algoritmo de RANSAC. Así pues, para la vocalización de la nube se probaron diferentes tamaños de hoja, con la intención de reducir el número de puntos de la nube hasta un valor adecuado para el procesamiento de la nube, y la posterior emisión de la misma a través de topics de ROS, de manera que no consumiera demasiado tiempo. Para ello, se comenzaron probando valores desde 0.1, y disminuyendo finalmente hasta alcanzar un valor de 0.015, el cual debido a la potencial del terminal donde se realizó el proyecto, se determinó como mejor valor posible. A continuación en la figura 6.2 diversas pruebas realizadas con la configuración del tamaño de hoja:

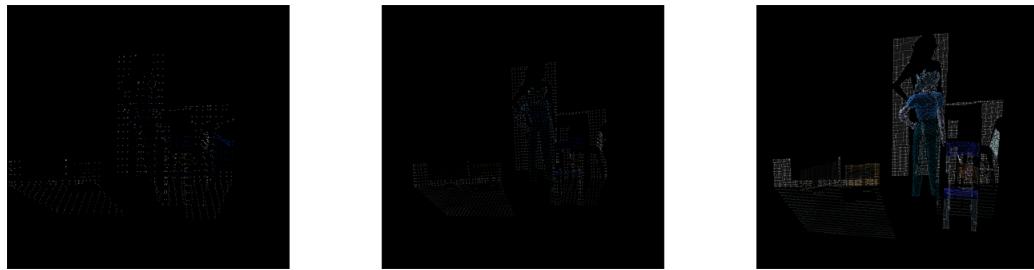


Figura 6.2: Pruebas del voxelizado con tamaño de hoja de 0.1 , 0.05 y 0.015 respectivamente

Acto seguido de realizar el voxelizado, se procedió a filtrar la nube para intentar eliminar los puntos de las paredes, así como del suelo. Para ello como bien se ha comentado anteriormente, usaremos el algoritmo de filtrado RANSAC. Para realizar dicho filtrado, se creó un objeto de segmentación, el cual se configuró con el método `pcl::SAC_RANSAC`, y un modelo de segmentación de tipo plano. Una vez hecho esto, y comprobado que funcionaba, se procedió a probar diferentes valores de threshold para pasarlo a la función que realiza el filtrado. Para ello, se comenzó a probar con valores desde 0.09, y se fue bajando hasta valores cercanos a 0.001. Finalmente tras realizar varias pruebas, se acabó eligiendo como valor adecuado más adecuado un threshold de 0.05, ya que permite eliminar los planos dominantes de la pared, así como del suelo, sin además eliminar demasiada geometría del resto de objetos de la nube. A continuación se muestran en la Figura 6.3 dos pruebas realizadas con la configuración de parámetros de RANSAC:

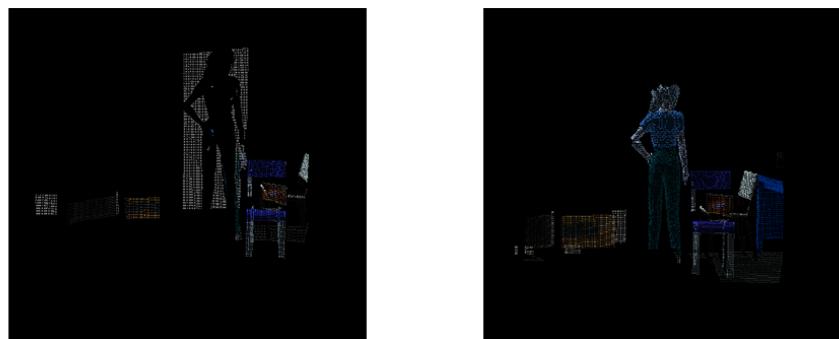


Figura 6.3: Pruebas de RANSAC con thresholds de 0.09 y 0.05 respectivamente

6.4 Creación del mapa 3D

Para la creación final del mapa 3D, el procedimiento seguido es el siguiente: una vez se detecta un cambio en el árbol de transformaciones de gmapping, esto se envía al primer fichero de Python, en el cual se almacena la nube de puntos, así como las coordenadas de los objetos en la imagen detectada por el turtlebot. Una vez hecho esto, la nube y las coordenadas del objeto se manda a otro fichero C++ donde se extraen las nubes de cada objeto, se les aplica un voxelizado y un filtrado mediante RANSAC, se les aplica una transformación para que se encuentren en el mismo frame que el mapa de gmapping, y una vez hecho esto, se combinan en una nube global.

Por ello, se intentaron probar diferentes métodos para reducir el error de la nube, ya que al realizar el mapeado de un entorno con objetos abundantes, o al realizar durante el mapeado giros o movimientos a altas velocidades, debido al retardo producido por la transmisión de los mensajes mediante ROS, como al tiempo de procesado de las nubes, como al error en la odometría del robot, el mapa final presenta solapamiento entre las nubes de puntos de los objetos, o algunos objetos se posicionan de manera incorrecta.

Para intentar reducir dicho error, se probaron diferentes mecánicas, siendo la primera de ellas, la de reducir las velocidades tanto lineales como angulares del robot a la hora de crear el mapa, lo cual arregla los errores de que los objetos se posicionen incorrectamente, además, también se probó a una vez juntados los objetos en la nube global, realizar un voxelizado para intentar combinar los puntos de los objetos duplicados, pero dicha opción se acabó descartando debido a que si el robot se encuentra estático, la nube de puntos iría reduciendo su número de puntos hasta que quedase prácticamente vacía, pero sería una opción viable, para una vez generado el mapa y haber sido guardado en memoria, aplicarle dicha transformación. Otra solución probada fue la de enviar los datos de las posiciones del robot con mayor frecuencia, de manera que cuando se detecten los objetos, su posición sea más exacta, pero no se acabó aplicando, dado que ralentizaba el resto de procesos, sin conseguir además mucha mejora.

Finalmente, se concluyó que debido al error producido entre la sincronización de las nubes de puntos con la posición del robot desde la cual se tomó las nubes, no se podía reducir completamente el error con dichos métodos, dado que el ángulo de rotación del robot que se asigna no tiene la precisión necesaria cuando este se mueve a una velocidad moderadamente rápida, pero por lo menos se consiguió reducir en cierta medida.

Así pues, se muestra a continuación en la Figura 6.4 la nube de puntos 3D tras haber realizado una vuelta al recinto, y se marca sobre el mismo en rojo las nubes de puntos mal posicionadas, debido a los giros bruscos que se realizaron durante la creación del mapa:



Figura 6.4: Mapeado 3D de la escena, con fallos en el posicionamiento de las nubes de los objetos debido a giros o movimientos a velocidades altas

En el siguiente apartado se mostrará el resultado final, incluyendo tanto el mapa 2D generado por Gmapping con los bounding boxes, como el un mapa 3D generado mientras el resto de procesos funcionaban simultáneamente.

6.5 Resultado final

Finalmente, se va a mostrar a continuación el resultado final del proyecto, realizando para ello el mapeado de el escenario mostrado en siguiente Figura 6.5

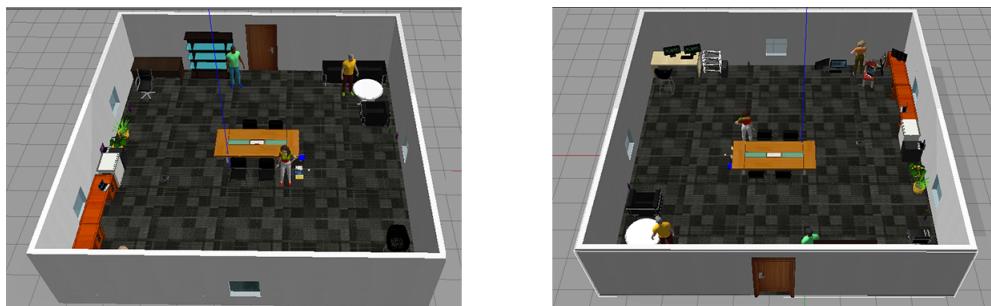


Figura 6.5: Escenario con diversos objetos, para comprobar el funcionamiento del sistema

Para realizar dicha prueba de mapeado, se recorrerá el escenario una única vez, a una velocidad baja/media, de manera que se pueda comprobar cómo afecta dicha velocidad a la hora de la creación del mapa. Para ello, se ejecutarán a la vez los diferentes nodos que se han

comentado anteriormente en el apartado del funcionamiento de ROS. Así pues, procedemos a mostrar el resultado de cada uno de los mapas individualmente (2D y 3D), y acto seguido, mostraremos como se ve la información condensada en un mismo mapa.

Primero, procederemos a mostrar varios instantes, donde se vea el mapa 2D y 3D juntos durante la creación de los mismos, así como las detecciones realizadas por YOLO, y los consiguientes bounding boxes 3D detectados en sus determinados instantes de tiempo. Dicho resultado se puede ver en la Figura 6.6.

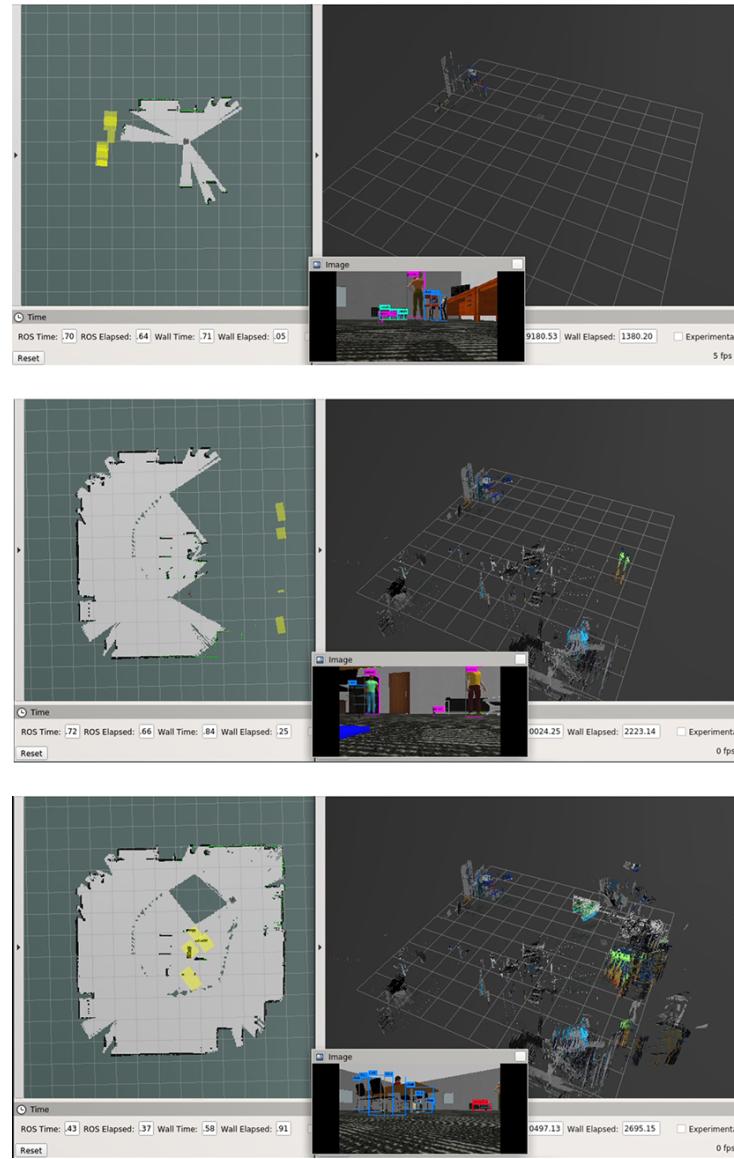


Figura 6.6: Mapas 2D y 3D junto a los bounding boxes detectados en instantes determinados de tiempo.

A continuación, se procederá a mostrar los mapa 2D y 3D generados tras realizar una

vuelta completa al escenario. Para ello, en la Figura 6.7 se puede ver el resultado del mapa 2D generado por Gmapping, mientras que en la Figura 6.8 se puede ver el mapa 3D generado mediante las nubes de puntos de los objetos segmentados.

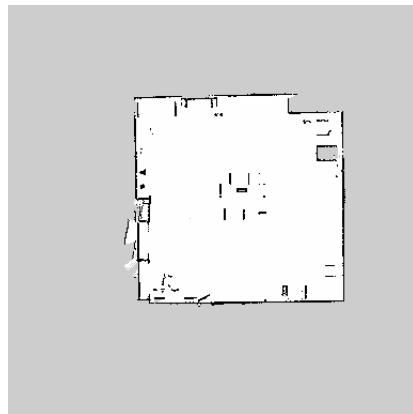


Figura 6.7: Mapa de ocupación 2d generado por Gmapping

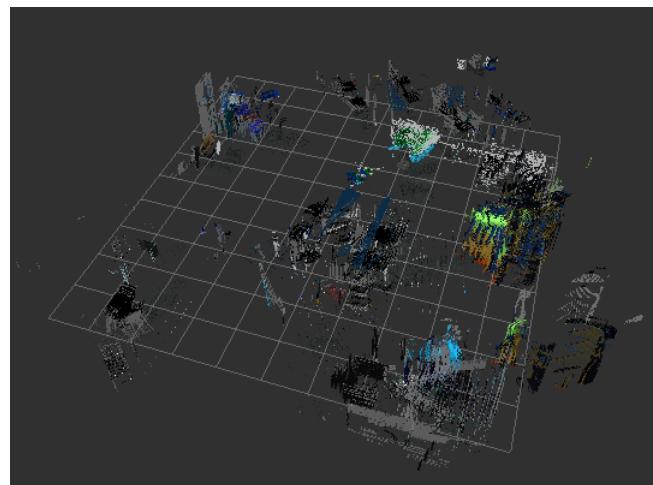


Figura 6.8: Mapa 3D generado tras dar una vuelta al recinto con velocidades bajas/medianas

Por último, procedemos a mostrar en la Figura 6.9 un mapa conjunto tanto por la información devuelta por Gmapping, como por el mapa 3D generado a partir de los objetos, a modo de comprobar que las transformaciones de los objetos se realizan correctamente.

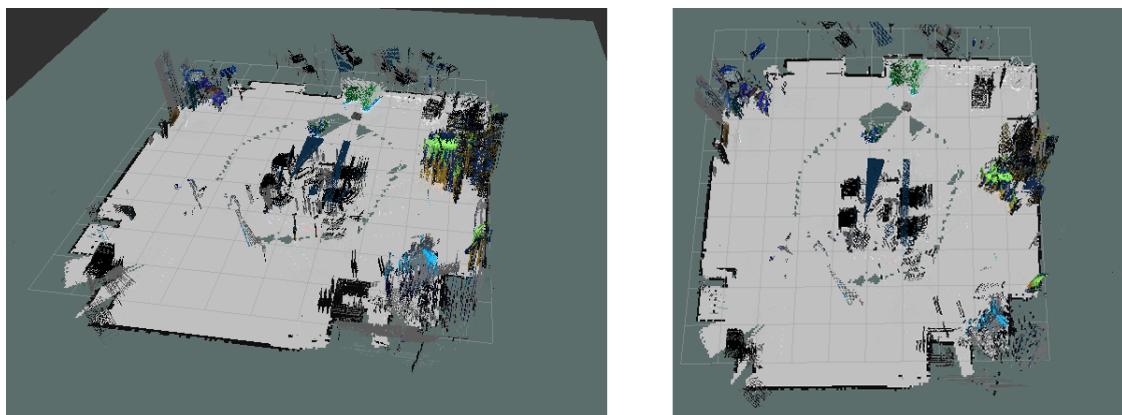


Figura 6.9: Mapa generado combinando la información 2D y 3D, visto desde perspectivas diferentes

7 Conclusiones

Finalmente, como conclusión, podemos decir que hemos logrado los objetivos propuestos, ya que hemos logrado implantar un sistema de generación de mapas 2D, con la introducción de datos de los objetos de la escena, detectados previamente mediante una IA de detección de objetos.

Además, se han llevado a cabo diferentes pruebas para comprobar cuándo se obtiene mejor respuesta por parte del sistema. En cuanto a las motivaciones para realizar este proyecto, gracias a él, se han podido desarrollar conocimientos y habilidades de las cuales, antes no se tenían, o se han mejorado los conocimientos y habilidades actuales. Esto es gracias a que los programas realizados y modificados eran tanto de Python como de C++, por lo que hemos podido conseguir lo que mencionamos, ya que antes de realizar este proyecto, el nivel que se tenía de Python era bajo. Respecto a la implementación de una inteligencia artificial, se acabó optando por implementar un sistema ya funcional, aunque gracias a la modificación del datasheet, y los pesos de las diferentes capas de la red neuronal utilizada, aunque no se construyó una red neuronal desde cero, se amplió el conocimiento de las mismas.

Por último, el aspecto que más fuerza ha cobrado en este proyecto ha sido ROS, ya que se profundizó ampliamente en su empleo para comunicar los diferentes sistemas involucrados en este proyecto, utilizando para ello conocimientos que antes no se tenían. Por lo que, finalmente, podemos decir que el propósito del proyecto, que reúne los objetivos propuestos y las motivaciones del trabajo, se ha llevado a cabo con éxito.

8 Trabajos Futuros

Por último, vamos a exponer una serie de trabajos que se podrían llevar a cabo en un futuro a modo de ampliación. Como ya se ha mencionado, debido al confinamiento, se han tenido que realizar varios cambios, por lo que, en primer lugar y como un trabajo de futuro, sería poder implementar en el sistema real con el Pepper. Además, otra posible mejora, sería el de incluir todos los sistemas que utilizan mensajes o topics de ROS para transmitir información, en un Nodelet, de manera que la obtención de los datos deseados se haga de manera más rápida, mejorando así los resultados de los diferentes mapas, y reduciendo en cierta medida los fallos debidos al retardo ocasionado por las comunicaciones de dichos datos, haciendo así el sistema más invariable respecto a la velocidad del robot. Por último, se podría mejorar el comportamiento de los bounding boxes 3D, de manera que en lugar de representarse únicamente aquellos detectados en cada instante de tiempo, estos se guardaran en un array global, el cual más adelante permitiera la visualización de todos los bounding boxes 3D de la escena en conjunto.

Bibliografía

- Bjelonic, M. (2016–2018). *YOLO ROS: Real-time object detection for ROS*. https://github.com/leggedrobotics/darknet_ros.
- Grisetti, G., Stachniss, C., y Burgard, W. (2007). Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE Transactions on Robotics*, 23(1). Descargado de <https://ieeexplore.ieee.org/abstract/document/4084563>
- Gupta, S., Arbelaez, P., y Malik, J. (2013). Perceptual organization and recognition of indoor scenes from rgb-d images. *CVPR 2013*. Descargado de https://www.cv-foundation.org/openaccess/content_cvpr_2013/html/Gupta_Perceptual_Organization_and_2013_CVPR_paper.html
- Gupta, S., Girshick, R., Arbeláez, P., y Malik, J. (2014). Learning rich features from rgb-d images for object detection and segmentation. *European Conference on Computer Vision*. Descargado de https://link.springer.com/chapter/10.1007/978-3-319-10584-0_23
- Hokuyo urg-04lx datasheet. (s.f.). https://www.hokuyo-aut.jp/dl/Specifications_URG-04LX_1513063395.pdf.
- Joseph, R., y Ali, F. (2018). Yolov3: An incremental improvement. *arXiv*.
- Lin, D., Fidler, S., y Urtasun, R. (2013). Holistic scene understanding for 3d object detection with rgbd cameras. *ICCV 2013*. Descargado de https://openaccess.thecvf.com/content_iccv_2013/html/Lin_Holistic_Scene_Understanding_2013_ICCV_paper.html
- Martín, F., y González, F. (2020). *Darknet ROS 3D: Real-time object 3d bounding boxes generation for ROS*. https://github.com/IntelligentRoboticsLabs/gb_visual_detection_3d.
- Mur-Artal, R., y Tardós, J. D. (2016). Orb-slam2: An open-source slam system for monocular, stereo, and rgbd cameras. *IEEE Transactions on Robotics*, 33(5). Descargado de <https://ieeexplore.ieee.org/abstract/document/7946260>
- Redmon, J., Divvala, S., Girshick, R., y Farhadi, A. (2016). You only look once: Unified, real-time object detection. *CVPR 2016*, 1(1), 1-10. Descargado de https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf
- Ren, X., Bo, L., y Fox, D. (2012). Rgb-(d) scene labeling: Features and algorithms. *IEEE Conference on Computer Vision and Pattern Recognition*. Descargado de <https://ieeexplore.ieee.org/abstract/document/6247999>

- Ren, Z., y Sudderth, E. B. (2016). Three-dimensional object detection and layout prediction using clouds of oriented gradients. *CVPR 2016*. Descargado de https://www.cv-foundation.org/openaccess/content_cvpr_2016/html/Ren_Three-Dimensional_Object_Detection_CVPR_2016_paper.html
- ROBOTIS. (2021). *Turtlebot3 especifications*. <https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications>.
- Song, S., y Xiao, J. (2016). Deep sliding shapes for amodal 3d object detection in rgb-d images. *CVPR 2016*. Descargado de https://openaccess.thecvf.com/content_cvpr_2016/html/Song_Deep_Sliding_Shapes_CVPR_2016_paper.html
- Song, S., y Xiao, J. (2017). 2d-driven 3d object detection in rgb-d images. *ICCV 2017*. Descargado de https://openaccess.thecvf.com/content_iccv_2017/html/Lahoud_2D-Driven_3D_Object_ICCV_2017_paper.html
- Whelan, T., Salas-Moreno, R. F., Glocker, B., Davison, A. J., y Leutenegger, S. (2016). Elasticfusion: Real-time dense slam and light source estimation. *Sage Journals*. Descargado de <https://ieeexplore.ieee.org/abstract/document/7946260>
- Xu, W., y Lee, E.-J. (2012). Human-computer natural user interface based on hand motion detection and tracking. Descargado de https://www.researchgate.net/publication/263637885_Human-Computer_Natural_User_Inter_face_Based_on_Hand_Motion_Detection_and_Tracking