TRABAJO FIN DE GRADO

Percepción Visual para Robots Mediante el Uso de Deep Learning y Cámaras RGBD

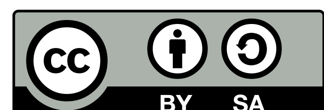
Realizado por
Fernando González Ramos

Para la obtención del título de
Grado en Ingeniería Telemática

Dirigido por
Dr. Francisco Martín Rico

Convocatoria curso 2023/24

Esta obra está bajo una licencia Creative Commons «Reconocimiento-CompartirIgual 4.0 Internacional».



Resumen

Para el Ser Humano, el sentido de la vista resulta primordial para así ser capaces de reconocer nuestro entorno y, de ese modo, poder interactuar sobre él y tomar decisiones. Del mismo modo, si deseamos que un robot sea capaz de desplegar un conjunto de comportamientos complejos basados en la interacción con su entorno, es absolutamente necesario que éste sea dotado de dicho sentido de la vista.

Los sensores más sofisticados creados hasta la fecha y, mediante los cuales, podemos tratar de imitar la visión humana, son las cámaras.

Sin embargo, su utilización no resulta, en ningún caso, trivial, pues el desbordante flujo de datos que las cámaras proporcionan convierte en una tarea compleja su procesamiento. Es por este motivo que, en muchas ocasiones, se requiere de la utilización de un potente hardware para poder realizar múltiples tareas de cálculo a la mayor brevedad de tiempo posible y, además, de un software altamente optimizado.

A lo largo de este trabajo se expone una herramienta creada para utilizar la información que las cámaras proporcionan, como la imagen y la profundidad, con la finalidad de calcular la localización espacial de cualquier objeto, así como sus dimensiones. La generación de estos datos por parte de la herramienta resulta de gran utilidad para la creación de algoritmos robóticos complejos.

Se trata de una herramienta basada en software y diseñada específicamente para su integración con el framework ROS (Robot Operating System) y cámaras denominadas *RGBD*, capaces de proporcionar información de profundidad.

El framework ROS, explicado en mayor detalle en capítulos posteriores, se compone de un conjunto de librerías software y herramientas que se integran por completo en el Sistema Operativo para permitir la elaboración de algoritmos robóticos complejos de la forma más sencilla posible. A día de hoy, representa el estándar en robótica de investigación, abarcando también el amplio espectro de la robótica industrial.

La herramienta que se expone en el transcurso del presente trabajo de investigación trata de resolver un problema fundamental relacionada con la capacidad de percepción visual en un robot. Para que un robot pueda relacionarse con su entorno (manipular objetos, navegar libremente entre posibles obstáculos, etc), al igual que las personas, es necesario que sean capaces de detectar todos aquellos objetos que los rodean y conocer su posición exacta dentro del entorno, así como sus dimensiones.

Este trabajo se centra, por tanto, en el desarrollo de una herramienta software para robots que permite la detección de objetos y su posicionamiento en el espacio, así como el cálculo de sus dimensiones (alto, ancho y profundidad).

Abstract

For humans, the sight is essential to recognize our environment and then to be able to interact with it as well as take decissions. Hence, if we want that a robot may be capable to deploy some complex behaviors based on the interaction with its environment, it is absolutely necessary for the robot to have the sense of sight.

The most sofisticated sensors at the day of today are the cameras. Thanks to them, the human sight can be emulated.

However, using cameras is not easy in any case. The cameras provide a great flow of data that must be processed. For this reason, many times very powerful hardware is required that allow to carry out multiple arithmetic tasks as quick as possible. In addition, a highly optimized software is required.

Through this project, it is exposed and explained one tool which has been created to use that information the cameras provide, like the image and the depth, with the objective of calculating the spatial localization of any object as well as their shape. The generation of this data by the tool is useful for the creation of complex robotic algorithms.

It is about a software-based tool and specifically designed for its integration with the ROS (Robot Operating System) framework and some kind of cameras named RGBD that are capable to provide depth information.

The ROS framework, explained deeply in the following sections, is composed by a collection of software libraries and tools fully integrated in the Operating System to allow the creation of complex robotic algorithms in a simple manner. Nowadays, ROS represents the standard in robotics researching, taking part in the industrial robotics too.

The tool explained along this researching work tries to solve a fundamental problem related to the visual perception capacity for robots. For a robot to be able to interact with its environment (manipulate objects, navigate avoiding obstacles, etc), like humans, they must be able to detect those objects around and know its position within the environment, as well as its shape.

Thus, this work is focused in the development of a software tool for robots that allow to detect objects, get their spatial position and their shape (height, weight and depth).

Índice general

1. Introducción **1**
1.1. ¿Qué es un Robot? 1
1.2. Componentes de un Robot 3
1.2.1. Sensores 3
1.2.2. Actuadores 5
1.2.3. Unidad de Procesamiento 8
1.2.4. Arquitecturas Software 9
1.3. Capacidades de un Robot 10
1.3.1. Visión 11
1.3.2. Navegación 12
1.3.3. Diálogo 13
1.3.4. Manipulación 14
1.4. ¿Qué es ROS? 15
1.5. Estado del arte 16
1.5.1. Redes Neuronales 17
1.5.2. Limitaciones 19

2. Requisitos y Metodología **21**
2.1. Sistemas Embebidos o Empotrados 21
2.2. Arquitectura Básica de un Computador 21
2.2.1. Procesador Monociclo 22
2.2.2. Procesador Multiciclo 23
2.2.3. Procesador Segmentado 23
2.3. Sistemas Operativos 25
2.3.1. Programas y Procesos 26
2.3.2. Planificación de Procesos 26
2.4. Sistemas Distribuidos 27
2.4.1. Recursos Accesibles 27
2.4.2. Estandarización 28
2.4.3. Escalabilidad 28
2.5. Metodología 28

3. Entorno y Herramientas **31**
3.1. ROS 31
3.1.1. Introducción 31
3.1.2. Comunidad 31
3.1.3. Grafo de Computación 32
3.1.4. Depuración y Monitorización 36
3.1.5. Estandarización 38
3.1.6. Sistema de Ficheros 39
3.2. YOLO 39
3.2.1. Introducción 39

3.2.2. Entrenamiento 39
3.2.3. Algoritmo 40
3.3. YOLACT 41
3.3.1. Etiquetado 42

4. Desarrollo del Trabajo 44
4.1. Preámbulo 44
4.2. Darknet ROS 3D 44
4.2.1. Introducción 44
4.2.2. Estructura del Sistema 45
4.2.3. Estructura del Software 50
4.2.4. Algoritmia 54

5. Evaluación y Despliegue 61
5.1. Plataformas Robóticas 61
5.2. Plataformas de Computación 63
5.2.1. Computadores de Escritorio 63
5.2.2. Computadores Embebidos 63
5.3. Entorno 65
5.4. Repercusión 65

6. Trabajos Futuros 67
6.1. Mapeo Semántico 67
6.2. Estimación de Trayectorias 67

A. Bibliografía 69

Índice de figuras

1.1. Ejemplos de Robots Cotidianos . . . 2
1.2. Sensor de Ultrasonidos HC-SR04 . . . 4
1.3. Funcionamiento sensor de ultrasonidos . . . 4
1.4. Sensor LM35 . . . 5
1.5. Motor de Corriente Continua . . . 6
1.6. Tren de Pulsos . . . 7
1.7. Brazo Robótico . . . 8
1.8. Microcontrolador vs Microprocesador . . . 9
1.9. Cámara RGBD *ASUS Xtion* . . . 12
1.10. Occupancy Grid Map . . . 13
1.11. Brazo Robótico Industrial . . . 14
1.12. Robot PR1 . . . 15
1.13. Ejemplo Detección Red Neuronal . . . 16
1.14. Esquema básico de una Red Neuronal . . . 17
1.15. Funcionamiento Neurona Artificial . . . 18
1.16. Función de Activación Sigmoide . . . 19
1.17. Morfología de una Imagen Digital . . . 20

2.1. Señal de Reloj . . . 23
2.2. Ejemplo Pipelining . . . 24
2.3. Estructura del Sistema . . . 25
2.4. Máquina de Estados de Proceso . . . 27
2.5. Modelo de Desarrollo en Cascada . . . 29

3.1. Grafo de Computación Básico . . . 34
3.2. Ejemplo Comandos *rosnode* y *rostopic* . . . 36
3.3. Ejemplo Visualización de Imágenes con RViz . . . 37
3.4. Ejemplo del comando *rosmsg* . . . 38
3.5. Ejemplo de Herramienta de Etiquetado: *Label Img* . . . 40
3.6. Yolact: Ejemplo de Detección . . . 41
3.7. Ejemplo de Etiquetado con *Labelme* . . . 42
3.8. Detección Yolact . . . 43

4.1. Arquitectura Software de Darknet ROS 3D . . . 45
4.2. Grafo Computacional de Darknet ROS 3D . . . 46
4.3. Tipos de Mensaje . . . 49
4.4. Bounding Boxes 3d Messages . . . 50
4.5. Estructura de Directorios . . . 51
4.6. Diagrama de Componentes . . . 52
4.7. Secuencia Ejecución de Callbacks . . . 54
4.8. Diagrama de Flujo Método Update . . . 57
4.9. Visual Markers . . . 60

5.1. Plataformas Robóticas Utilizadas . . . 62

5.2. Arquitectura Software sobre Plataforma Robótica 62
5.3. Nvidia Jetson Nano 64
5.4. Nvidia Jetson Tx2 64
5.5. Repositorio Público de GitHub 66

6.1. Ejemplo de uso Filtro Gaussiano 68

Índice de extractos de código

3.1. Ejemplo Sistema Reactivo 34
4.1. Algoritmo Iterativo del nodo Darknet 3D 55
4.2. Callback del Point Cloud 56
4.3. Callback de los Bounding Boxes 56
4.4. Transformada del Point Cloud 58
4.5. Point Cloud Decodificado 59

1. Introducción

A continuación, se presenta el trabajo relacionado con el desarrollo del software denominado **Gb Visual Detection 3D**. Se trata de una herramienta software destinada a robótica y específicamente diseñada para su utilización en el campo de la robótica de investigación. Fue desarrollada como parte del trabajo de investigación llevado a cabo en el grupo de robótica *Gentlebots* de la Universidad Rey Juan Carlos, lo que dio nombre a la herramienta con el prefijo *Gb*.

Esta herramienta, hace uso de una cámara RGBD, capaz de proporcionar datos de profundidad de la imagen, además de color; y de una red neuronal denominada YOLO (You Only Look Once) capaz de detectar objetos concretos contenidos en una imagen.

Gb Visual Detection 3D es capaz de recolectar toda esta información para calcular los datos acerca de las dimensiones tridimensionales de cada uno de los objetos de interés, y su posición en coordenadas espaciales.

Esta información que proporciona la herramienta (posición y dimensiones espaciales) no se procesa de ningún modo ni se utiliza para ninguna clase de algoritmo específico ya que Gb Visual Detection 3D es un componente software pensado para su integración con cualquier otro componente software que desee hacer uso en sus propios algoritmos de esta información que proporciona.

Sin embargo, en capítulos finales de este trabajo se expone de qué manera ya ha sido integrada la herramienta como parte de un sistema robótico completo, haciendo uso de ella en entornos de operación reales.

Con el fin de facilitar dichas labores de integración, se ha decidido utilizar el framework **ROS**, explicado con detalle en posteriores capítulos.

Además, se trata de un proyecto de Open Source (Software Libre), lo que significa que cualquier persona tiene acceso al código fuente del software, puede utilizarlo en sus propios proyectos, modificarlo e incluso contribuir a su mantenimiento o mejoría. De esta forma, se contribuye a realizar software robusto y estándar para su uso en robótica.

1.1. ¿Qué es un Robot?

La definición que proporciona la Real Academia Española acerca del término *robot* es la siguiente: *Máquina o ingenio electrónico programable que es capaz de manipular objetos y realizar diversas operaciones.*

Esta definición, desde un punto de vista de un ingeniero, resulta demasiado restrictiva.

Podemos considerar un robot a aquel artefacto dotado de **autonomía** para llevar a cabo tareas de mayor o menor complejidad. En la mayoría de los casos lo que se

pretende es imitar el comportamiento humano.

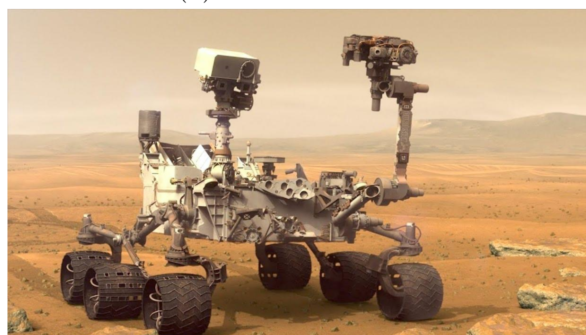
Frecuentemente se hace mención a la **inteligencia** de un robot como forma de medir la complejidad de las tareas que es capaz de realizar. Sin embargo, desde un punto de vista filosófico, no podemos considerar que un robot posea inteligencia, ya que ésta sólo se le atribuye a los humanos. En cualquier caso, se trata de un concepto que genera cierta controversia.

En el ámbito de la ciencia ficción existen infinidad de robots que podemos considerar inteligentes debido a que se comportan como los humanos. Este es el caso del mítico robot *C3PO*. No obstante, a día de hoy el ser humano se encuentra muy lejos de ser capaz de desarrollar este tipo de robots.

En el área de la investigación, lo que se pretende es generar **comportamientos autónomos** en los robots para que éstos sean capaces de llevar a cabo tareas concretas. Por este motivo, convivimos rodeados de robots que realizan sus funciones en áreas muy diversas. A continuación se mencionan algunos ejemplos, ilustrados también en la figura 1.1.

- Industria: Brazos robóticos para las cadenas de producción.
- Coches Autónomos: Coches que aparcan o incluso conducen de manera autónoma.
- Investigación Espacial: Robots *Curiosity* o *Perseverance*.
- Medicina: Robot *DaVinci*.
- Domésticos: Robot *Roomba*.
- Militares: Robots para desactivar explosivos.

(a) Robot Da Vinci



(b) Robot Curiosity

Figura 1.1: Ejemplos de Robots Cotidianos

Por supuesto, no todos estos robots poseen el mismo grado de autonomía. Algunos son teleoperados y, otros, incluso actúan de forma cooperativa formando grupos de robots.

1.2. Componentes de un Robot

Cualquier tipo de robot se compone, en definitiva, de dos partes. Por un lado, el hardware y por otro, el software. El hardware comprende la parte física del robot. Es decir, aquellas partes mecánicas y los componentes electrónicos.

Dos de los elementos hardware principales que podemos encontrar en un robot son los **sensores** y los **actuadores**.

Por su parte, el software reside en los programas. Éstos dotan al robot de dicha inteligencia previamente mencionada. Se encargan de procesar los datos obtenidos por los sensores y elaborar respuestas sin intervención externa. Esta respuesta se lleva a cabo por medio de los actuadores.

Por tanto, el software tiene como finalidad coordinar los diferentes elementos hardware del robot.

Hoy en día, la humanidad posee gran tecnología hardware que permitiría llevar a cabo aplicaciones de gran complejidad del mismo modo que las haría una persona. Entonces, ¿qué impide crear un robot similar a los de la Ciencia Ficción?

La principal limitación no se encuentra en el hardware, sino en el software. Concretamente, en la forma en la que éste ha de organizarse para poder procesar una gran cantidad de estímulos bajo diversas condiciones. Esto da lugar al campo de estudio de las **arquitecturas software** del que hablaré unos capítulos más adelante.

1.2.1. Sensores

Los sensores son aquellos componentes hardware que se encargan de medir diferentes magnitudes físicas del entorno del robot. Por ejemplo distancias, temperatura, luz, etc.

Son vitales pues todo el conocimiento que el robot es capaz de poseer del entorno que lo rodea se obtiene a través de los sensores. Sin ellos, el robot no sería capaz de elaborar ningún tipo de tarea, del mismo modo que no podría hacerlo un humano desprovisto de vista, oído y demás sentidos.

Existen diversos tipos de sensores pero, todos ellos, se pueden clasificar en dos clases: **activos** y **pasivos**.

Los sensores activos son aquellos cuyo funcionamiento se basa en la producción de un estímulo físico en el medio. Posteriormente, analizan la interacción de dicho estímulo con el entorno y, a partir de ello, se obtienen datos.

El sensor de ultrasonidos, ilustrado en la figura 1.2 es un gran ejemplo de este tipo de sensores.

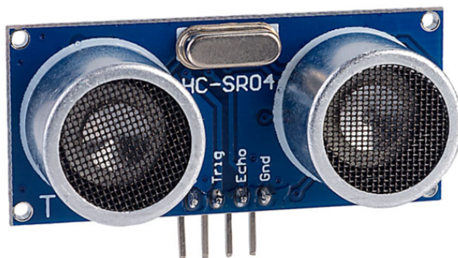


Figura 1.2: Sensor de Ultrasonidos HC-SR04

Este sensor, denominado *HC-SR04* es ampliamente utilizado en robótica. Se utiliza para medir distancias. El principio de su funcionamiento es el siguiente:

Posee un transmisor de ultrasonidos, así como un receptor. Por tanto, su modo de operación es emitir un pulso, es decir, una señal de ultrasonidos de duración limitada (del orden de unos pocos de μs). Posteriormente, se espera a recibir el *echo*.

Cuando la señal emitida impacta en algún objeto, se produce una refracción de la señal que, al cabo de un tiempo, es captada por el receptor, tal y como se ilustra en la figura 1.3. Esta señal es la que se denomina *echo*.

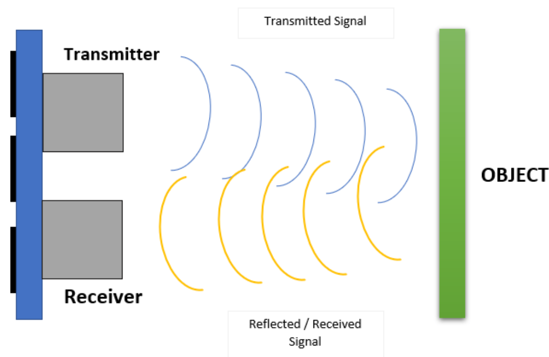


Figura 1.3: Funcionamiento sensor de ultrasonidos

Sabiendo que la velocidad del sonido en el aire es de $340m/s$, podemos conocer la distancia al objeto de la siguiente forma:

$$v = \frac{\Delta X}{t}$$

Siendo ΔX el espacio y t el tiempo.

Si sustituimos el valor de la velocidad del sonido y despejamos el espacio, obtenemos la siguiente expresión:

$$\Delta X = 340 \cdot t$$

El tiempo es un parámetro conocido pues es el tiempo que se tarda en recibir el *echo* desde que se emitió la señal. De este modo, habríamos calculado el valor de ΔX. No obstante, este dato arroja el valor de la distancia total recorrida por la señal (ida y vuelta). Dado que sólo nos interesa conocer la distancia al objeto, debemos dividirlo entre 2. Por tanto, la expresión matemática queda de la siguiente manera:

$$d = \frac{340 \cdot t}{2} \tag{1.1}$$

De este modo, podemos conocer la distancia de cualquier objeto con respecto al sensor.

Tal y como se ha explicado, el sensor HC-SR04 se trata de un sensor *activo* pues produce un estímulo físico en el entorno (onda de ultrasonidos) para posteriormente estudiar el estímulo de respuesta. En este caso, la respuesta resulta ser el tiempo que tarda en recibirse la señal reflejada.

Resulta de vital importancia recalcar que en múltiples casos, los sensores no proporcionan explícitamente la información que necesitamos sino que generan datos que necesitan ser **procesados** para, a partir de ellos, obtener la información de interés. El lugar donde todos estos datos son procesados supone un elemento crucial para el funcionamiento de un robot. En el capítulo 1.2.3 se explicará con mayor nivel de detalle.

Por su parte, los sensores pasivos, simplemente miden magnitudes del entorno sin producir ningún estímulo sobre él. Un ejemplo de este tipo de sensores es el sensor **LM35**, representado en la figura 1.4. Es ampliamente conocido como uno de los sensores de temperatura más comunes. Se trata de un sensor capaz de medir temperaturas en un rango que varía desde los -55ºC a los 150ºC, lo cual lo hace ideal para medir temperaturas populares.

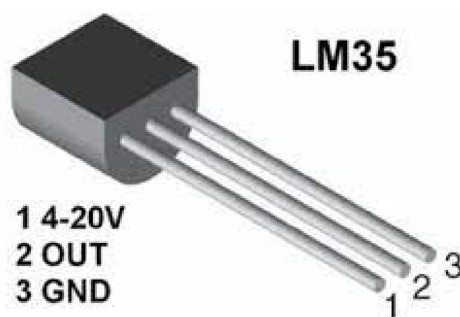


Figura 1.4: Sensor LM35

1.2.2. Actuadores

Los actuadores son todos aquellos componentes que permiten al robot interactuar con todo aquello que le rodea. Gracias a ellos el robot puede llevar a cabo acciones que

supongan un cambio en su entorno o, incluso, en el propio robot.

Para ejemplificar esta definición podemos considerar dos tipos de actuadores. El primero consta de un brazo provisto de una pinza mediante la cual es capaz de agarrar objetos, tal y como se muestra en la figura 1.7.

En este caso, el brazo consta de diferentes partes móviles unidas entre sí mediante **servomotores**. Cabe hacer una mención especial a este tipo de elementos hardware.

Motores y Servomotores Eléctricos

Un motor es aquello que produce movimiento. En robótica se poseen diferentes tipos de motores: hidráulicos, neumáticos, eléctricos, etc; dependiendo de su modo de funcionamiento.

En esta sección nos centraremos en los motores eléctricos.

El principio físico del funcionamiento de los motores eléctricos se basa en la interacción de campos eléctricos y magnéticos, los cuales producen movimiento. Esto se consigue introduciendo una corriente eléctrica a través de una bobina. Una gran variedad de robots utilizan este tipo de motores alimentados con corriente continua, tal como se ilustra en la siguiente imagen.

Figura 1.5: Motor de Corriente Continua

Por su parte, un servomotor se trata de un tipo de motor que puede ser controlado mediante una señal eléctrica permitiendo de ese modo realizar movimientos rotatorios hacia posiciones determinadas. Es decir, este tipo de motores poseen dos entradas para corriente continua (voltaje y tierra) y otra por la que recibirá una señal denominada **PWM**.

PWM (Pulse Width Modulation -modulación por ancho de pulso-) es un tipo de modulación digital. Las señales digitales moduladas por PWM se comportan como un tren de pulsos de diferentes anchuras. Es en las anchuras de estos pulsos donde se codifica la información que transmite la señal.

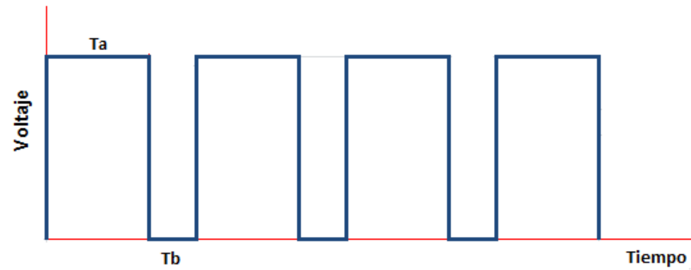


Figura 1.6: Tren de Pulsos

En la figura 1.6 podemos observar una señal digital formada por una secuencia de pulsos (escalones), haciendo así que la señal varíe en el tiempo con valores de voltaje altos y bajos.

T_a denota el periodo de tiempo que la señal se encuentra en nivel alto, mientras que T_b hace referencia al periodo en el que se encuentra a nivel bajo. Variando estos anchos de pulso podemos indicar al servomotor hacia qué sentido debe girar, a qué velocidad debe moverse o e qué posición debe colocarse.

Los brazos robóticos suelen estar formados por diversos servomotores de modo que tienen gran movilidad. Mediante este actuador, el robot sería capaz de mover objetos, produciendo de esta forma cambios en su entorno. Este tipo de brazos, dan lugar al campo de estudio de la robótica denominado **manipulación**. La manipulación de objetos aborda el clásico problema de ser capaz de mover un objeto desde su posición inicial a una posición de destino.

Teniendo en cuenta las propiedades físicas del objeto (peso, dimensiones, dureza, orientación, etc), así como las del brazo robot; y teniendo en cuenta las restricciones del entorno (posibles obstáculos que el brazo ha de evitar) se debe mover de lugar el objeto. Esta acción resulta de gran simpleza para un humano ya que niños de apenas dos años son capaces de realizarla. Sin embargo, entraña gran complejidad desde el punto de vista matemático (modelización mediante ecuaciones de las físicas del objeto, brazo y entorno) así como de programación software.

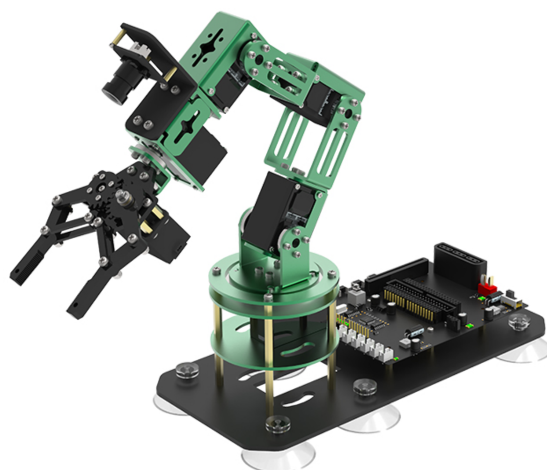


Figura 1.7: Brazo Robótico

El segundo tipo de actuadores que podemos considerar son las ruedas. Las ruedas son elementos indispensables para multitud de robots hoy en día.

Del mismo modo que los brazos robóticos, éstas son accionadas por medio de motores o servomotores.

Ambos tipos de actuadores dan lugar a dos clases de robots ampliamente utilizados: Los denominados *robots manipuladores* y los *robots móviles*. Los primeros suelen ser utilizados en entornos **estáticos** y suelen desempeñar tareas más repetitivas. En cambio, los robots móviles, habitualmente deben ser capaces de lidiar con entornos dinámicos, es decir, muy cambiantes y, además, suelen desempeñar tareas sensiblemente más complejas que los anteriores.

1.2.3. Unidad de Procesamiento

Como se ha mencionado en capítulos previos, todos aquellos datos provenientes de los sensores (entradas) han de ser procesados en algún lugar para elaborar respuestas que deberán ser llevadas a cabo por los actuadores (salidas). Es en este lugar, por tanto, donde reside la **inteligencia**[1] del robot.

La unidad de procesamiento es, por tanto, un elemento indispensable en un robot y puede estar definida por dos elementos: **microcontroladores** y **microprocesadores**.

Un microcontrolador se trata de un circuito integrado que dispone de puertos de entrada/salida normalmente digitales (aunque también pueden ser analógicos); memoria RAM y unidad aritmético-lógica (ALU), que es donde se realizan las operaciones matemáticas (habitualmente sumas y restas).

[1]Téngase en cuenta el contexto en el que se usa el concepto *inteligencia*, como sinónimo de *autonomía* para realizar tareas.

Por otro lado, un microprocesador está compuesto eminentemente por la unidad aritmético-lógica. Se trata de un elemento hardware especialmente diseñado y enfocado en la realización de cálculos matemáticos. Por tanto, su velocidad de cálculo es notablemente superior a la de los microcontroladores. No obstante, los microprocesadores no disponen de entradas y salidas sino que requieren de periféricos adicionales, memoria RAM y buses de entrada/salida.

(a) Microcontrolador

(b) Microprocesador

Figura 1.8: Microcontrolador vs Microprocesador

En la figura 1.8 se pueden apreciar un microcontrolador, concretamente un PIC (*Circuito Integrado Programable*), y un procesador Intel Xeon 5600.

Es en el microcontrolador o el microprocesador donde el software del robot se ejecuta para procesar los datos de entrada y generar salidas a los actuadores. Parece razonable pensar que cuanto más compleja sea la tarea que el robot debe desempeñar, más complejo será el software.

Es en esta última cuestión donde la estructura del código de programación resulta de vital importancia para el buen funcionamiento de cualquier comportamiento autónomo complejo. Este tema centra el estudio de las **arquitecturas software**.

1.2.4. Arquitecturas Software

El campo de estudio de las arquitecturas software para robots se centra en el modo en que se organizan los distintos componentes que forman parte del código de programación de un robot.

Imaginemos un robot compuesto de diversos sensores y actuadores manejados cada uno de ellos mediante una pieza de software distinto. Es necesario coordinar de algún modo cuándo el robot ha de procesar los datos provenientes de cierto sensor o

conjunto de sensores y cuándo y de qué manera ha de reaccionar el robot a dichos estímulos.

De modo que, además de los **algoritmos de control** de cada uno de los elementos que forman el robot, es necesario un nivel superior de abstracción que sea capaz de gestionar el funcionamiento de cada una de las piezas de software. Esto es así porque el robot, que es capaz de llevar a cabo un cierto número de **comportamientos**, no utiliza los mismos elementos de control para todas ellas.

Un ejemplo muy ilustrativo es el de un coche robot el cual tiene la capacidad de moverse evitando obstáculos gracias a un sensor de ultrasonidos localizado en su parte frontal y, además, en ausencia de luz enciende los faros.

En este caso, el robot posee dos comportamientos: moverse sin chocar y encender o apagar las luces.

Parece lógico que el software debería estar compuesto por un sistema de control que procese a cada instante las lecturas de distancia del sensor y comande a los motores las órdenes oportunas en consecuencia. Además, habría otro sistema de control que se encarga de encender o apagar los faros. Sin embargo, este segundo sistema no se encuentra en funcionamiento simultáneamente con el anterior sino que existe un estímulo concreto (en este caso, la ausencia de luz) que conlleva la **activación** de este comportamiento.

Esta es la tarea fundamental que ha de desempeñar la arquitectura software del robot.

En ocasiones, la forma de coordinar qué componente software ha de activarse o desactivarse en cada momento, no está completamente definida a nivel de arquitectura software sino que forma parte del diseño del sistema.

Así como la arquitectura define en qué lugares del sistema se encuentran los diferentes componentes software, además de sus interconexiones; el diseño abarca los detalles de implementación y funcionamiento del software.

Como cabe esperar, el diseño no puede ser agnóstico de la arquitectura, ya que la arquitectura software que se emplee influirá de manera directa en el diseño de cada uno de los componentes software.

Por este motivo, y aunque no sea considerada una arquitectura software, cabe mencionar que uno de los modelos de diseño de software más utilizados en robótica son las **Máquinas de Estado Finito (FSM)**, que sin ser una arquitectura en sentido riguroso sí permiten estructurar y organizar el conjunto de tareas que ha de desempeñar uno o varios componentes software.

1.3. Capacidades de un Robot

Los robots están destinados a realizar **tareas**. Estas tareas, pueden componerse de varios comportamientos. Por ejemplo, la tarea de cocinar puede englobar el comportamiento de vigilar las sartenes, atender a lo que el chef pide y mantener la mesa limpia.

Cada uno de estos comportamientos, tal y como se explicó en el capítulo anterior, no tienen porque desarrollarse de forma simultánea sino que la arquitectura software se encarga de gestionarlos.

A su vez, cada uno de estos comportamientos requieren de un conjunto de *habilidades* que el robot debe poseer para poder ejecutarlos. A estas habilidades se les denomina **capacidades**.

Resulta sencillo de entender si pensamos en que para que el robot sea capaz de atender a las peticiones del chef, éste debe poseer la capacidad de escuchar e interpretar comandos de voz. Así mismo, para poder vigilar las sartenes, el robot debe poseer la capacidad de la visión.

Por tanto, las capacidades necesarias para que un robot pueda llevar a cabo cualquier comportamiento, por complejo que sea, son las siguientes:

- Visión
- Navegación
- Diálogo
- Manipulación

A continuación, se abordarán brevemente cada una de estas capacidades.

1.3.1. Visión

La visión en robótica desempeña la importante labor de ser capaz de interpretar imágenes tratando así de emular el comportamiento de la vista humana.

Se trata de una capacidad que entraña gran complejidad y cuyo elemento principal son las **cámaras**.

Las cámaras son los sensores más sofisticados. Proporcionan un flujo desbordante de datos, lo cual hace que su procesamiento resulte en ocasiones computacionalmente costoso, es decir, requiere de hardware potente (buenos procesadores gráficos y gran cantidad de memoria) y un software lo más optimizado posible para obtener el máximo rendimiento de dicho hardware.

La mayoría de las cámaras comunes son las denominadas *RGB* debido a que capturan el color de cada píxel y éste es expresado como combinación lineal de tres colores que son el rojo, el verde y el azul (**R**ed, **G**reen, **B**lue).

Sin embargo, resultan de especial interés en el campo de la robótica las denominadas **cámaras RGBD**, como la que se ilustra en la figura 1.9. La peculiaridad de este tipo de cámaras es que son capaces de captar no sólo el color de cada píxel sino su distancia con respecto a la cámara. De este modo, es posible obtener imágenes tridimensionales.

Figura 1.9: Cámara RGBD *ASUS Xtion*

1.3.2. Navegación

La navegación atañe a la capacidad del robot para desplazarse desde un punto inicial a un punto de destino evitando colisionar con los posibles obstáculos que hubiera entre ambos puntos.

Normalmente, para que un robot sea capaz de navegar por un determinado entorno, hacen falta dos cosas:

- El robot debe poseer un **mapa** del entorno
- El robot ha de estar **localizado** en ese mapa

Conociendo la morfología del entorno que rodea al robot (proporcionada por el mapa) y la posición exacta que ocupa en dicho entorno (localizado en el mapa), un algoritmo será capaz de elaborar una ruta hasta un punto de destino que se encuentre en el mapa.

Un mapa, en definitiva, es una forma de representar el conocimiento que el robot posee de su entorno. Existen diferentes tipos de mapas, pero uno de los más comunes es el denominado *Occupancy Grid Map*, lo que se traduce al español como *Mapa de Celdas de Ocupación*. A continuación se muestra un ejemplo de este tipo de mapas.

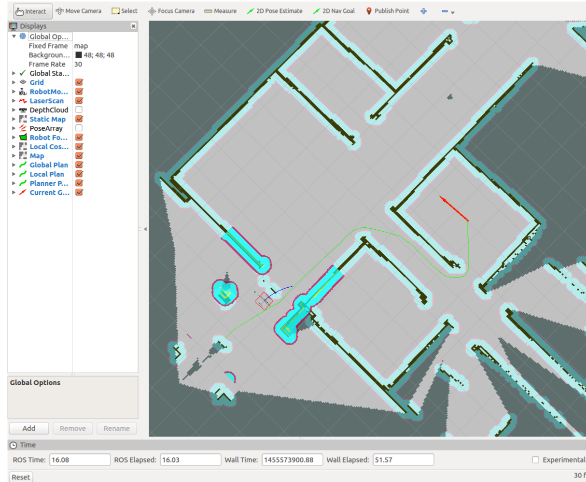


Figura 1.10: Occupancy Grid Map

Su principio de funcionamiento se basa en celdas. El entorno se divide en cuadrículas donde cada una de las celdas puede encontrarse en tres estados distintos: **libre**, **ocupada**, o **desconocido**.

Cuando el robot detecta con sus sensores que hay un obstáculo en un determinado punto, la celda correspondiente a ese punto en el mapa pasa al estado *ocupado*. Por consiguiente, todo el espacio vacío donde el sensor no detecta obstáculo son celdas *libres*. Por último, aquellas zonas inaccesibles para los sensores del robot y que, por tanto, se desconoce si están libres u ocupadas, dan lugar a celdas en el estado *desconocido*.

1.3.3. Diálogo

El diálogo concierne a la capacidad de interpretar órdenes o comandos de voz, así como ser capaz de responder del mismo modo.

La principal forma en la que los seres humanos nos relacionamos se basa en el lenguaje oral y gestual. Por tanto, para que un robot pueda comportarse de un modo **social** y pueda mostrarse cercano a cualquier ser humano, debe poseer estas capacidades.

Para abordar este problema, existe un campo de investigación conocido como **NLP** (Natural Language Processing).

El funcionamiento general de un algoritmo de NLP se basa en ser capaz de reconocer **intenciones** en la oraciones. Para ello, habitualmente se divide el texto a procesar en *tokens*. Un *token* se trata de una porción mínima de texto con sentido. Parece razonable que una buena aproximación para dividir un texto en *tokens* es separarlo en palabras o, dicho de otro modo, dividirlo por los espacios en blanco. Sin embargo, ésta no es la única manera de hacerlo.

Posteriormente, los *tokens* se asocian unos con otros con la finalidad de hallar qué intención tiene el hablante. Por ejemplo, para la oración *”Ve a la cocina y tráeme un vaso de agua, que tengo mucha sed”*, asociando los tokens *cocina*, *vaso* y *agua*, se

puede adivinar que la intención del hablante es que el robot traiga un vaso de agua de la cocina.

A partir de este momento, el robot podría elaborar una respuesta o iniciar una acción.

1.3.4. Manipulación

Es la capacidad que permite a un robot realizar cambios en su entorno mediante la variación de la posición de los objetos que lo forman.

Para que esto sea posible, el robot ha de estar dotado de un brazo con el cual pueda ser capaz de sostener objetos.

Por tanto, un robot que posea un brazo como el que se ilustró en la figura 1.7, podría ser capaz de agarrar elementos que se encuentren en su entorno y cambiarlos de lugar o de posición. Esto es lo que se conoce como manipulación.

Por consiguiente, el brazo debe ser capaz de moverse sin colisionar con los posibles obstáculos con los que se pudiera encontrar. Para ello, y de un modo parecido a como se hace en navegación (explicado anteriormente), ha de calcularse una **ruta** por la cual el brazo ha de moverse. Para ello, es necesario tener en cuenta la fisionomía del propio brazo robótico (longitud, número de motores, etc) y del objeto que se desea alcanzar.

Existen diversos paquetes de software que implementan algoritmos para abordar diferentes tareas de manipulación. Además, este campo de estudio tiene su origen anteriormente a la navegación, visión o procesamiento de lenguaje, pues estas tres áreas conciernen eminentemente a la robótica de investigación o **robótica social** mientras que la manipulación lleva muchos años implantada en los robots de carácter industrial como el que se muestra en la siguiente imagen.

Figura 1.11: Brazo Robótico Industrial

1.4. ¿Qué es ROS?

En secciones posteriores se abordan con mayor nivel de detalle los aspectos técnicos más relevantes de ROS. Este capítulo tiene como finalidad familiarizar al lector con los aspectos más básicos de lo que ROS representa, para así facilitar una mayor comprensión.

ROS, por sus siglas en inglés **Robot Operating System**, se trata de un conjuntos de librerías y herramientas que proporcionan al desarrollador todo lo necesario para la elaboración de sistemas robóticos complejos. Su arquitectura está basada en un sistema distribuido por intercambio de mensajes. Es decir, se compone de múltiples componentes software (Nodos), que son pequeños programas en ejecución, que se comunican entre sí para, de este modo, poder resolver tareas complejas.

ROS nace en el año 2006 en la *Universidad de Stanford* (EE.UU) con el ánimo de solucionar un problema recurrente hasta entonces en robótica: La constante reinvención de la rueda, es decir, los investigadores y desarrolladores dedicaban altos esfuerzos a crear software que abordaban siempre las mismas tareas. Debido a la falta de una metodología **estándar** de trabajo, era necesario escribir nuevo software cada vez que era necesaria la utilización de un nuevo sensor, actuador, o de algún protocolo de comunicaciones.

De este modo, se pretendió crear un **framework**, es decir, una capa software sobre la cual se pueda crear código de programación para un robot que fue construido en la propia Universidad. El robot denominado *PR1*.

Figura 1.12: Robot PR1

En 2008, El investigador y fundador de *Willow Garage* (centro de investigación enfocado a productos de robótica), Scott Hassan, se vio atraído por el proyecto desarrollado en la Universidad de Stanford hasta el punto en que decidió dar comienzo a un programa de robótica en Willow Garage utilizando dicho framework.

De esta manera, ROS fue desarrollado en Willow Garage tomando como base la primigenia idea desarrollada en la Universidad de Stanford. En 2009, la primera versión (distribución) de ROS fue liberada.

1.5. Estado del arte

Como he explicado anteriormente, una de las capacidades más importantes en robótica de investigación es la visión. Ésta se lleva a cabo haciendo uso de cámaras que proporcionan imágenes que pueden ser procesadas para extraer información relevante.

Sobre las imágenes se pueden realizar multitud de operaciones, así como pueden aplicarse algoritmos de procesamiento de distinta índole para obtener diferentes parámetros de interés. Sin embargo, la tarea más importante y necesaria para crear un sistema de visión similar al que poseen los humanos es la de extraer de dicha imagen objetos de interés.

Es decir, es necesario ser capaz de identificar diferentes clases de objetos que se encuentran en la imagen. De este problema se encarga lo que hoy denominamos como *Deep Learning*. El Deep Learning (aprendizaje profundo) es un área de la computación que se encarga de desarrollar software que permite la detección de objetos de interés en las imágenes. A los algoritmos encargados de llevar a cabo estas detecciones se les conoce como **Redes Neuronales** debido a que su principio de funcionamiento se asemeja al de las neuronas de un cerebro humano.

De modo que una red neuronal es capaz de *aprender* a identificar un número determinado de objetos y, una vez realizado este aprendizaje, sería capaz de detectarlos dando lugar a resultados como el que se muestra en la siguiente imagen.

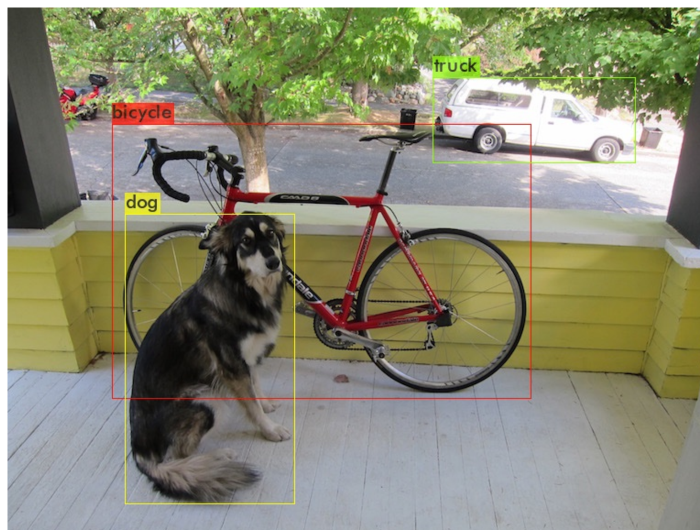


Figura 1.13: Ejemplo Detección Red Neuronal

Si aplicáramos el algoritmo de detección de una red neuronal no sólo a una imagen sino a todas las imágenes que la cámara proporciona a cada instante, seríamos capaces de poseer una detección instantánea (en tiempo real) de los objetos de interés que se encuentren en el campo de visión de la cámara.

Sin embargo, una restricción a tener en cuenta que no siempre hará esto posible, es la capacidad de cómputo que la red neuronal necesite para ejecutar su algoritmo de detección. Si dicho algoritmo requiere una alta capacidad de cómputo, el sistema

se ralentizará y no será posible procesar las imágenes en tiempo real. Por este motivo, generalmente las redes neuronales requieren de hardware específico capaz de ejecutarlas, como son las tarjetas gráficas.

1.5.1. Redes Neuronales

Una red neuronal se trata de un modelo computacional cuyo funcionamiento trata de asemejarse al cerebro animal. Los elementos que forman una red neuronal son: **número de neuronas**, **número de capas** y **tipo de conexión entre las capas**.

Tal y como se aprecia en la figura 1.14, una red neuronal siempre ha de poseer al menos dos capas que se corresponden con la entrada y la salida. Opcionalmente, podrá estar formada de más capas internas normalmente conocidas como capas *ocultas*.

En este caso, tenemos las capas de entrada y de salida; y dos capas ocultas.

Además, cada una de dichas capas pueden estar formadas por distinto número de neuronas que están conectadas con las neuronas de la siguiente capa.

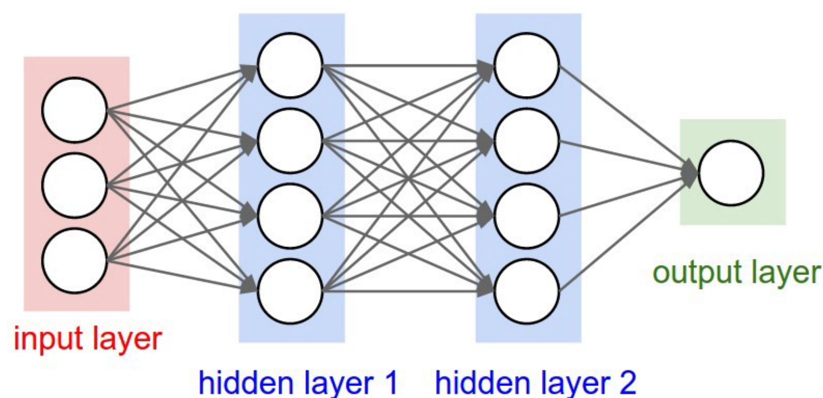


Figura 1.14: Esquema básico de una Red Neuronal

Pero... ¿A qué denominamos *neurona*[2]?

Funcionamiento de una Neurona

Una neurona se trata de un sistema que toma un determinado número de entradas, realiza ciertas operaciones sobre ellas y, posteriormente, genera una o varias salidas. En el caso de las redes neuronales, la entrada de una neurona proviene de la salida de una neurona de la capa anterior. Por consiguiente, la salida de una neurona supone la entrada de otra neurona situada en la siguiente capa.

Analizaremos el comportamiento interno de una neurona basándonos en la figura 1.15.

[2] A partir de este momento, se utilizará el concepto *neurona* para hacer referencia a las neuronas artificiales que componen las redes neuronales

El conjunto de datos X_i, donde $i \in [1, n]$, son las entradas de la neurona. Estos datos provienen de otras neuronas anteriores. En el caso de las neuronas de la primera capa, los datos de entrada son los datos de partida de la red neuronal.

Sin embargo, estos datos X_i se ponderan antes de entrar en la neurona. Es decir, se multiplican por unos valores denominados **pesos** y que en la figura se denotan como W_{ij}, donde $i \in [1, n]$ y j representa la capa.

Los pesos son coeficientes de escalado. Es decir, su función es amplificar o minimizar el valor de entrada a la neurona de modo que no todos los valores X_i tendrán la misma *influencia* en los cálculos realizados por la neurona.

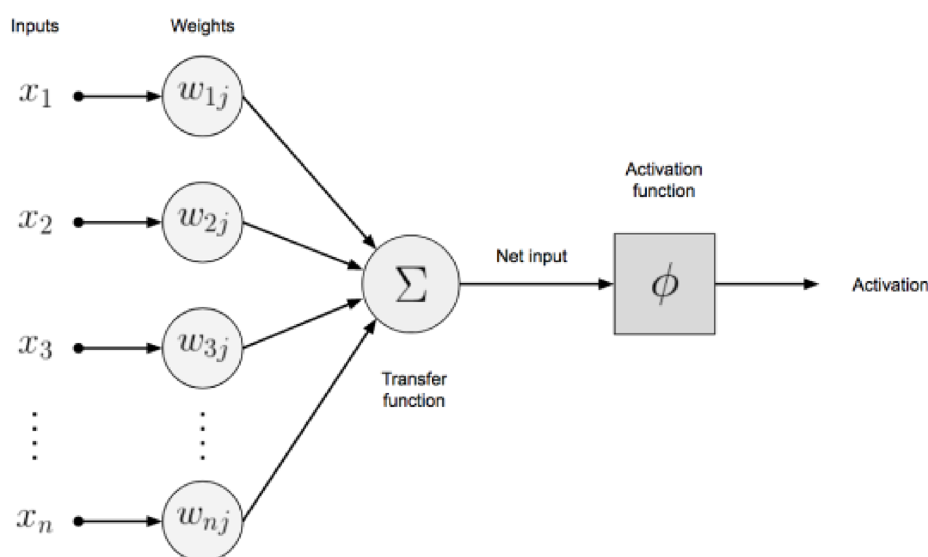


Figura 1.15: Funcionamiento Neurona Artificial

Por tanto, obtenemos que $net_input = \sum_{i=1}^{n} x_i * w_{ij} + b$, donde b se trata de una constante que se suma con el objeto de asegurar que al menos unos pocos nodos en cada capa se activan.

Por último, y tal como se ilustra en la figura 1.15, este valor calculado pasa por una función denominada **Función de Activación** que será la que decida si la neurona se activa o no.

Normalmente, el resultado que produce la función de activación suele encontrarse en el rango de 0 a 1 o de -1 a 1. Cuando una neurona da como resultado un valor distinto de cero, se dice que ésta se ha activado.

Existen diversos tipos de funciones de activación. En la figura 1.16 se muestra un ejemplo de la función *sigmoide* cuya expresión matemática es la siguiente:

$$g(x) = \frac{1}{1 + e^{-x}} \tag{1.2}$$

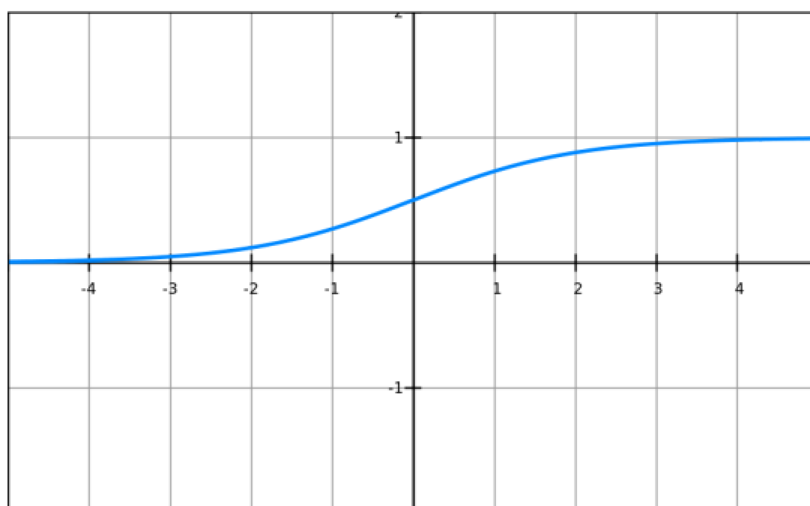


Figura 1.16: Función de Activación Sigmoide

Es función del **entrenamiento** ajustar los valores de los pesos de forma apropiada para que amplifiquen o atenúen las señales de entrada de forma correcta para que así la red neuronal sea lo más rigurosa posible en sus predicciones.

1.5.2. Limitaciones

Las redes neuronales convencionales resultan muy útiles para detección y clasificación de objetos. Sin embargo, dada la utilización de imágenes bidimensionales, sólo podemos obtener información en dos dimensiones. Esto significa que la clasificación de la detección que la red neuronal proporciona como salida siempre vendrá dada en píxeles. ¿De qué manera?

Bounding Boxes

Tal y como se aprecia en la figura 1.13, la clasificación de las detecciones consta del cálculo de las zonas que delimitan cada uno de los objetos detectados. En esta figura se aprecia, por ejemplo, la zona limítrofe de la bicicleta o la del perro.

Estas zonas limítrofes son las denominadas **bounding boxes**, y siempre se definen en forma poligonal. En las redes neuronales más comunes, son rectangulares por lo que un bounding box queda perfectamente definido por la posición (en píxeles) de sus vértices superior izquierdo e inferior derecho.

Una imagen digital se representa como un conjunto de píxeles dispuestos en dos dimensiones (x, y) donde cada uno de esos píxeles contiene información de color, normalmente en el espacio de colores denominado RGB (Red, Green, Blue).

Es decir, cada píxel contiene la cantidad de rojo, verde y azul que contiene. La combinación lineal de estos tres valores da lugar a toda la variedad de colores que se pueden representar.

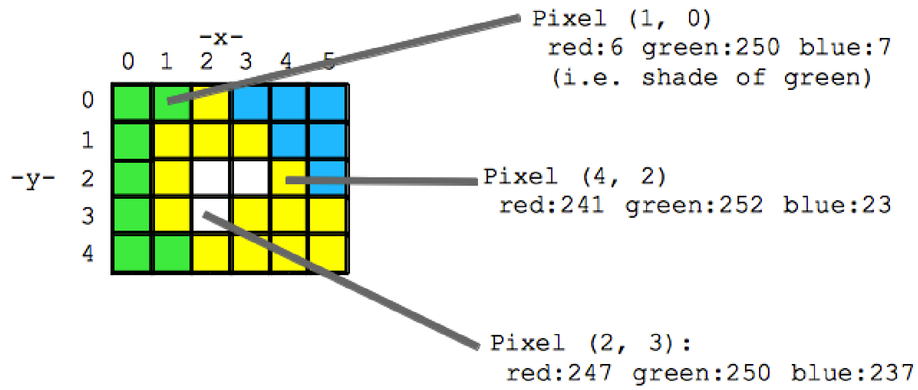


Figura 1.17: Morfología de una Imagen Digital

Dada la morfología de las imágenes digitales, representado en la figura 1.17, el sistema de coordenadas que se utiliza posee dos dimensiones (x, y) y cuyo eje de coordenadas se localiza en el vértice superior izquierdo. De esta forma, la coordenada y crece hacia abajo y la coordenada x crece hacia la derecha.

Imaginemos que dada una detección, tenemos un bounding box definidos por los vértices $v1(28, 3)$ y $v2(115, 82)$.

Sobre este bounding box podemos realizar diferentes cálculos. Por ejemplo, podemos saber que su ancho es de $115 - 28 = 87$ píxeles y su alto es $82 - 3 = 79$ píxeles.

Del mismo modo, su centro se podría calcular como $c(\frac{28+115}{2}, \frac{3+82}{2}) \Rightarrow c(71, 42)$.

Sin duda, esta clase de bounding boxes son muy interesantes para muchos tipos de software que requieran de detección de objetos. Por ejemplo, reconocimiento facial.

Sin embargo, desde el punto de vista de la robótica, resulta poco útil conocer los píxeles de una imagen que componen cierta detección de un objeto porque lo realmente útil y lo que permitirá al robot interactuar con dicho objeto y/o tomar decisiones es el conocimiento de la **localización espacial** de dicho objeto. Es decir, dónde se encuentra en términos físicos y cuál es su volumen.

Conociendo estos parámetros, podría dotarse un robot de la autonomía suficiente para manipular el objeto, o navegar por el entorno teniéndolo en cuenta (bien sea para acercarse a él, evitarlo, etc...).

2. Requisitos y Metodología

Para ser capaz de ofrecer una solución al problema de la detección espacial de objetos en el entorno de un robot, es necesario caracterizar qué requisitos ha de cumplir dicha herramienta.

En la mayoría de los casos, los robots poseen un **ordenador de abordo** en el cual se ejecuta el software que lo dota de autonomía. Sin embargo, no es así en todos los casos.

En ocasiones, el software es ejecutado en un ordenador independiente al robot. Especialmente cuando los requisitos computacionales son muy altos y requieren de ordenadores con altas capacidades hardware. En estos casos, el robot posee un ordenador cuya función es la de enviar y recibir paquetes de información con el anterior, teniendo de ese modo la carga computacional el ordenador remoto y no en el de abordo.

A lo largo de este capítulo se expone el estudio llevado a cabo sobre los requisitos que ha de cumplir una herramienta como *Gb Visual Detection 3D* destinada a su uso en sistemas robóticos complejos, y que dan lugar a la metodología de trabajo utilizada durante su concepción, desarrollo e implementación.

2.1. Sistemas Embebidos o Empotrados

Un sistema embebido se trata de una arquitectura de computación basada en microprocesadores o microcontroladores dedicados, normalmente, a llevar a cabo tareas muy concretas.

Este tipo de sistemas también se denominan **sistemas empotrados** debido a que su principal ventaja es que se encuentran integrados en el propio dispositivo. En este caso, en el robot.

Por lo tanto, es posible realizar tareas computacionales a bordo de un robot utilizando ciertos microprocesadores o microcontroladores que suelen tener reducidas dimensiones.

Sin embargo, en este tipo de sistemas, la **optimización** de los recursos es de vital importancia, ya que éstos son limitados.

2.2. Arquitectura Básica de un Computador

En la sección 1.2.3 se explica la diferencia entre microcontroladores y microprocesadores. Sin embargo, es preciso añadir algunos conceptos.

Los microcontroladores poseen lo que se denomina *memoria de instrucciones*. Esta memoria es donde se almacenan, en orden y de forma secuencial (una tras otra),

todas las instrucciones (operaciones) que se han de ejecutar. Por tanto, la Unidad Aritmético-Lógica de un microcontrolador se encarga de leer una a una dichas instrucciones y ejecutarlas. Por lo tanto, sólo se puede ejecutar un programa a la vez.

Sin embargo, los microprocesadores, normalmente, se destinan a sistemas más complejos que incluyen buses de entrada y salida, jerarquía de memoria (varios niveles de memoria: caché, RAM, disco duro, etc...). Por ello, además deben ser capaces de ejecutar varios programas simultáneamente.

2.2.1. Procesador Monociclo

Las instrucciones que el procesador ha de ejecutar se almacenan en la **memoria de instrucciones**. Por tanto, del mismo modo que los microcontroladores, el procesador debe leer cada una de estas instrucciones, decodificarlas y ejecutarlas en orden secuencial.

Los procesadores, además, poseen una memoria de muy rápido acceso de escritura y lectura denominada **registros**. Es en estos registros donde se almacena temporalmente los operandos antes de llevar a cabo una operación. Del mismo modo, el resultado de dicha operación también se vuelca a un registro. Los registros del procesador son muy limitados ya que se trata de una memoria muy pequeña. Sin embargo, su reducido tamaño hace que los tiempos de búsqueda sean muy cortos. Además, al encontrarse dentro de la propia electrónica del procesador, la latencia de acceso es muy pequeña.

Los procesadores mayormente utilizados en la actualidad poseen un repertorio de instrucciones que da luagr a los procesadores denominados *RISC*. Existen 3 tipos de instrucciones:

- Aritmético-Lógicas: Operaciones aritméticas (sumas y restas, principalmente) y operaciones lógicas.
- De Salto (Branch): Instrucciones que evalúan una condición y, en caso de cumplirse, transfiere el control del flujo del programa a una nueva dirección de memoria.
- Acceso a Memoria: Cargar datos desde la memoria de datos (Load) y almacenar datos en la memoria de datos (store).

Todas estas instrucciones, una vez codificadas, ocupan la misma longitud. Para ejecutar una nueva instrucción, el procesador debe decodificarla primero. Para ello, se llevan a cabo las siguientes **etapas**:

- **Fetch (F):** Buscar en la memoria de instrucciones la siguiente instrucción a ejecutar.
- **Decode (D):** Decodificación de la instrucción para obtener sus campos.
- **Execution (X):** Ejecución de la operación.
- **Memory Access (M):** En el caso de ser necesario, se acede a memoria de datos para leer (load) o escribir (store).

- **Writeback (WB):** Si es necesario, se almacena el resultado de la operación en un registro.

En todos los procesadores, existe una **señal de reloj**, como la que se ilustra en la figura 2.1.

Este tipo de señales se utilizan para sincronizar eventos. Se trata de una señal digital que sólo tiene dos valores (nivel alto y nivel bajo). Normalmente, el evento se produce en el flanco de nivel bajo a nivel alto, es decir, cuando la señal pasa de un nivel bajo a un nivel alto.

Por tanto, la velocidad de esta señal de reloj determina la velocidad con la que se llevan a cabo los eventos.

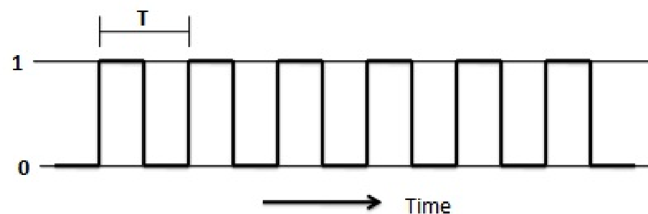


Figura 2.1: Señal de Reloj

En el caso del procesador monociclo, cada instrucción debe ejecutarse en un ciclo de reloj.

A los ciclos de reloj que transcurren durante la ejecución de una instrucción se le denonima *CPI* (ciclos por instrucción). Para el procesador Monociclo, CPI = 1.

2.2.2. Procesador Multiciclo

El procesador monociclo resulta ser notablemente ineficiente debido a que el ciclo de reloj debe tener la duración de la instrucción más larga. Esto da lugar a que haya muchos lapsos de tiempo en los que no se está realizando ninguna tarea en el procesador.

Para tratar de solventar este problema, los procesadores multiciclo no implementan el paradigma de CPI = 1 sino que la duración del ciclo de reloj viene dada por la **etapa más larga**. Por tanto, en el caso de la instrucción más corta (etapas F, D y X) se llevará a cabo en 3 ciclos de reloj. En el caso de la instrucción más larga (etapas F, D, X, M, WB) se llevarán a cabo en 5 ciclos de reloj.

2.2.3. Procesador Segmentado

Los anteriores tipos de arquitecturas de procesadores presentan limitaciones técnicas en la velocidad de ejecución que son capaces de alcanzar debido a que son secuenciales.

Es por esto que existen arquitecturas alternativas que son capaces de llevar a cabo más de una tarea o proceso simultáneamente. A este fenómeno se le denomina **paralelismo**. Existen dos tipos de paralelismo:

- *Paralelismo Interno o Implícito:* Una única CPU es capaz de paralelizar algunas tareas por medio de la **segmentación**
- *Paralelismo Explícito:* Es el que se consigue mediante replicación del hardware, es decir, varias CPU's.

Para llevar a cabo la segmentación, basta con comenzar la ejecución de una nueva instrucción en cada ciclo de reloj. De este modo, se obtiene un flujo de ejecución de instrucciones como el que se muestra en la figura 2.2. A esta técnica se le conoce como **Pipelining**.

Tal y como se aprecia, una vez que se ha finalizado la ejecución de la primera etapa de la primera instrucción, comienza a ejecutarse la segunda etapa de la primera instrucción y, además, la primera etapa de la segunda instrucción, de manera que se ejecutan simultáneamente.

Para poder aplicar esta técnica, son necesarias dos condiciones: Que el procesador sea multiciclo, ya que es necesario que cada etapa se ejecute en un ciclo de reloj distinto; y que no se acceda a los mismos componentes hardware en distintas etapas (en el caso de ser así, debe replicarse dicho elemento hardware). Esta última condición es el motivo por el cual, en este tipo de procesadores, se separa la memoria de instrucciones y la memoria de datos. En la etapa de *Fetch* se accede a memoria de instrucciones y en la etapa de *Memory Access* se accede a memoria de datos.

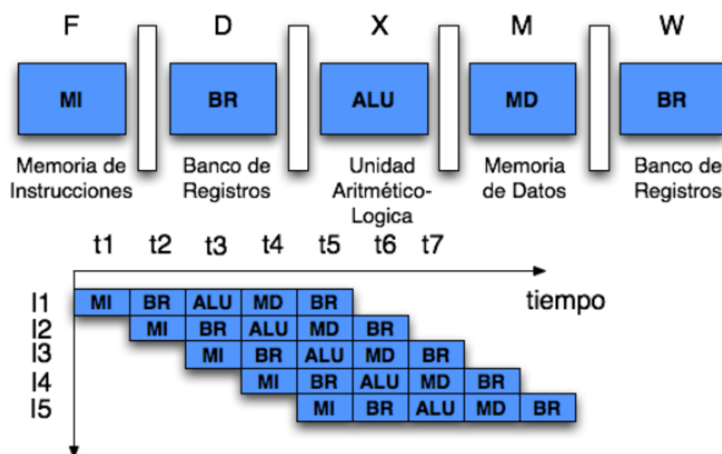


Figura 2.2: Ejemplo Pipelining

Este tipo de procesadores presentan mayor complejidad a nivel hardware así como en los compiladores para poder generar código capaz de ejecutarse en el procesador.

Sin embargo, la mayoría de procesadores que se usan a día de hoy son segmentados ya que su eficiencia supera notoriamente a los procesadores monociclo y multiciclo.

2.3. Sistemas Operativos

Un Sistema Operativo es un conjunto de programas que proporcionan una interfaz que facilita la utilización de la máquina de modo que no sea necesario preocuparse por la gestión del tiempo de ejecución de un programa, organización de la memoria... etc.

De este modo, la estructura del sistema se puede representar tal y como se ilustra en la figura 2.3.

Como se puede observar, en primer lugar se encuentra el Hardware de la máquina (CPU, memoria, disco duro, etc) y, sobre él, el sistema operativo. El sistema operativo interactúa con el procesador mediante el repertorio de instrucciones que éste le proporciona. Por tanto, **el sistema operativo ha de ser compatible con la arquitectura de procesador sobre el que se va a ejecutar**.

Por encima del sistema operativo se encuentran las librerías, que proporcionan una interfaz para permitir la interacción entre las aplicaciones y el sistema operativo mediante **llamadas al sistema**.

En último lugar se encuentran las aplicaciones de usuario.

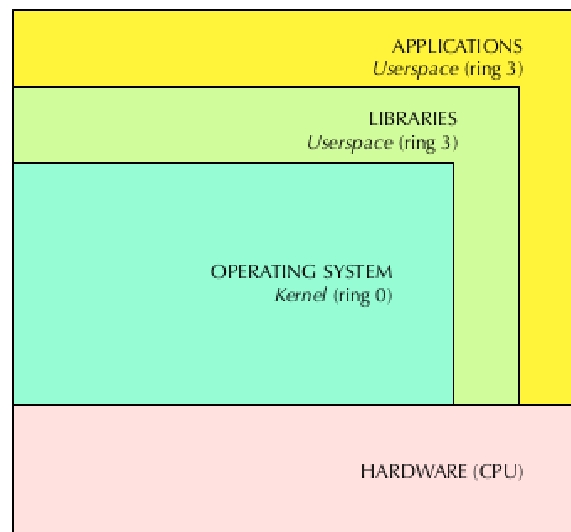


Figura 2.3: Estructura del Sistema

El Sistema Operativo es, por tanto, un componente software vital para el óptimo funcionamiento de un sistema de computación debido a que sin la **abstracción** que el sistema operativo proporciona de la máquina sería muy difícil programar software sofisticado.

Entre las abstracciones más importantes que el sistema operativo proporciona, destacan los **programas** y los **procesos**.

2.3.1. Programas y Procesos

Para facilitar el uso de la máquina, el sistema operativo proporciona una amplia variedad de conceptos abstractos como pueden ser los ficheros, los directorios, programas, procesos, etc.

Un programa es un conjunto de datos e instrucciones que implementan un algoritmo.

Un proceso es un programa en ejecución. Lo habitual es que se encuentren varios procesos ejecutándose a la vez y de forma independiente unos de otros.

Pero, ¿cómo se pueden ejecutar diversos procesos al mismo tiempo si los procesadores, tal como se expuso anteriormente, son secuenciales?

2.3.2. Planificación de Procesos

Los procesadores ejecutan instrucciones de un modo secuencial, es decir, una instrucción tras otra. En el mejor de los casos, tratándose del procesador segmentado, se lleva a cabo un paralelismo implícito que permite optimizar los recursos hardware de la CPU. Sin embargo, en una máquina podemos tener cientos de procesos ejecutándose a la vez. ¿Cómo es esto posible?

Una de las tareas fundamentales de las que ha de encargarse un sistema operativo es de la planificación de procesos.

El planificador (scheduler) del sistema operativo se encarga de gestionar qué proceso se ejecuta en la CPU en cada instante de tiempo. De manera que lo que se hace es multiplexar la CPU asignando *cuantos* (porciones) de tiempo a los distintos procesos. Cada proceso se ejecuta en la CPU durante ese intervalo de tiempo y, una vez finalizado, comienza a ejecutarse otro proceso desde el punto en el que hubiese quedado en su anterior ejecución.

Desde un punto de vista riguroso, todos estos procesos no se ejecutan de forma concurrente en la CPU pero gracias a la tarea del planificador, éste es el efecto que produce.

Tal y como se muestra en la figura 2.4, cada proceso puede encontrarse en uno de los tres estados *bloqueado*, *listo para ejecutar* y *ejecutando*.

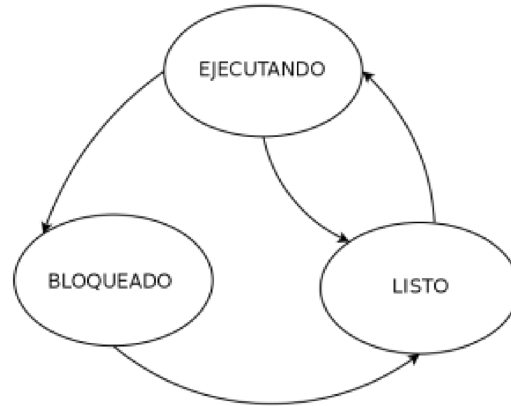


Figura 2.4: Máquina de Estados de Proceso

El planificador del sistema operativo se encarga, por tanto, de decidir en qué estado se encuentra cada proceso en cada momento.

2.4. Sistemas Distribuidos

El concepto de *Sistema Distribuido* se ha tratado de definir de diversas maneras a lo largo de la extensa literatura existente. Sin embargo, una definición, en mi opinión, muy acertada y sencilla es la que se proporciona en el libro titulado *Sistemas Distribuidos* de Andrew S. Tannenbaum [13], que dice así:

> Un sistema distribuido es una colección de ordenadores independientes que se presentan al usuario como un único sistema coherente.

Atendiendo y profundizando en esta definición, un sistema distribuido está formado por componentes (ordenadores y/o unidades de procesamiento) **autónomos** pero que, sin embargo, interactúan entre sí, bien sea compartiendo información, distribuyéndose las tareas, o de alguna otra forma; de modo que, el usuario de ese sistema distribuido lo percibe como un único sistema. Es decir, el usuario no tiene conocimiento de que dichas tareas se están ejecutando en un sistema distribuido porque su comportamiento es idéntico al que debería tener si se tratara de un sistema formado por un único computador.

Hoy en día, la mayoría de los sistemas destinados a robótica se basan en sistemas distribuidos debido a que proporciona ciertas ventajas, descritas a continuación, que son cruciales para poder implementar algoritmos robóticos complejos.

2.4.1. Recursos Accesibles

Una de las funciones principales de un sistema distribuido es facilitar el acceso a recursos. Bien sea por parte de un usuario o de las aplicaciones.

Permite poder acceder a recursos remotos y compartirlos de forma controlada y eficiente.

Para ejemplificar esta característica, pensemos en una flota de robots camareros cuya función es atender de la forma más eficiente posible a todas las mesas donde se encuentran los clientes. Bajo este contexto, sería razonable pensar que cada robot necesitase conocer la situación de cada uno de sus homólogos para así saber qué mesas están siendo ya atendidas y de este modo poder planificar su plan de acción.

La evidente necesitad de un sistema distribuido en este contexto es sencilla de entender del siguiente modo: Cada robot poseería un ordenador y, todos ellos, se comunicarían entre sí para compartir la información, es decir, los **recursos**.

2.4.2. Estandarización

Un sistema distribuido debe proporcionar un conjunto de reglas comunes y estándar para hacer posible la comunicación y cooperación de distintos sistemas autónomos entre sí.

Este conjunto de reglas comunes son los denominados **protocolos**.

Esta estandarización, a su vez, permite que distintos desarrolladores sean capaces de crear aplicaciones, así como sistemas autónomos completos, con capacidades totales de integración con otros sistemas.

2.4.3. Escalabilidad

La escalabilidad es un requisito muy importante de cualquier sistema distribuido y, por supuesto, también de los sistemas robóticos.

La escalabilidad hace referencia a que el sistema pueda ampliarse a un gran número de sistemas autónomos sin afectar a su rendimiento. Además, debe ser posible de aplicar mejoras al sistema de una manera sencilla. Cuando un sistema cumple estas características, se dice que el sistema **escala** o que es **escalable**.

Es habitual que la complejidad de las tareas a desempeñar por los robots en el ámbito de la robótica de investigación vaya en aumento. Este hecho, ligado a la necesidad, en ocasiones, de interacción entre diferentes sistemas, tal y como se menciona en el apartado anterior 2.4.3, convierte a los sistemas distribuidos en ideales para implementación de sistemas robóticos complejos.

2.5. Metodología

Tal y como se ha estudiado anteriormente, los sistemas robóticos se basan, en gran medida, en sistemas empotrados. Lo cual implica que la arquitectura de los procesadores cobra gran importancia para ser capaz de optimizar sus recursos. Del mismo modo, para poder gestionar de la forma más eficiente posible los recursos del procesador, el Sistema Operativo juega un papel fundamental.

Por tanto, debido a todas estas limitaciones y restricciones, se ha elegido el sistema operativo GNU/Linux como sistema operativo de desarrollo e implementación de mi herramienta.

GNU/Linux es un Sistema operativo basado en Open Source (Software Libre), lo cual significa que cualquier persona tiene acceso a su código fuente, así como a la modificación del mismo. Este concepto es muy importante porque desarrollar herramientas basadas en Open Source implica que la *comunidad* (usuarios y desarrolladores) tienen la posibilidad de contribuir a la mejora del sistema, así como de adecuarlo a nuevas tecnologías que puedan tener lugar.

Además, Linux es un sistema operativo ideal para su ejecución en máquinas con bajos recursos debido a sus políticas de planificación de procesos.

En cuanto a la ejecución de sistemas distribuidos, queda patente que resulta crucial para ser capaz de desplegar un sistema robótico robusto. Por tanto, es necesario que el sistema operativo elegido sea capaz de soportar este tipo de arquitecturas.

La gran mayoría de los sistemas distribuidos desplegados alrededor del mundo funcionan sobre Linux. En robótica, existen distintas herramientas que permiten aprovechar el máximo rendimiento de arquitecturas software basadas en procesos distribuidos haciendo uso de Linux. Se abordará esta cuestión en detalle en el próximo capítulo.

Por todos estos motivos, mi herramienta de percepción visual para robots implementa un modelo de software distribuido fácilmente escalable y estándar sobre un sistema operativo GNU/Linux que permite su óptima integración con una amplia variedad de sistemas embebidos.

Además, para el desarrollo de este proyecto se ha utilizado un modelo en cascada, representado en la figura 2.5 y cuya finalidad es establecer las distintas fases que tendrán lugar durante el ciclo de vida del producto, desde su concepción hasta su puesta en ejecución.

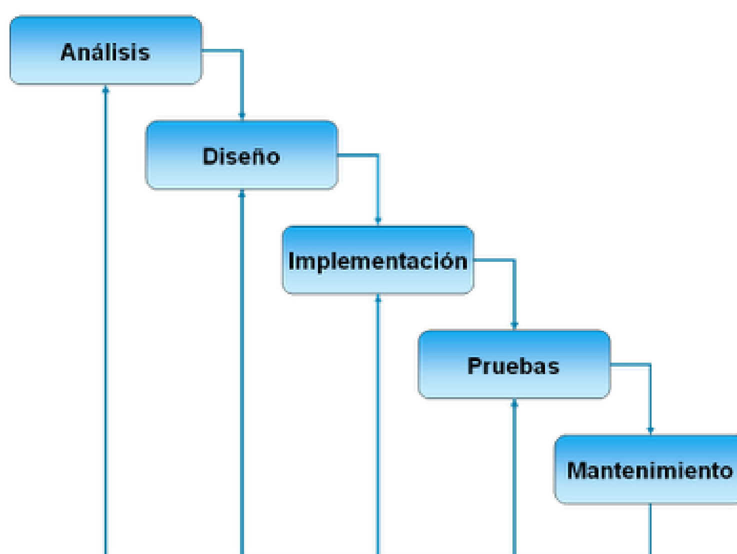


Figura 2.5: Modelo de Desarrollo en Cascada

El desarrollo de cualquier sistema complejo debe estructurarse en diferentes etapas o fases con el fin de garantizar la calidad en el producto final y buenas prácticas, aportando ambas valor al producto.

Estas fases pueden sufrir variaciones en función de la índole del proyecto. Sin embargo, las más habituales son las siguientes:

- **Análisis:** previamente al comienzo del desarrollo de un producto, es necesario analizar los problemas a resolver. Este análisis tiene como finalidad la elaboración de requisitos del sistema. Normalmente son requisitos de alto nivel y poco detallados.
- **Diseño:** durante esta etapa, los requisitos de alto nivel son refinados hasta el punto de conseguir unos requisitos más detallados (conocidos como requisitos de bajo nivel) y el detalle de la arquitectura (software o hardware) que se utilizará.
- **Implementación:** es la fase más técnica del desarrollo de un producto. En ella se lleva a cabo el diseño planteado en la etapa previa.
- **Pruebas:** se realizan todos los tests necesarios que permitan comprobar que la implementación cumple con las especificaciones elaboradas durante las etapa de análisis y diseño. Resulta vital tener unos requisitos bien definidos de modo que no sean ambiguos para poder evaluar de un modo preciso si el sistema implementado cumple con las necesidades reflejadas en los requisitos. Normalmente se realizan dos tipos de tests:
  - **Tests Unitarios:** pruebas que se realizan sobre segmentos de código muy concretos, en el caso del software; o componentes pequeños, en el caso del hardware. Se compara si el resultado obtenido de la operación es el deseado. Se suelen utilizar múltiples casos de prueba para evidenciar que esa pequeña porción del sistema funciona correctamente. Estos tests unitarios se llevan a cabo sobre múltiples porciones pequeñas y acotadas del sistema.
  - **Tests de Integración:** tras la realización de tests unitarios sobre distintos elementos aislados del sistema, se evalúa el correcto funcionamiento una vez que dichos elementos se encuentran interactuando entre ellos.
- **Mantenimiento:** una vez que el sistema está totalmente desarrollado y se encuentra operando, pueden surgir inconvenientes por parte de los usuarios, o puntos de mejora. En la fase de mantenimiento se recopilan todos estos datos con la finalidad de mejorar el sistema o adecuarlo a las nuevas necesidades que puedan surgir. Tal y como se aprecia en la figura, la fase de mantenimiento puede conllevar la reiteración de alguna de las etapas anteriores.

3. Entorno y Herramientas

A lo largo de este capítulo, se presentan aquellas herramientas que se han utilizado en el desarrollo de este trabajo. Se presentan tres herramientas fundamentales: **ROS**, **YOLO** y **YOLACT**.

3.1. ROS

3.1.1. Introducción

ROS (Robot Operating System) se trata de un framework, es decir, de un conjunto de librerías software y herramientas que permiten construir aplicaciones para robots.

Debido a las características concretas de ROS, comúnmente es definido como un **meta-sistema operativo**. Esto es así debido a su total integración sobre el Sistema Operativo de la máquina, añadiendo sobre éste una capa de software adicional que facilita el desarrollo de este tipo de aplicaciones destinadas a comportamientos robóticos.

ROS no es la única herramienta que permite el desarrollo de software para robots. Entonces, ¿Por qué ROS?

Tal y como se estudiará en capítulos posteriores, ROS está basado en la filosofía UNIX que trata de abordar problemas complejos mediante su descomposición en problemas sencillos de resolver. De este modo, ROS trabaja con un modelo de computación distribuida que permite que diversos programas encargados de resolver tareas muy concretas y simples sean capaz de colaborar al unísono para, de este modo, abordar tareas extremadamente complejas.

A continuación, se mencionan los pilares más importantes de la filosofía y métodos en los que ROS se fundamenta.

3.1.2. Comunidad

ROS es un proyecto Open Source, lo cual posibilita que multitud de personas alrededor del mundo contribuya a su desarrollo y evolución. De este modo, ROS posee una amplia comunidad de desarrolladores repartidos por todo el globo, lo cual lo convierte en un framework en constante crecimiento.

Poseer una comunidad de este calibre implica una gran facilidad para el acceso a documentación por parte de los desarrolladores de aplicaciones de ROS.

Existe un portal web denominado ROS Wiki que contiene toda la documentación sobre paquetes de software y librerías de ROS; Así como enlaces a diversos tipos de tutoriales para distintos niveles de conocimiento.

Además, existe a disposición de los usuarios un foro mediante el cual se discute sobre temas relacionados con el funcionamiento de ROS, así como de sus aplicaciones. Se trata de ROS Answers.

Gracias a la extensa comunidad de ROS, se facilita la capacidad de integración entre diferentes paquetes de software y, tal y como se explicó en secciones anteriores, esta característica es vital en el software robótico pues éste debe ser especialmente escalable.

3.1.3. Grafo de Computación

ROS implementa un sistema de computación distribuida mediante paso de mensajes. Esto significa que el framework de ROS proporciona a los programadores un conjunto de librerías mediante las cuales se puede llevar a cabo un sistema distribuido basado en el intercambio de mensajes entre diferentes *nodos*. Todos estos nodos y las comunicaciones que se producen entre ellos dan lugar a lo que se conoce como **Grafo de Computación**.

En la figura 3.1, donde se muestra un grafo de computación muy básico a modo de ejemplo, podemos apreciar tres nodos. Cada uno de estos nodos se trata de un programa que lleva a cabo una tarea concreta y, siguiendo la filosofía de UNIX, lo más sencilla posible.

A continuación, se ahondará sobre los elementos que conforman el grafo de computación y que posibilita sus comunicaciones.

Nodo ROS

Cada uno de estos nodos toma una serie de datos de entrada, posteriormente realiza una tarea de procesamiento o cálculo con esos datos y, por último genera una o varias salidas como resultado.

Cada una de estas entradas son proporcionadas por otro nodo, de igual modo que cada una de las salidas generadas son utilizadas como entrada de uno o más nodos para realizar sus propias tareas.

Las operaciones que es capaz de llevar a cabo un nodo pueden ser de distinta índole, sin embargo, la metodología que se utiliza es común. Comúnmente, cada nodo presenta una arquitectura **iterativa**, lo cual significa que el punto de entrada de la ejecución del software codificado en dicho nodo se basa en una función que contiene un bucle. Es decir, se ejecutan un número determinado de tareas secuenciales y, cuando estas tareas finalizan, vuelve a ejecutarse la misma secuencia.

Este tipo de arquitectura software es ampliamente utilizada en robótica. Esto se debe a que los sistemas en los que el robot ha de interactuar son sistemas **reactivos**.

Un sistema reactivo es aquel sistema que se caracteriza por la necesidad de toma de decisiones en base a los cambios que se producen de forma instantánea (a cada instante) en el entorno.

Para ejemplificar la implementación de un sistema reactivo básico que define un comportamiento robótico, se presenta el código en C++ 3.1 que implementa el comportamiento básico de un led que se enciende cuando la temperatura es mayor de 30ºC.

```c
void main(int argc, char ** argv)
{
    float temp;

    while (true)
    {
        temp = temperature();

        if (temp > 30)
        {
            encender_led();
        }
        else
        {
            apagar_led();
        }
    }
}
```

Extracto de código 3.1: Ejemplo Sistema Reactivo

Tal y como se puede observar, el código contiene un bucle que se ejecutará durante toda la vida del proceso. En el interior de este bucle, tenemos dos partes diferenciadas:

- Lectura de Datos: Obtenemos nuevos datos de entrada sobe los que realizar las operaciones. En este caso, los datos de entrada podrían provenir de un sensor de temperatura.
- Toma de Decisiones: En función de los datos de entrada, se genera una salida. En este caso, la salida produce directamente un efecto físico (encender o apagar un led) pero otra posible salida podría ser un 0 o un 1 que represente si el led ha de encenderse o apagarse. En este caso, podría existir otro programa (nodo) que tome este valor y se encargue de actualizar el estado del led.

Por tanto, este tipo de sistemas reactivos son vitales en la implementación de algoritmos robóticos dada la **impredecibilidad del entorno** en el que el robot desempeña sus funciones.

A continuación, se muestra un grafo de computación de ROS muy básico compuesto por un nodo **publicador** y dos nodos **subscriptores**:

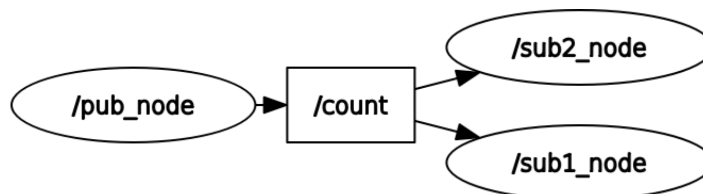


Figura 3.1: Grafo de Computación Básico

Como se puede observar, nos encontramos ante un grafo de computación formado por tres nodos denominados *pub_node*, *sub1_node* y *sub2_node*. Cada uno de estos nodos, tal como se explicó anteriormente, se trata de un proceso, es decir, un programa en ejecución que, gracias al framework ROS son capaces de intercambiar mensajes.

El intercambio de estos mensajes se lleva a cabo mediante el uso de sockets TCP, protocolo denominado **TCPROS**; o sockets UDP, que recibe el nombre de **UDPROS**.

Tal y como se mencionó anteriormente, un nodo puede intercambiar múltiples tipos de mensajes distintos. De igual forma, un nodo puede recibir más de un tipo de mensaje al mismo tiempo. Por este motivo, la topología de comunicaciones entre nodos que ROS implementa se basa en el método de **Publicación y Subscripción**.

Para ejemplificar el funcionamiento del método de publicación y subscripción podemos pensar en una emisora de radio. Una estación emisora emite cierta información (música, noticias, etc) en una frecuencia determinada. Por consiguiente, todos aquellos interesados en recibir dicha información no tienen más que sintonizar dicha frecuencia y automáticamente comenzarán a recibirla. Este es el funcionamiento básico de un método de publicación y subscripción.

Para hacer esto posible, ROS proporciona el concepto de **Topic**. Los topics son buses de comunicación de los que los nodos hacen uso para intercambiar sus mensajes. Cada uno de estos topics recibe un nombre. En el ejemplo gráfico de al figura 3.1 podemos apreciar un topic denominado */count*.

El nodo *pub_node* publica sus mensajes en */count* de modo que todos aquellos nodos que deseen recibir la información contenida en dichos mensajes, deben subscribirse a ese topic, como en el caso de *sub1_node* y *sub2_node*.

Esta topología de comunicaciones da lugar a un grafo de computación formado por nodos y topics. Este grafo de computación es la principal característica de ROS, ya que conforma el sistema de computación distribuida que se requiere en robótica.

ROS Master

El concepto de **ROS Master** es de vital importancia para comprender adecuadamente el funcionamiento del grafo de computación.

El ROS Master es nodo ROS que se encarga de gestionar el grafo de computación para que funcione adecuadamente y, de ese modo, el resto de nodos puedan comunicarse entre sí. ¿De qué manera lo hace?

En primer lugar, cuando un nodo es creado, éste se lo notifica al Master de manera que éste tiene constancia de todos los nodos que se encuentran en la red, sus direcciones IP y puertos TCP o UDP en cada caso.

De este modo, cuando un nodo desea subscribirse a un topic determinado, el ROS Master se encarga de notificar al nodo publicador que está publicando mensajes en ese topic, cuál es el nodo subscriptor (Su dirección IP y puerto). De igual forma, a dicho nodo subscriptor se le informa sobre el publicador. Por tanto, siguiendo este proceso,

ambos nodos ya pueden comunicarse entre ellos estableciendo un socket TCP o UDP. Este proceso se realiza para todos los nodos del grafo.

Por tanto, la función del ROS Master es crucial para que este sistema distribuido funcione correctamente.

Deslocalización de los Nodos

El hecho de que ROS utilice la pila TCP/IP para llevar a cabo sus comunicaciones, proporciona gran fortaleza a este sistema haciéndolo completamente distribuido no sólo a nivel de proceso sino físicamente. Esto significa que cada uno de esos nodos que conforman el grafo de computación, incluido el ROS Master, pueden ejecutarse en distintas máquinas siempre y cuando todas ellas pertenezcan a la misma subred. Esto permite, por tanto, ejecutar ciertos nodos que quizá realizan tareas computacionalmente más complejas, fuera del ordenador de abordo del robot como por ejemplo, en un PC *servidor*.

De igual forma, esta característica permite que diversos robots, cada uno con su computador embarcado, sean capaces de permanecer en constante comunicación. Este hecho supone una gran fortaleza ya que permite facilitar notoriamente la creación de algoritmos capaces de gestionar flotas de robots que actúen al unísono.

3.1.4. Depuración y Monitorización

ROS proporciona un amplio número de herramientas que permiten una óptima monitorización del grafo de computación, permitiendo de ese modo facilitar la depuración del sistema, corregir y/o evitar errores, etc.

Alguna de las herramientas más básicas y simples son los comandos *rostopic*, *rosnode* y *rosmsg* cuyas funciones están relacionadas con los topics de ROS, los nodos y los mensajes respectivamente.

Cada uno de estos comandos recibe, opcionalmente, diferentes parámetros que permiten listar los topics y los nodos activos en el grafo de computación, así como los tipos mensajes disponibles en el sistema (y que por tanto, pueden usarse en la comunicación entre nodos). A continuación, en la figura 3.2, se muestra un ejemplo de la salida que proporcionan los comandos *rosnode list* y *rostopic info* para el grafo de computación ejemplificado en la figura 3.1.

```
fernando@ubuntu-ssd:~$ rosnode list
/pub_node
/rosout
/sub1_node
/sub2_node
fernando@ubuntu-ssd:~$
```

```
fernando@ubuntu-ssd:~$ rostopic info /count
Type: std_msgs/Int32

Publishers:
 * /pub_node (http://ubuntu-ssd:45777/)

Subscribers:
 * /sub2_node (http://ubuntu-ssd:43265/)
 * /sub1_node (http://ubuntu-ssd:38707/)


fernando@ubuntu-ssd:~$
```

Figura 3.2: Ejemplo Comandos *rosnode* y *rostopic*

Tal y como se aprecia, el comando *rosnode list* ofrece los nodos que conforman el grafo de computación que se encuentra en ejecución. Por tanto, podemos discernir el nodo publicador (*pub_node*) y los dos nodos subscriptores (*sub1_node* y *sub2_node*). Pero además, existe un nodo denominado *rosout*. Este nodo se trata del ROS Master.

Resulta además de gran interés analizar el resultado del comando *rostpic info*. Este comando recibe como argumento el nombre del topic del cual se desea la información. En este caso, */count*. Este comando siempre ofrece la información desglosada en tres bloques:

- **Tipo de Mensaje:** En este caso, *std_msgs/Int32*. Como se ha explicado anteriormente, cada topic debe tener asignado un tipo de mensaje que será el que se transmita por él. En la próxima sección se ahondará en esta cuestión.
- **Publicadores:** Todos los nodos que publican mensajes en este topic. En este caso, encontramos el nodo *pub_node*. Además, se muestra la dirección IP y el puerto en el que se encuentra dicho nodo con el formato *IP:puerto*. En ocasiones, la IP se reemplaza por el nombre de host de la máquina en la que se ejecuta.
- **Subscriptores:** Se muestran, haciendo uso del mismo formato que en la sección anterior, todos los nodos que se encuentran subscritos a este topic. En el ejemplo se muestran los dos que ya conocemos.

RViz

Otra herramienta destacable es RViz. Se trata un programa con interfaz gráfica que permite monitorizar múltiples tipos de mensajes, pudiendo de este modo visualizar los datos que se intercambian a través de los diferentes topics. Esta funcionalidad resulta de gran utilidad cuando los mensajes que se intercambian los nodos son imágenes (bidimensionales o tridimensionales) pues cada uno de estas imágenes se pueden visualizar. En la figura 3.3 se muestra un ejemplo. También pueden visualizarse con RVIz trayectorias de robots en movimiento, la detección de sensores de distinta índole, etc.

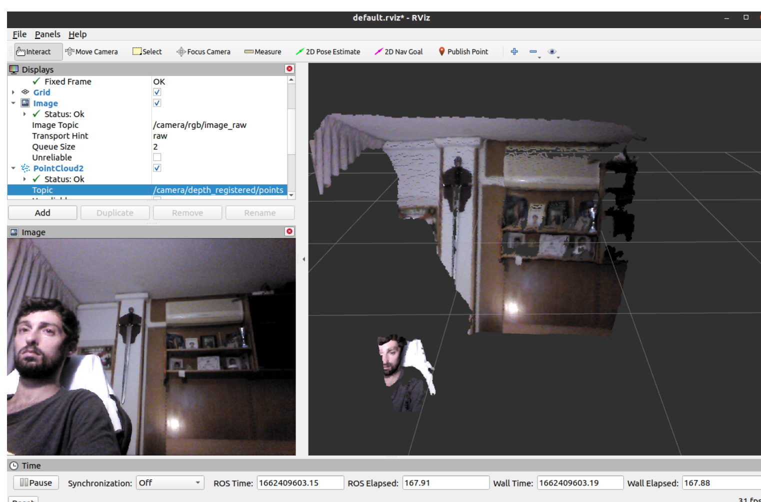


Figura 3.3: Ejemplo Visualización de Imágenes con RViz

A lo largo del desarrollo del presente trabajo se hará referencia a RViz en distintas ocasiones para visualizar de manera gráfica los datos que se obtienen.

3.1.5. Estandarización

La estandarización hace referencia a que diferentes piezas de software sean capaces de interactuar entre sí, o incluso integrarse una con la otra sin demasiado esfuerzo debido a que ambas comparten un conjunto de características comunes que las convierten en **compatibles**. Es pertinente explicar este concepto llegados a este punto porque lo que convierte a ROS en un framework estándar es, fundamentalmente, sus mensajes. Los tipos de mensajes de ROS, que permiten conformar mensajes que contienen datos para que puedan ser intercambiados entre nodos a través de los topics, siguen un prototipo común y una forma concreta de ser definidos. En la figura 3.4 se muestra cuál es la definición de un tipo de mensaje concreto. Se trata del tipo de mensaje *Twist* englobado dentro del paquete ROS *geometry_msgs* que contiene, además de este, otros tipos de mensajes.

Este tipo de mensaje se utiliza para expresar velocidades. Por ello contiene los los campos *linear* y *angular* (velocidades lineal y angular, respectivamente).

Por tanto, y tal como se puede observar, un tipo de mensaje de ROS puede estar definido, a su vez, por otros tipos de mensajes. En este caso se trata de dos campos de tipo *geometry_msgs/Vector3* que se constituyen a su vez por tres campos de tipo *float64* (número en coma flotante que se expresa en 64 bits de memoria).

```
fernando@ubuntu-ssd:~$ rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figura 3.4: Ejemplo del comando *rosmsg*

Al poseer cada tipo de mensajes una estructura definida, cualquier programador puede codificar un Nodo ROS que sea capaz de subscribirse a un topic determinado por el que reciba mensajes de dicho tipo y realizar el procesado pertinente sobre los mismos. De este modo, el programador no debe preocuparse por cómo fue programado el nodo publicador, ya que lo único que es necesario saber es qué tipo de mensaje se va a recibir puesto que la estructura nunca varía.

Esta esencia de la estandarización se encuentra, además, íntimamente relacionada con el concepto mencionado en secciones previas de que el software para robots debe ser modular y escalable, ya que de otro modo sería inabarcable la codificación de algoritmos que regulen tareas complejas.

Gracias a que los mensajes son estándar y la arquitectura de software distribuido mediante intercambio de mensajes por publicación-subscripción que ROS nos ofrece, lo convierte en un meta sistema operativo adecuado para abordar este tipo de algoritmos destinados a comportamientos robóticos.

3.1.6. Sistema de Ficheros

ROS proporciona su propia estructura de ficheros, lo que permite la correcta organización de cada una de las piezas de código (paquetes ROS).

La manera habitual de trabajar en ROS es haciendo uso de un *workspace* (o espacio de trabajo), el cual albergará todos los paquetes que se deseen desarrollar o utilizar.

Además, ROS permite compilar todo el código fuente contenido dentro de un workspace con un solo comando. Del mismo modo, los ficheros ejecutables generados se pueden ejecutar sin necesidad de conocer previamente la ruta exacta en la que se encuentran. Haciendo uso de los comandos *rosrun* o *roslaunch*, el meta sistema operativo ROS buscará los binarios necesarios dentro del árbol de ficheros de cada workspace.

3.2. YOLO

3.2.1. Introducción

La tarea de detectar y localizar categorías genéricas de objetos en una imagen estática resulta altamente complicada debido a la gran variedad de posibles variaciones como la iluminación, la orientación, tamaño, deformación, etc. Además, un mismo objeto puede presentar diferentes formas y colores, como puede ser el caso de un coche, por ejemplo, o de una persona que puede llevar distinta ropa o encontrarse en distintas posturas.

YOLO, *You Only Look Once*, se trata de un sistema *Open Source* (software libre) de detección de objetos en tiempo real. Debido a su algoritmo, es capaz de detectar múltiples objetos en una misma imagen a frecuencias notablemente altas, incluso llegando a los 150 FPS (frames por segundo). Esto significa que el sistema sería capaz de realizar detecciones de objetos sobre 150 imágenes distintas a cada segundo. Por supuesto, este rendimiento dependerá, tal y como se menciona en capítulos anteriores, de la potencia de cálculo del que disponga el computador en el que se esté ejecutando este software.

3.2.2. Entrenamiento

Anteriormente a la ejecución de YOLO, es necesario que la red neuronal sea entrenada. Durante el proceso de entrenamiento, se obtienen los valores óptimos de los

pesos de la red neuronal (ver sección 1.5.1).

Obtener unos pesos adecuados hará que cada una de las capas que forman la red neuronal sea capaz de computar sus parámetros de salida adecuadamente con el fin de obtener finalmente una detección fidedigna.

Para llevar a cabo este entrenamiento, se parte de un conjunto amplio de imágenes que contengan todas aquellas clases de objetos que se pretende que la red neuronal sea capaz de detectar. Es deseable una gran variedad de imágenes bajo diferentes ángulos, distintas condiciones de iluminación, saturación, etc...

A este conjunto de imágenes se le denomina **dataset**. Lo habitual es componer dos datasets distintos: Uno para el entrenamiento, y otro para ejecutar tests tras la finalización del entrenamiento con la finalidad de comprobar que el entrenamiento ha sido satisfactorio. Ambos datasets **no deben mezclarse**, es decir, no se debe usar ninguna imagen del dataset de test para entrenar ya que entonces no se puede validar de forma eficiente el correcto funcionamiento de la red neuronal.

Una vez definido el dataset de entrenamiento, lo que se debe hacer es etiquetar cada una de las imágenes, de modo que la red neuronal pueda distinguir cada una de las clases (aún está en la fase de *aprendizaje*).

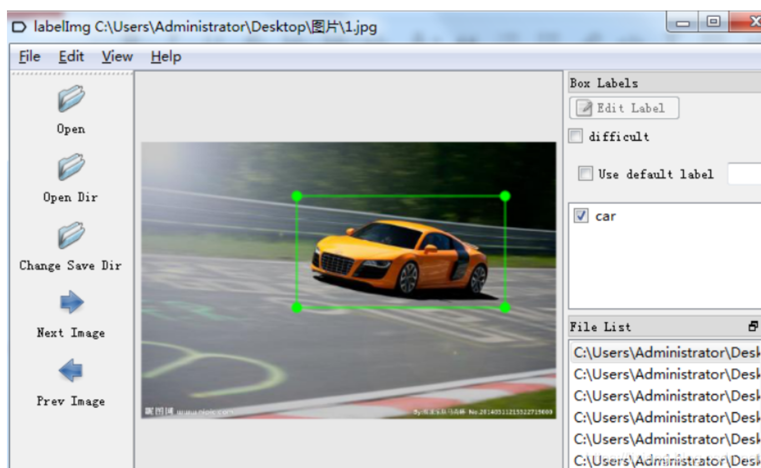


Figura 3.5: Ejemplo de Herramienta de Etiquetado: *Label Img*

La figura 3.5 representa un ejemplo de etiquetado de una imagen con una herramienta denominada *Label Img*. Se trata de una herramienta escrita en lenguaje Python, pero no es la única, pues existen diversas herramientas de etiquetado.

Como se puede apreciar, durante el etiquetado, se define **manualmente** el bounding box que rodea al objeto (en este ejemplo, el coche) y se le asigna una etiqueta (en este caso, *car*).

3.2.3. Algoritmo

A continuación, se explica de forma muy esquemática el algoritmo de detección de bounding boxes. Analizar en profundidad el funcionamiento de YOLO no se encuentra

entre los objetivos de este trabajo.

La red neuronal utiliza bounding boxes precalculados, basándose en el bounding box de la detección anterior. Utiliza 4 coordenadas para cada bounding box: t_x, t_y, t_w y t_h, siendo t_x, t_y las coordenadas de los píxeles del vértice superior izquierda. t_w y t_h corresponden al hancho y alto. De este modo, queda completamente definido el bounding box con forma rectangular.

En caso de que un bounding box se superponga más de un determinado offset con el que se obtuvo en la predicción previa, se descarta ya que se asume que es el mismo objeto. Estos bounding boxes no contienen ninguna clase, pues no se ha realizado aún la predicción de las clases.

Posteriormente, se realiza la predicción de la clase a la que pertenece (o puede pertenecer) cada bounding box. Para ello se utiliza la información obtenida durante la fase de **entrenamiento** de la red neuronal.

El motivo de la división del frame en bounding boxes es que realizar la detección sobre diferentes partes de la imagen es más eficiente (en términos computacionales y en términos de disminución del error) que aplicar una detección al frame completo.

3.3. YOLACT

YOLACT se trata de una red neuronal similar a YOLO. La diferencia entre ambas es que YOLACT, además de realizar detección de objetos, realiza lo que se denomina *segmentación*. Esta segmentación proporciona una **máscara** que permite distinguir los límites del objeto detectado del resto de la imagen, tal y como se ilustra en la figura 3.6.

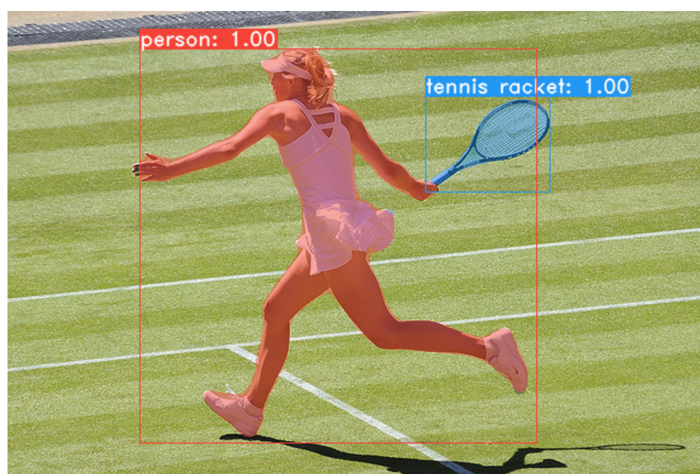


Figura 3.6: Yolact: Ejemplo de Detección

En el ejemplo, se pueden distinguir dos clases de objetos detectadas: una persona y una raqueta de tenis. Tal y como se observa, además de proporcionar dicha detección que incluye el nombre de la clase y el bounding box, se proporciona una máscara para cada objeto que permite distinguir con exactitud qué píxeles pertenecen al objeto.

Gracias a esta máscara que YOLACT proporciona, cualquier cálculo que se pretenda llevar a cabo utilizando la información facilitada por la red neuronal, será más preciso.

3.3.1. Etiquetado

Para que YOLACT sea capaz de proporcionar la máscara, es necesario que el etiquetado del dataset que se usará durante la fase de entrenamiento de la red neuronal se lleve a cabo de un modo distinto a como se hace en el caso de YOLO.

El etiquetado debe definir de la forma más precisa posible el borde exacto del objeto. Por lo tanto, no es suficiente con especificar un bounding box rectangular sino que éste debe ser poligonal.

En la figura 3.7 se muestra un ejemplo de etiquetado con el programa *Labelme*. Como se puede observar, cada bounding box está formado por un conjunto de vértices unidos entre sí, dando lugar a bounding boxes poligonales.

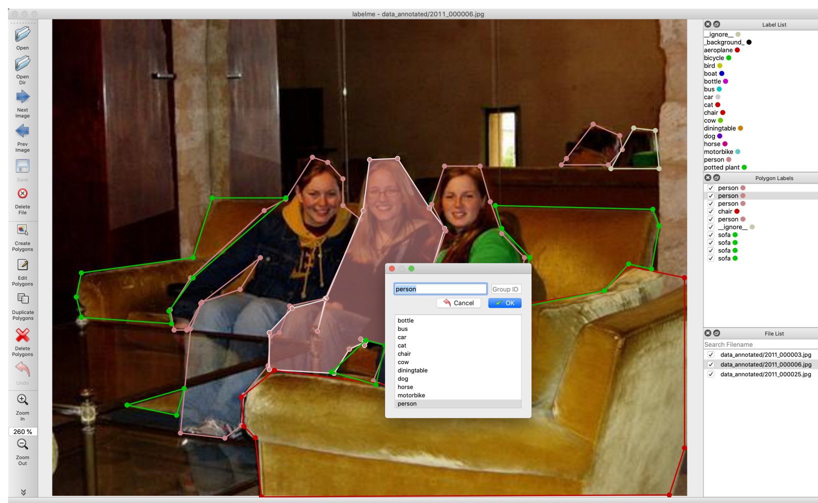


Figura 3.7: Ejemplo de Etiquetado con *Labelme*

La detección que tendría lugar con el etiquetado mostrado en el ejemplo, sería algo similar a la figura que se muestra a continuación:

Figura 3.8: Detección Yolact

A diferencia de YOLO, el entrenamiento resulta más dificultoso. Sin embargo, los resultados obtenidos durante la ejecución de la red neuronal resultan ser notablemente más precisos.

La utilización de YOLO o de YOLACT dependerá de la aplicación que se le desee dar.

4. Desarrollo del Trabajo

4.1. Preámbulo

La herramienta **Gb Visual Detection 3D** ha sido implementada de dos modos distintos, dando lugar a dos paquetes ROS distintos:

- **Darknet ROS 3D**: Utiliza la red neuronal YOLO, haciendo uso del paquete ROS que hace posible su utilización dentro del framework, denominado *darknet_ros*.
- **Yolact ROS 3D**: Utiliza la red neuronal YOLACT, haciendo uso del paquete ROS que hace posible su utilización dentro del framework, denominado *yolact_ros*.

A todos los efectos, la algoritmia y los métodos utilizados para la implementación de ambos paquetes son los mismos, así como su arquitectura y diseño.

El motivo de la existencia de ambas implementaciones de la herramienta se debe a motivos de experimentación con la pretensión de obtener mayor precisión debido a la utilización de YOLACT.

YOLACT, en efecto, mejora la precisión de la herramienta, aunque con peor rendimiento en cuanto a recursos computacionales. La red neuronal YOLACT necesita recursos más potentes que YOLO para poder alcanzar su mejor rendimiento.

Yolact ROS 3D no ha sido evaluada en ninguno de los computadores embebidos mencionados en el capítulo 5 debido a que estos computadores no presentan los recursos necesarios.

Durante este capítulo, para la explicación del desarrollo de la herramienta, se tomará como ejemplo *Darknet ROS 3D*, ya que todo lo expuesto es aplicable para ambas implemetaciones.

4.2. Darknet ROS 3D

4.2.1. Introducción

Darknet ROS 3D se trata de un paquete ROS implementado en lenguaje C++ cuya finalidad es la detección de objetos, así como el cálculo de su posicionamiento en el entorno y sus dimensiones. De esta manera, puede ser utilizado para complementar algoritmos de navegación donde sea necesario tener en cuenta posibles obstáculos inesperados. Así mismo, otro campo de aplicación es el de la manipulación. En este caso, es vital conocer la posición exacta del objeto y sus dimensiones pues sobre estos datos se realizarán los cálculos necesarios que determinarán los movimientos pertinentes para poder manipularlo.

Este paquete software, por tanto, dotará al robot de la capacidad de visión necesaria para que éste sea capaz de llevar a cabo múltiples tareas de distinta índole.

Como componentes hardware, es necesaria una cámara RGBD, así como un computador capaz de ejecutar el Sistema Operativo Linux con el framework ROS.

4.2.2. Estructura del Sistema

En esta sección, se explica de qué manera se integra el sistema de detección de objetos que *Darknet ROS 3D* representa con el resto de paquetes de ROS. Para facilitar este análisis, se abordará desde una perspectiva de **sistema** teniendo en cuenta, por tanto, sus entradas y salidas.

A nivel de software, la estructura de este paquete se basa en un único nodo que, esencialmente, se subscribe al topic de salida de *darknet_ros* y al del pointcloud de la cámara RGBD. Los datos de salida se publican en otros dos topics.

Tal y como se aprecia en la figura 4.1, el nodo principal de *Darknet ROS 3D* se subscribe a los topics del Point Cloud que publica el driver de la cámara y a los bounding boxes que la red neural YOLO (paquete *darknet_ros*) publica.

Además, cabe destacar que el paquete *darknet_ros* necesita subscribirse también a una de las salidas del driver de la cámara. En este caso, a la imagen RGB que éste proporciona. De este modo, podrá realizar sus cálculos y generar la salida de la red neuronal.

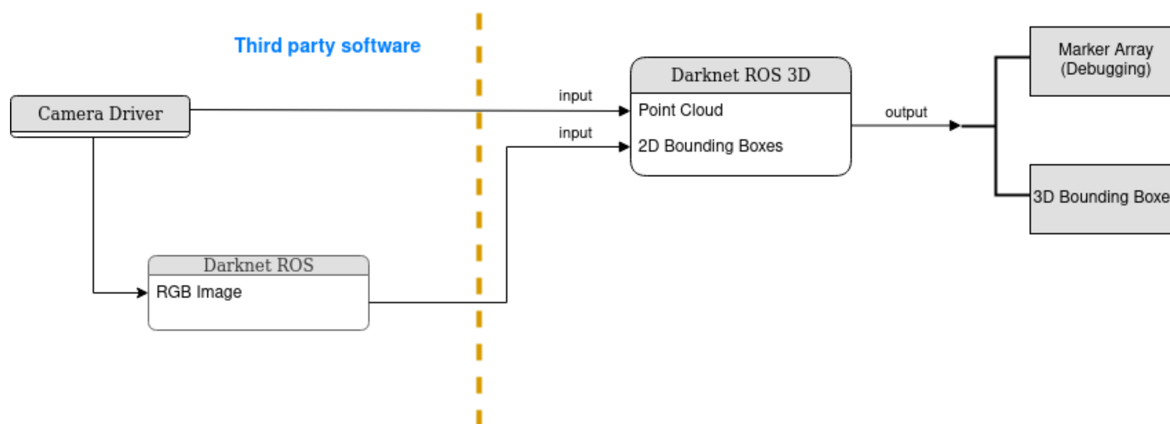


Figura 4.1: Arquitectura Software de Darknet ROS 3D

La estructura de este software puede entenderse como un sistema compuesto por diferentes subsistemas. Cada uno de estos subsistemas (driver de la cámara, *Darknet ROS* y *Darknet ROS 3D*) funcionan al unísono dando lugar al grafo de computación que se ilustra en la figura 4.2. Cabe mencionar que el concepto de *subsistema* es una mera abstracción para una mejor comprensión de la estructura del sistema completo pero en ningún caso forma parte de la terminología específica de ROS.

Tanto el driver de la cámara como *Darknet ROS* son componentes software necesarios para el funcionamiento de *Darknet ROS 3D*. Sin embargo, tal y como se detalla

más adelante, *Darknet ROS 3D* podría tomar como entradas la salida de cualquier otro sistema que no sea *Darknet ROS* siempre y cuando su tipo de mensajes sea el mismo. Este hecho dota al software de la capacidad de ser reutilizable. Esto se debe a que se utiliza el meta sistema operativo ROS.

A continuación, se encuentra un diagrama que muestra, con mayor nivel de detalle, cada uno de los subsistemas mencionados, así como sus interconexiones. Este conjunto de nodos y arcos, interconectados entre sí mediante topics, da lugar al grafo de computación del sistema.

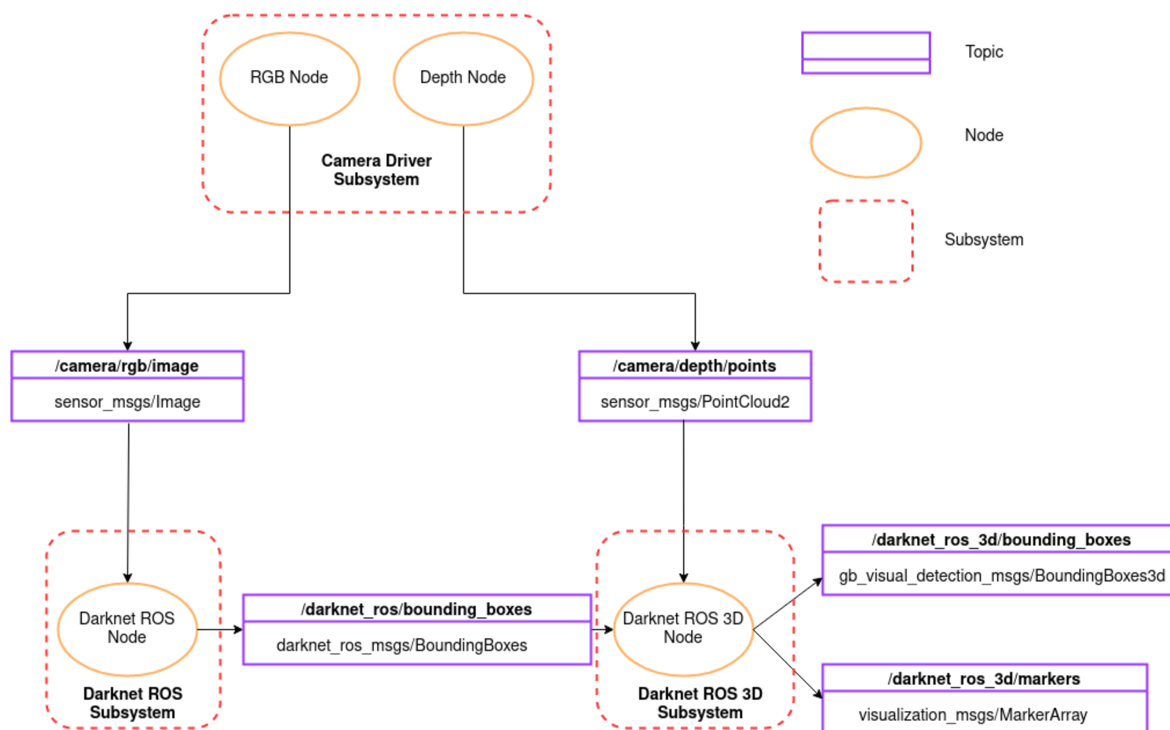


Figura 4.2: Grafo Computacional de Darknet ROS 3D

Tanto el driver de la cámara RGBD como el paquete *darknet_ros* suponen **dependencias** necesarias para el correcto funcionamiento de *Darknet ROS 3D*.

Dependiendo del modelo de cámara que se quiera utilizar, será necesario usar un paquete ROS u otro como driver de dicha cámara. Gracias al uso de ROS, el paquete *Darknet ROS 3D* es totalmente agnóstico del driver de la cámara, lo que significa que no importa que se use un driver u otro. Lo único importante es que este driver proporcione mensajes de tipo *sensor_msgs/Image* y *sensor_msgs/PointCloud2*.

Darknet ROS 3D se trata de un paquete ROS que contiene un único nodo que se subscribe al topic del point cloud de la cámara RGBD y al topic de los bounding boxes que proporciona *darknet_ros*. Posteriormente, combina la información obtenida de ambos topics para calcular nuevos bounding boxes, los cuales son publicados.

Además, se publican en otro topic mensajes de tipo *visualization_msgs/MarkerArray* que son utilizados para visualización y depuración con RViz. Más adelante se ahondará con mayor detalle en su propósito.

Por tanto, y tal como se aprecia en la figura 4.2, el paquete *Darknet ROS 3D* se compone de un único nodo subscriptor y publicador a la vez, subscribiéndose a dos topics (entradas del sistema) y publicando en otros dos topics (salidas del sistema).

A continuación, analizaré cada una de estas entradas y salidas del sistema. Como todas ellas se tratan de topics, cada uno de ellos tiene asociado un tipo de mensaje. Los nombres de estos topics que se especifican en la figura 4.2 son de ejemplo. El nombre específico de estos topics es irrelevante ya que, tal y como se detalla en secciones posteriores, éste es un parámetro configurable. Sin embargo, los tipos de los mensajes sí son relevantes ya que constituyen la composición de los mensajes a los que *Darknet ROS 3D* tendrá que subscribirse y los que tendrá que publicar.

Entradas al Sistema *Darknet ROS 3D*

Darknet ROS 3D requiere de dos entradas, obtenidas a través de topics, que son:

- **Bounding Boxes:** se trata de mensajes de tipo *darknet_ros_msgs/BoundingBoxes* que proporciona el paquete *darknet_ros*. Estos son los mensajes que contienen la información relativa a los bounding boxes calculados por YOLO para cada frame de la cámara. Cada bounding box se corresponde con un objeto detectado en dicho frame. Puede haber frames que contengan más de un bounding box y puede haber frames que no contengan ninguno en el caso de no detectarse ningún objeto reconocible por la red neuronal. Este tipo de mensajes contienen tres campos: Dos de ellos de tipo *std_msgs/Header* y el último de tipo *darknet_ros_msgs/BoundingBox[]*.

  El tipo *Header*, contiene tres campos que son:

  - **uint32 seq:** número de secuencia representado como un entero de 4 bytes sin signo. El número de secuencia suele tener por objetivo dotar al mensaje de un identificador único e incremental. De modo que el primer mensaje tendría un $seq = 0$; siendo los mensajes posteriores $seq = 1, 2, 3...n$; para n mensajes.
  - **time stamp:** contiene el timestamp del mensaje. Se trata de una unidad temporal que indica el momento exacto en el que el mensaje se publicó.
  - **string frame_id:** contiene el **frame** de referencia que se usó para generar este mensaje.

    En ROS, un frame es un sistema de coordenadas que podemos tomar como referencia para ciertos cálculos matemáticos de índole geométrica. Por ejemplo, para calcular las coordenadas de la posición de un objeto, es necesario determinar previamente con respecto a qué coordenadas se va a calcular. Este tipo de cálculos geométricos son vitales en robótica. Para ejemplificar su uso, pensemos en una persona que quiere coger con su mano una botella. Las personas somos capaces de detectar y reconocer objetos con la vista. Sin embargo, la distancia desde los ojos a la botella no es la misma que desde la mano a la botella. De hecho, a medida que el brazo se va extendiendo hacia la botella, su distancia irá variando, mientras que la distancia de los ojos a

la botella nunca cambia a no ser que la persona mueva la cabeza o se desplace. En este caso concreto, podríamos tener dos sistemas de coordenadas de referencia (con respecto a los ojos, o con respecto a la mano). Del mismo modo, conociendo ambas referencias (frames) se pueden realizar **operaciones de transformación** sobre las medidas tomadas en un frame para obtener esas mismas medidas con respecto a otro frame. A estas operaciones matemáticas, basadas en matrices, se les denomina en ROS, **transformadas**. Es decir, conociendo la distancia de la botella con respecto a los ojos y conociendo también la distancia entre los ojos y la mano, puede determinarse mediante una operación matemática cuál será la distancia de la botella con respecto a la mano. Esto sería una transformada entre el frame *ojos* y el frame *mano*.

Este tipo de operaciones, los humanos las llevamos a cabo en el cerebro continuamente de forma inconsciente.

Los dos primeros campos, de tipo header, se denominan *header* e *image_header* y se corresponden con las cabeceras del mensaje y de la imagen a partir de la cual se constituyó el mensaje, respectivamente.

Puede parecer redundante, pero no lo es. *darknet_ros* recibe una imagen de la cámara, calcula los bounding boxes, compone el mensaje y, por último, lo publica. Este proceso se realiza iterativamente. Por tanto, los campos de la cabecera (frame, número de secuencia y timestamp) pueden diferir. En el caso del timestamp, por ejemplo, es obvio que desde que se capturó el fotograma por parte de la cámara hasta que *darknet_ros* lo recibió mediante un topic, calculó los bounding boxes y constituyó el mensaje de tipo *darknet_ros_msgs/BoundingBoxes* pasó un tiempo, y por consiguiente, ese timestamp variará.

El último campo, se trata de un array de tipo *darknet_ros_msgs/BoundingBox*, el cual a su vez contiene la siguiente estructura:

- **float64 probability:** certeza en la detección. Se trata de una probabilidad comprendida entre 0 y 1.
- **int64 xmin:** píxel menor del eje X de la imagen correspondiente al bounding box.
- **int64 ymin:** píxel menor del eje Y de la imagen correspondiente al bounding box.
- **int64 xmax:** píxel mayor del eje X de la imagen correspondiente al bounding box.
- **int64 ymax:** píxel mayor del eje Y de la imagen correspondiente al bounding box.
- **int16 id:** identificador del bounding box.
- **string Class:** clase de objeto.

Nótese que los campos (xmin, ymin) forman las coordenadas (en píxeles) de la esquina superior izquierda del bounding box; y los campos (xmax, ymax), las

coordenadas de la esquina inferior derecha.

A continuación, a modo resumen, se muestra la salida del comando *rosmsg* para los tres tipos de mensajes donde se pueden apreciar todos los campos que posee cada uno:

```
fernando@ubuntu-ssd:~$ rosmsg show std_msgs/Header
uint32 seq
time stamp
string frame_id

fernando@ubuntu-ssd:~$ rosmsg show darknet_ros_msgs/BoundingBox -r
float64 probability
int64 xmin
int64 ymin
int64 xmax
int64 ymax
int16 id
string Class

fernando@ubuntu-ssd:~$ rosmsg show darknet_ros_msgs/BoundingBoxes -r
Header header
Header image_header
BoundingBox[] bounding_boxes

fernando@ubuntu-ssd:~$
```

Figura 4.3: Tipos de Mensaje

- **Point Cloud:** Mensaje de tipo *sensor_msgs/PointCloud2* que codifica la nube de puntos (point cloud) de la cámara para así albergar la información relativa a las coordenadas espaciales de cada píxel, así como de otros parámetros. Toda esta información se codifica en un array de bytes de longitud $width \cdot height$, siendo éstas las dimensiones de la imagen.

Salidas del Sistema Darknet ROS 3D

El resultado de los cálculos llevados a cabo para cada una de las entradas por parte de *Darknet ROS 3D* se publican en dos topics. Estos dos topics conforman las salidas del sistema.

Tal y como veremos a continuación, uno de los topics se corresponde con la información misma obtenida a partir de los datos de entrada, mientras que el otro alberga mensajes que contienen una representación gráfica de dicha información destinada a su visualización por parte del usuario de la herramienta.

El tipo de mensaje *gb_visual_detection_msgs/BoundingBoxes3d*, contiene una cabecera del tipo *std_msgs/Header*, con todos los datos anteriormente mencionados, y un array de bounding boxes tridimensionales del tipo *gb_visual_detection_msgs/BoundingBox3d*.

Este tipo de mensaje caracteriza un bounding box tridimensional, cuya diferencia con respecto a los bounding boxes mencionados anteriormente es que contiene información de profundidad.

Los bounding boxes tradicionales, por tanto, son contornos rectangulares definidos por las coordenadas de sus vértices (en píxeles), de modo que quedan representadas sus dimensiones en coordenadas bidimensionales (x, y).

Sin embargo, en estos nuevos bounding boxes tridimensionales, se pretende contener información de profundidad. Por este motivo, la unidad de medida de las coordenadas ya no son los píxeles de una imagen, sino la unidad de distancia del Sistema Internacional: el metro. Además, es necesario introducir las coordenadas de un vértice más para así definir los ejes de coordenadas (x, y, z). Por tanto, estos nuevos bounding boxes contendrán los puntos de coordenadas $(\mathbf{x_{min}}, \mathbf{y_{min}}, \mathbf{z_{min}})$ y $(\mathbf{x_{max}}, \mathbf{y_{max}}, \mathbf{z_{max}})$.

Por tanto, los nuevos mensajes que Darknet ROS 3D proporciona como salida albergan una estructura similar a los de *darknet_ros* pero con la información necesaria para caracterizar los bounding boxes tridimensionales. A continuación se muestra la salida del comando *rosmsg* para este tipo de mensajes:

```
fernando@ubuntu-ssd:~$ rosmsg show gb_visual_detection_3d_msgs/BoundingBoxes3d
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
gb_visual_detection_3d_msgs/BoundingBox3d[] bounding_boxes
  string Class
  float64 probability
  float64 xmin
  float64 ymin
  float64 xmax
  float64 ymax
  float64 zmin
  float64 zmax
```

Figura 4.4: Bounding Boxes 3d Messages

Como se puede apreciar, el tipo de mensaje *gb_visual_detection_3d_msgs/BoundingBox3d*, caracteriza a un único bounding box tridimensional. En su interior se encuentran los campos ya conocidos *Class* y *probability* y, posteriormente, cada uno de los componentes que forman los vértices de un prisma rectangular, es decir, (x, y, z) mínimos y máximos.

Este tipo de mensaje, tal y como ya se ha mencionado, es muy similar a los proporcionados por *darknet_ros* con la salvedad de que las coordenadas ya no se expresan en píxeles, sino en metros y se añade una coordenada más, necesaria para definir los tres vértices que caracterizan la posición y dimensiones de un prisma en el espacio

4.2.3. Estructura del Software

A continuación, y una vez finalizado el análisis del paquete *Darknet ROS 3D* a nivel de sistema, se estudiará cuál es su estructura desde el punto de vista del desarrollo del software. Se profundizará, por tanto, en el árbol de directorios que contienen el código fuente del software y cómo se relacionan entre sí los distintos ficheros.

Todo el contenido del paquete ROS se encuentra en un directorio denominado *darknet_ros_3d*. Este directorio, contiene dos ficheros fundamentales que, además, son

necesarios para que ROS sea capaz de tener en cuenta este directorio como un paquete software de ROS. Estos dos ficheros se detallan a continuación:

- **package.xml**: En este fichero se detallan algunos metadatos relativos al paquete ROS. Los más habituales son el creador, el mantenedor, la versión del software, la licencia del mismo y una breve descripción de su funcionalidad. Además, se detallan algunos parámetros relevantes para la compilación del paquete y su posterior integración dentro del ecosistema de ROS: Las dependencias.
- **CMakeLists.txt**: En este fichero se detallan las reglas de compilación y enlazado de cada una de las librerías que contenga el paquete, si las contuviera, así como de cada uno de los ejecutables que finalmente se quieran obtener.

Como se dijo previamente, la existencia de estos dos ficheros son esenciales para que un paquete ROS pueda ser considerado como tal y, por tanto, poderse compilar y ser integrado dentro del workspace junto al resto de paquetes.

El resto de la estructura del paquete no presenta diferencias destacables con respecto a la morfología habitual de los paquetes ROS. En la figura 4.5 se muestra el árbol de directorios que contiene.

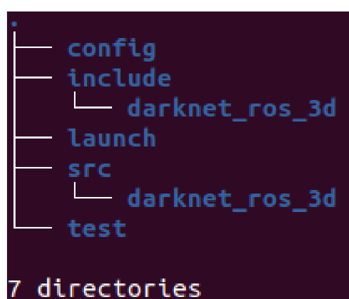


Figura 4.5: Estructura de Directorios

Los directorios fundamentales que componen este paquete se pueden clasificar teniendo en cuenta su funcionalidad o razón de ser. De este modo, tenemos 3 categorías: **código fuente**, **configuración** y **ejecución**.

Código Fuente

El código fuente, completamente desarrollado mediante el lenguaje de programación C++, se encuentra dividido en una librería específica y un programa principal que hace uso de la misma. Además, esta librería se encuentra a su vez dividida en fichero de cabeceras (localizado en el directorio *include*) y fichero de implementación (localizado en el directorio *src*).

Tal y como se aprecia en la figura 4.6, el programa principal lo compone el fichero **darknet3d_node.cpp**. Este programa contiene la estructura principal del nodo ROS que se encargará de realizar todas las tareas.

En primer lugar, inicializa el nodo, de modo que el ROS Master tenga constancia del nuevo nodo que formará parte del grafo de computación, que se llamará *darknet_3d*.

A continuación se declara un objeto de la clase principal de la librería **Darknet3D.h**. Por medio de este objeto, se podrá acceder al único método público que se encuentra definido en dicha clase. Se trata del método *update* y se le llama de manera iterativa, con una tasa de iteración establecida como constante (es decir, inmutable) a 10 hercios.

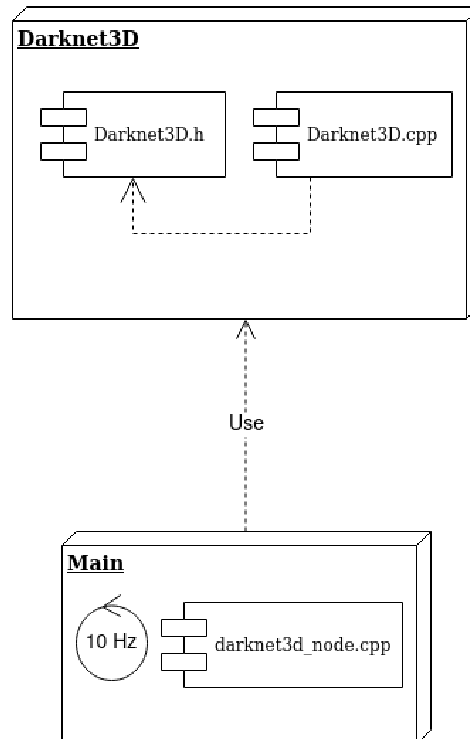


Figura 4.6: Diagrama de Componentes

Los nodos ROS se pueden programar bajo dos paradigmas distintos:

- **Orientado a Eventos:** Existen unas funciones denominadas *de callback* que se ejecutan cuando cierto evento ocurre, de manera que toda la lógica que desee realizarse debería estar contenida dentro de dichas funciones de callback.
- **Nodos Iterativos:** La función o conjunto de funciones que contienen la lógica del nodo son ejecutadas dentro de un bucle de manera iterativa.

Darknet ROS 3D está programado de un modo iterativo ya que de este modo se puede controlar la **tasa de iteración**, es decir, a qué frecuencia se quiere ejecutar la lógica del programa. Bajo el paradigma de orientación a eventos, no sería posible controlar este aspecto debido a que cada vez que llegue un evento nuevo (por ejemplo un nuevo mensaje por el topic al que el nodo *darknet_3d* se subscribe) se llevará a cabo la lógica del programa. No se puede predecir cuándo llegará un nuevo evento y, por lo tanto, no se puede establecer una tasa de procesamiento. En este caso, ya que se realizan operaciones sobre imágenes y que, por tanto, pueden contener cálculos matemáticos complejos, no poder controlar la tasa de iteración del nodo ROS es algo indeseable que puede llegar a ocasionar una sobrecarga por exceso de tiempo de cómputo en la máquina donde se está ejecutando y dar lugar, por tanto, a una pérdida de rendimiento del sistema.

Configuración

La primera tarea que realiza el nodo por medio de la librería *Darknet3D*, es llevar a cabo su configuración. La librería, carga un fichero en formato **yaml** que contiene todos los parámetros que el nodo necesita para configurarse. Estos parámetros son usados posteriormente en distintas partes del código como parte de su algoritmia. Los parámetros son los siguientes:

- **darknet_ros_topic:** Es el topic de salida de *darknet_ros*. Este es el topic al que se subscribe *darknet_ros_3d* para obtener los bounding boxes de entrada.
- **output_bbx3d_topic:** Este parámetro define cuál será el topic de salida de los bounding boxes tridimensionales calculados.
- **point_cloud_topic:** Define el topic al que el nodo se tiene que subscribir para obtener la información de la nube de puntos de la cámara RGBD. Constituye la segunda entrada que el sistema necesita.
- **working_frame:** Constituye el frame con respecto al cual se realizarán los cálculos de las coordenadas de los bounding boxes.
- **minimum_detection_threshold:** Se trata de un parámetro específico para el algoritmo encargado de calcular los bounding boxes. Se detallará su uso en secciones posteriores.
- **minimum_probability:** Probabilidad mínima a partir de la cual se tienen en cuenta los objetos obtenidos por el topic de entrada definido en el parámetro *darknet_ros_topic*. Cualquier detección cuya probabilidad sea menor, no se tendrá en cuenta en el algoritmo.
- **interested_classes:** Se trata de una lista que contiene todas las clases de objetos a considerar. Las detecciones de entrada cuyas clases sean distintas, no se tienen en cuenta en el algoritmo.

Cada uno de estos parámetros tiene un valor por defecto. Sin embargo, el usuario de este paquete ROS debe ser el encargado de adecuar cada uno de los parámetros de configuración a su entorno y/o sus necesidades. Se trata de una forma de extender la compatibilidad y polivalencia del software al mayor número de entornos posible, de manera que el usuario que haga uso del paquete, no necesite modificar el código de programación del software para poder utilizarlo.

Si bien es cierto que esto es algo que podría hacerse sin ningún tipo de problema debido a la licencia de Open Source, lo que se persigue es que el usuario no necesite poseer los conocimientos técnicos necesarios para poder abordar una modificación del software. A través de los parámetros de configuración, un usuario debería ser capaz de individualizar el paquete ROS para el entorno concreto en el que se vaya a ejecutar.

Ejecución

Para conseguir poner en funcionamiento el sistema completo representado en la figura 4.2 es necesario ejecutar, por este orden, el driver de la cámara que se quiera

usar, el nodo de *darknet_ros* y el nodo de *Darknet ROS 3D*. Cada uno de ellos de forma independiente.

Para evitar tener que ejecutar cada uno de los programas manualmente, *Darknet ROS 3D* contiene en el directorio **launch** un fichero que permite ejecutar Darknet ROS y Darknet ROS 3D simultáneamente.

Se trata de un tipo de fichero con extensión *.launch* que admite el framework ROS y que funciona a modo de script, de modo que se pueden indicar múltiples nodos a ejecutar, parámetros de entrada para cada uno de ellos, etc. Incluso pueden incluirse estos ficheros *launch* dentro de otros de modo que sea más legible, mantenible y escalable.

Gracias a este fichero denominado *darknet_ros3d.launch*, se puede ejecutar Darknet ROS y Darknet ROS 3D con la única instrucción *roslaunch darknet_ros_3d darknet_ros3d.launch*.

Se trata del comando *roslaunch* y recibe dos parámetros de entrada: el nombre del paquete ROS donde se encuentra el fichero launch, y el nombre del fichero. ROS automáticamente localiza el paquete dentro del workspace e interpreta el fichero launch, el cual tiene un formato de xml.

4.2.4. Algoritmia

Por una parte, el nodo se subscribe al point cloud de la cámara RGBD y a los bounding boxes de Darknet ROS. Se empiezan a recibir sendos mensajes de forma asíncrona. Para cada uno de estos mensajes que se reciben, lo único que se hace es guardarlos en dos atributos distintos, de modo que siempre se guardará el último mensaje recibido por cada uno de los topics. Adicionalmente, para cada bounding box que se recibe, se guarda en otra variable el timestamp (tiempo en milisegundos) en el que se recibió.

Cada una de las tareas se realiza de manera asíncrona tal y como ilustra la siguiente figura:

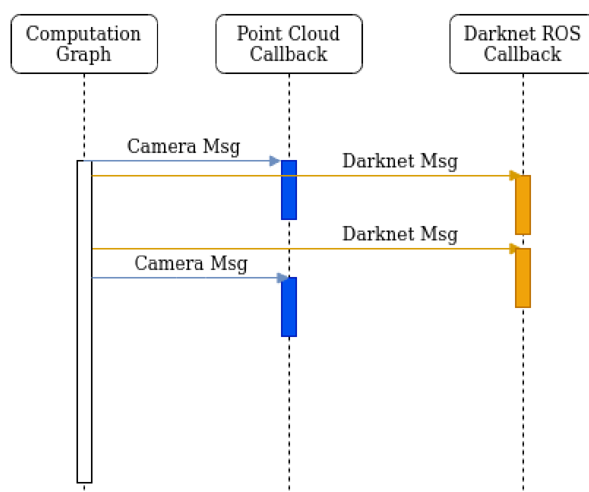


Figura 4.7: Secuencia Ejecución de Callbacks

En primer lugar, se define lo que se denomina una función de **callback** para cada tipo de mensaje a cuyo topic nos hemos subscrito. En este caso, como nos subscribimos al topic de la cámara y al topic de Darknet ROS, se recibirán dos tipos de mensaje distintos. Concretamente, el primero será de tipo *sensor_msgs::PointCloud2*; y el segundo, de tipo *darknet_ros_msgs::BoundingBoxes*.

Estas funciones de callback no son más que manejadores, es decir, son las funciones que encapsulan la lógica que se quiere llevar a cabo cada vez que se reciba un mensaje por cada uno de los topics.

En la figura 4.7 se muestra el flujo de ejecución de las distintas funciones de callback.

La tasa a la que se reciben los mensajes por ambos topics dependerá de la frecuencia a la que ambos publicadores publiquen dichos mensajes. Sea como fuere, nunca se recibirán mensajes a una tasa mayor de 10 Hz, ya que el nodo es iterativo, tal y como se mencionó en la sección 4.2.3. Esto significa que no sólo se procesan los mensajes a una tasa dada sino que también se reciben, como mucho, a esa tasa.

Con este escenario, pueden suceder dos cosas: Si los nodos publicadores publican a una tasa más baja, lo único que sucederá es que se procesará el último mensaje que se haya recibido hasta que se reciba uno nuevo. Si por el contrario, publican a una tasa más elevada, se recibirá cada vez el último mensaje que se haya publicado.

En el extracto de código 4.1 se muestra la implementación de este algoritmo iterativo.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "darknet_3d");
  darknet_ros_3d::Darknet3D darknet3d;

  ros::Rate loop_rate(10);

  while (ros::ok())
  {
    darknet3d.update();

    ros::spinOnce();
    loop_rate.sleep();
  }

  return 0;
}
```

Extracto de código 4.1: Algoritmo Iterativo del nodo Darknet 3D

Tal y como se explicó en secciones anteriores, se inicializa el nodo, se establece la frecuencia de iteración a 10 Hz y se va ejecutando iterativamente el método *update* de la librería *Darknet3D*.

Sin embargo, llegados a este punto, podemos observar que dentro del bucle existe la siguiente instrucción: **ros::spinOnce()**.

Esta función, pertenece a la librería *roscpp*, que es la que proporciona al programador de una interfaz (conjunto de funciones, tipos, etc) para interactuar con el framework ROS. Lo que *spinOnce()* permite es procesar la cola de mensajes. ¿Qué significa esto?

Como sabemos, todos los nodos del grafo de computación dependen del nodo Master, quien se encarga de establecer las comunicaciones entre publicadores y subscriptores por medio de los topics. Siguiendo este esquema de funcionamiento, cuando un mensaje se publica, queda en cola hasta que sea procesado. Cada nodo subscriptor posee una cola de mensajes a partir de la cual puede ir procesándolos. La librería *roscpp* permite definir cuál queremos que sea el tamaño de la cola. En este caso, es 1. De esta manera siempre se procesará el último mensaje recibido.

Por tanto, la tarea de la función *spinOnce()* es la de leer el siguiente mensaje que haya en la cola. Podemos decir entonces que la llamada esta función es la que propicia la ejecución de las funciones de callback (en el caso de que haya un mensaje que procesar) representadas en la figura 4.7 y cuyo contenido se muestra en los siguientes extractos de código:

```
void
Darknet3D::pointCloudCb(const sensor_msgs::PointCloud2::
   ConstPtr& msg)
{
  point_cloud_ = *msg;
}
```

Extracto de código 4.2: Callback del Point Cloud

```
void
Darknet3D::darknetCb(const darknet_ros_msgs::BoundingBoxes::
   ConstPtr& msg)
{
  last_detection_ts_ = ros::Time::now();
  original_bboxes_ = msg->bounding_boxes;
}
```

Extracto de código 4.3: Callback de los Bounding Boxes

Lo más llamativo de ambos callbacks es que realizan tareas muy sencillas y concretas. No es deseable llevar a cabo tareas computacionalmente pesadas en las funciones de callback, ya que esto ralentiza su ejecución y puede romper la iteratividad del programa. Por tanto, lo que se hace es guardar los mensajes recibidos en las variables *point_cloud_* y *original_bboxes_* para su posterior ejecución en cada una de las llamadas al método *update*.

Este método contiene todo el código relativo al procesamiento de los mensajes, incluyendo la publicación de los mensajes de salida.

Para facilitar la comprensión de las tareas realizadas por el método *update*, método que, tal y como ya se ha explicado, constituye el núcleo de este nodo; se proporciona un diagrama de flujo representado en la figura 4.8 que se muestra a continuación.

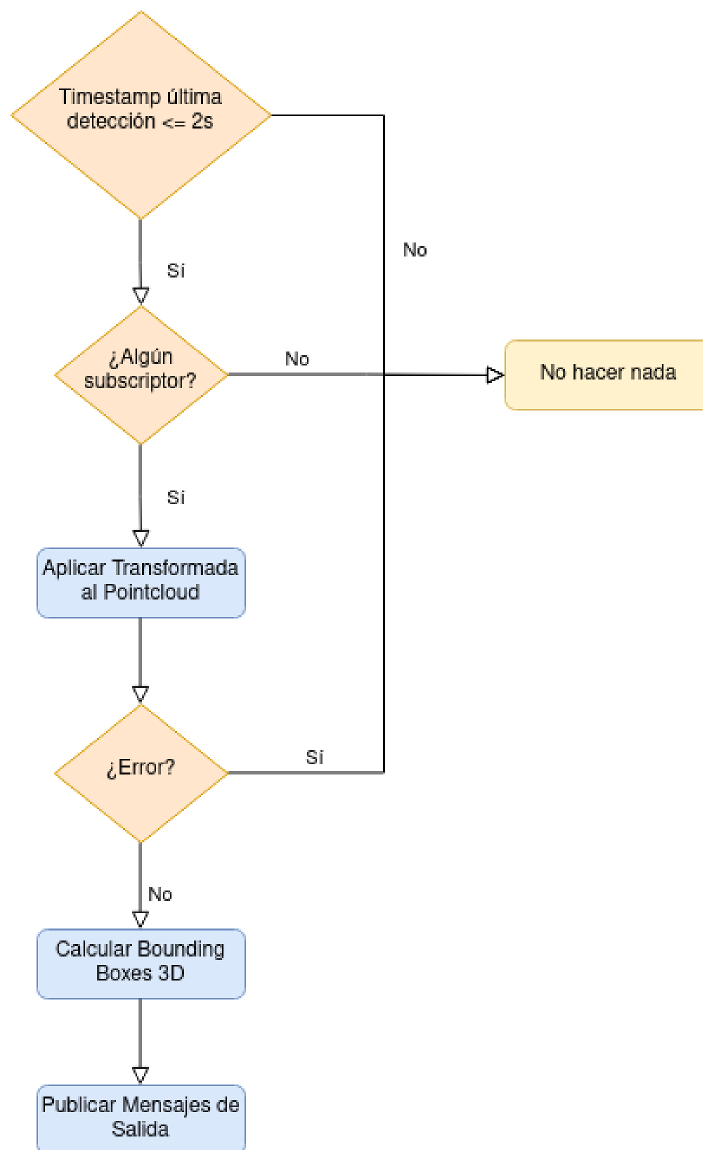


Figura 4.8: Diagrama de Flujo Método Update

Tal y como vemos, lo primero que se hace es comprobar si la última detección (bounding box publicado por Darknet ROS) se obtuvo hace dos segundos o menos. En el caso de que la detección se recibiera hace más de dos segundos, no se realiza ningún procesamiento.

Del mismo modo, si no existe ningún nodo que se encuentre en ese momento subscrito a ninguno de los dos topics de salida de Darknet ROS 3D, no se realiza ningún procesamiento.

Si existe algún subscriptor, se comienzan a realizar las tareas necesarias para el cálculo de los bounding boxes 3D y la publicación de los mensajes de salida en tres fases que se desarrollan en las siguientes secciones.

Aplicar Transformada al Point Cloud

Tal y como se menciona en la sección 4.2.2, antes de realizar ningún tipo de cálculo, es necesario establecer un sistema de coordenadas (frame) de referencia. En este caso, las coordenadas de referencia vienen dadas por el parámetro de configuración *working_frame*. Comúnmente lo deseable es que el sistema de coordenadas de referencia lo establezca la transformada (TF) de la cámara. Por este motivo, por defecto el working frame es *camera_link*, aunque esto puede variar en función del driver de la cámara utilizado.

Dado que el sistema de coordenadas de referencia viene dado por el working frame, la primera tarea es aplicar una transformada al pointcloud para que de ese modo, todas las medidas de distancia estén referenciadas con respecto a la referencia deseada (en este caso, *camera_link*).

Para llevar a cabo esta transformada, se utiliza la librería **PCL** (*Point Cloud Library*). Se trata de una librería ampliamente conocida por su utilidad para el manejo de imágenes 2D y 3D.

La función que se utiliza, *pcl_ros::transformPointCloud*, está definida en el extracto de código 4.4. Esta función permite obtener en *out*, el Point Cloud calculado (vía transformada) en el frame *target_frame* a partir del Point Cloud de entrada *in* y un objeto de tipo *tf::TransformListener* necesario para realizar dicha transformada.

```
bool pcl_ros::transformPointCloud(
            const std::string &                 target_frame,
            const sensor_msgs::PointCloud2 &    in,
            sensor_msgs::PointCloud2 &          out,
            const tf::TransformListener &       tf_listener
            );
```

Extracto de código 4.4: Transformada del Point Cloud

Al término de la ejecución de esta función y, si su ejecución fue satisfactoria, tendremos un objeto de tipo *sensor_msgs::PointCloud2* que almacena el Point Cloud con respecto al working frame. El Point Cloud que se usa como entrada (parámetro *in*) es la variable actualizada en la función de callback relativa a la subscripción al topic del Point Cloud de la cámara RGBD (representación gráfica en la figura 4.7) denominada *pointCloudCb*.

El tipo de mensaje *PointCloud2* contiene toda la información relativa al color de los píxeles (rgb), distancia, intensidad, etc, codificada en un array unidimensional. Sin embargo, para poder tratar con la información de interés, es decir, las coordenadas XYZ, se hace uso de nuevo de la librería PCL (Point Cloud Library) para decodificar toda esta información y almacenarla en una estructura más sencilla de manejar. Esto se lleva a cabo tal y como se aprecia en el segmento de código 4.5.

Se declara la variable *pcrgb* que contendrá los datos decodificados del point cloud por la librecía PCL. La variable *local_pointcloud* es el point cloud previamente transformado al frame de referencia.

La función *pcl::fromRosMsg* se encarga de configurar apropiadamente la variable *pcrgb*, la cual contiene información relativa al color rgb de cada píxel y también de sus coordenadas XYZ.

```
pcl::PointCloud<pcl::PointXYZRGB>::Ptr pcrgb(
    new pcl::PointCloud<pcl::PointXYZRGB>);

pcl::fromROSMsg(local_pointcloud, *pcrgb);
```

Extracto de código 4.5: Point Cloud Decodificado

Una vez obtenido el point cloud en el frame de referencia y habiendo tratado los datos con la librería PCL con la finalidad de que facilitar el manejo de los mismos, se lleva a cabo el cálculo de los bounding boxes tridimensionales.

Cálculo de Bounding Boxes 3D

La construcción de los bounding boxes se basa en un algoritmo iterativo en el que se recorren cada una de las clases detectadas en el último bounding box obtenido de Darknet ROS. Sin embargo, sólo tendremos en cuenta para el cálculo de los bounding boxes aquellas clases definidas en el parámetro de configuración *interested_classes* mencionado en la sección 4.2.3.

Para comenzar, se obtiene el índice correspondiente al píxel central del bounding box, ya que se asume que en un alto porcentaje de los casos, este píxel corresponde al centro del objeto detectado. Con este índice, se calcula la posición que ocupa ese mismo píxel en el point cloud. De este modo, se obtiene el punto concreto, perteneciente a la nube de puntos, que se corresponde con el píxel central del bounding box. Ésta será la **semilla**, es decir, el punto de referencia para poder determinar si el resto de puntos del point cloud pertenecen o no al objeto detectado.

Una vez obtenida esta semilla, se recorren cada uno de los píxeles del bounding box y se obtiene su correspondiente índice en el point cloud. Si la diferencia en profundidad (coordenada X) es menor que el parámetro de configuración *mininum_detection_threshold*, se considera un píxel perteneciente al objeto de interés y se actualizan las coordenadas (Xmin, Ymin, Zmin) y (Xmax, Ymax, Zmax).

Una vez que se han recorrido todos los píxeles del bounding box y que, por tanto, las coordenadas que determinarán el bounding box 3D están actualizadas, se compone el mesaje de tipo *gb_visual_detection_3d_msgs/BoundingBox3d* que será publicado.

Publicación de Mensajes de Salida

Tal y como se explicó anteriormente y se encuentra representado en la figura 4.2, el nodo de Darknet ROS 3D publica mensajes de salida en dos topics diferentes. Uno de ellos contienen los bounding boxes 3D calculados. Estos mensajes contienen los datos principales de salida generados por el subsistema.

Sin embargo, el topic denominado */darkent_ros_3d/markers* contiene mensajes de visualización que permiten representar, de un modo gráfico, la información contenida

en el topic anterior, haciendo uso de la herramienta de monitorización y depuración integrada RViZ.

En al figura 4.9, se encuentra un ejemplo de **Visual Marker** representado gráficamente en RViZ.

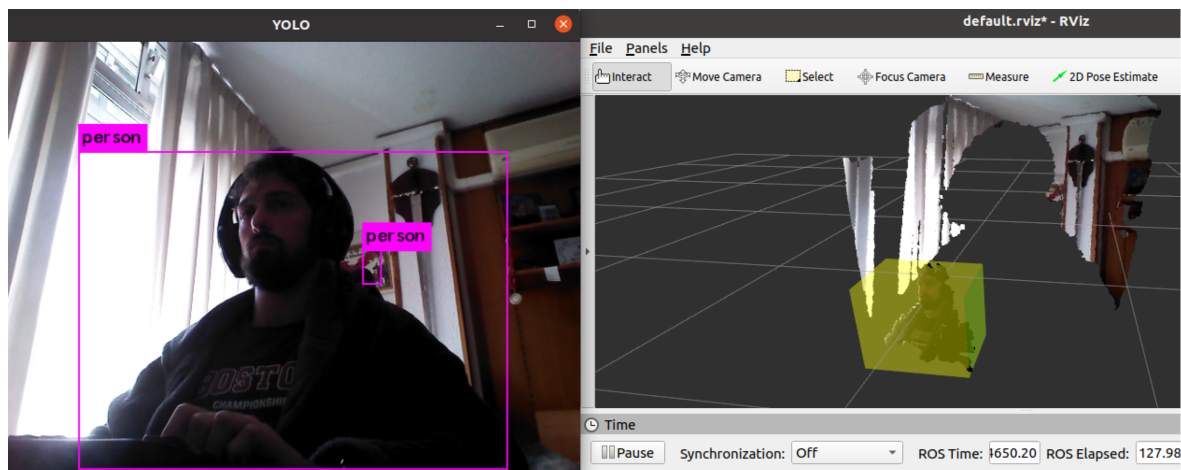


Figura 4.9: Visual Markers

Lo que se observa en el lado izquierdo de la figura es la detección de *Darknet ROS* (utilizando la red neuronal *YOLO*). En ella podemos ver dos bounding boxes de la clase *person*.

En el lado derecho, se encuentra el programa de monitorización y depuración RViZ (integrado en el meta sistema operativo ROS). Lo que se está mostrado es el point cloud de la cámara 3D y los *visual markers* publicados por Darknet ROS 3D. El resultado de mostrar los mensajes de ambos topics simultáneamente es que vemos un prisma sobre el point cloud. Este prisma, envuelve precisamente a la detección de la clase *person* obtenida por YOLO.

Lo que está sucediendo en este ejemplo es que Darknet ROS 3D está realizando los cálculos explicados anteriormente a partir de la información proporcionada por Darknet ROS. Tras calcular los bounding boxes 3D, los publica en un topic (el cual no se visualiza) destinado a que cualquier otro nodo ROS pueda hacer uso de esa información.

Adicionalmente, publica en otro topic *visual markers* (uno por cada bounding box 3D calculado) destinado a su visualización en RViZ.

El ejemplo de la figura 4.9 resulta muy ilustrativo, ya que se aprecia un segundo bounding box en el fondo de la imagen obtenida por YOLO, sin emabargo no se aprecia un *visual marker* en RViZ correspondiente a ese bounding box. El motivo de que esto suceda es que en el ejemplo se ha utilizado un valor de 0.7 en el parámetro de configuración *minimum_probability*. Es decir, Darknet ROS 3D sólo tendrá en cuenta aquellos bounding boxes que contengan probabilidades mayores o igual a 0.7. de este modo, se puede filtrar detecciones espúreas.

5. Evaluación y Despliegue

Darknet ROS 3D es una herramienta basada en la **filosofía UNIX**. Esta filosofía se trata de hacer multitud de programas que hagan cosas muy sencillas pero que, precisamente por ser sencillas, son muy eficientes. El resultado de la interconexión de varios programas de estas características es un sistema capaz de llevar a cabo tareas complejas.

Esto significa que Darknet ROS 3D no es un fin en sí mismo sino que proporciona un conjunto de datos que pueden ser tomados por cualquier otro programa (un nodo ROS, por ejemplo) para llevar a cabo tareas más complejas.

Desde este punto de vista, la eficiencia del paquete ROS, así como la precisión en los datos obtenidos dependen de distintos factores.

A lo largo de este capítulo se abordarán diferentes **plataformas robóticas** en las que se ha desplegado Darknet ROS 3D, así como las diferentes **plataformas de computación**. Posteriormente, se detalla el tipo de entornos en los que el robot tuvo que operar.

5.1. Plataformas Robóticas

Para el despliegue de la herramienta Darknet ROS 3D, se han utilizado dos plataformas robóticas principalmente: el robot **Turtlebot 2** y el robot **Tiago** representadas en la figura 5.1. Ambos son robots comerciales.

Ambos robots se basan en una plataforma móvil con dos ruedas, por lo que tienen una mecánica de movimiento diferencial, es decir, ambas ruedas giran 360 grados y en los dos sentidos, permitiendo a los robots girar sobre sí mismos o trazar trayectorias curvas regulando la velocidad y/o sentido de giro de cada una de las ruedas.

(a) Turtlebot 2

(b) Tiago

Figura 5.1: Plataformas Robóticas Utilizadas

En cuanto a sensores, la única diferencia sustancial entre ambos robots es que el Tiago posee una cámara RGBD modelo ASUS Xtion como la que se representa en la figura 1.9 incorporada en la cabeza (a modo de ojos) y el Turtlebot 2 no. En su defecto, se utilizó una cámara modelo Orbecc Astra.

La arquitectura software utilizada para ambos casos, la cual se encuentra representada en la figura 5.2, no difiere a la que usaría sobre prácticamente cualquier otra plataforma robótica.

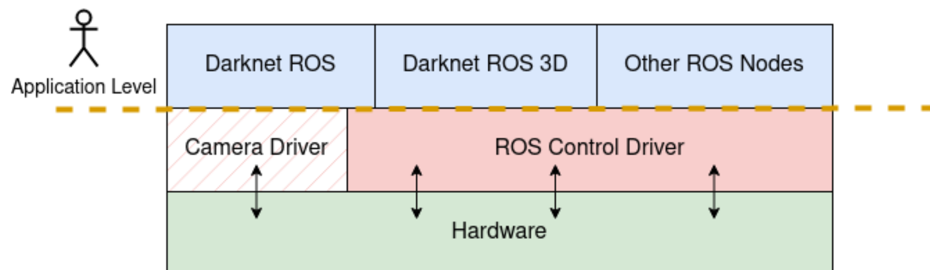


Figura 5.2: Arquitectura Software sobre Plataforma Robótica

Analizando esta arquitectura, vemos que en el nivel más bajo se encuentra el hardware. Este nivel representa toda la capa física del sistema (sensores, actuadores, placas electrónicas, etc...). En el nivel superior se encuentran los drivers. Si se trata de una plataforma robótica comercial, lo común es que el propio fabricante o la comunidad

de desarrolladores, proporcionen un driver específico para dicha plataforma. En el caso del Tiago, la cámara se encuentra incluida en el propio robot, sin embargo, en el caso del Turtlebot, se requiere un driver específico para la cámara puesto que se trata de un sensor añadido que no forma parte de la plataforma y por tanto de su driver.

Estos drivers proporciona una **interfaz** entre cualquier aplicación ROS y el robot. Es decir, nos permite acceder a todos los sensores y actuadores del robot mediante los mecanismos que ROS proporciona (por ejemplo, topics).

Por encima de este nivel de abstracción, tenemos el nivel de aplicación. En este nivel estarían localizados todos los nodos ROS que constituyen la aplicación robótica de alto nivel (algoritmos, estructuras de control, etc). Es en este nivel en el que se encuadra Darknet ROS 3D, así como otro tipo de nodos, como Darknet ROS, que constituye una dependencia del nodo anterior. Además, otras aplicaciones que puedan utilizar la información que Darknet ROS 3D proporciona, se ejecutan a este nivel.

5.2. Plataformas de Computación

5.2.1. Computadores de Escritorio

Además de un robot, para el despliegue de Darknet ROS 3D, es necesaria una unidad de procesamiento que sea capaz de ejecutar el meta sistema operativo de ROS, sobre Linux, y tenga la potencia de cómputo suficiente para ejecutar la red neuronal de YOLO.

YOLO, mediante el paquete ROS que lo engloba (Darknet ROS), puede operar de dos modos distintos: con **CPU** o con **GPU**. Sin embargo, el rendimiento de la red neuronal utilizando exclusivamente la CPU, es muy bajo, con tasas de detección no superiores a 1 FPS (frame por segundo) sobre procesadores de escritorio tanto Intel como AMD.

Con estas tasas resulta imposible la realización de algoritmos que sean capaces de operar sobre entornos reales de manera eficiente y reactiva.

Sin embargo, cuando Darknet ROS se ejecuta haciendo uso de la GPU, su rendimiento mejora notablemente, llegando a tasas de 10 FPS sobre GPU Nvidia GT 750 2GB o 15 FPS sobre GPU Nvidia GTX 1060 6GB.

5.2.2. Computadores Embebidos

Montar un computador de escritorio sobre una plataforma robótica móvil es una tarea dificultosa por razones de peso, alimentación, espacio, etc. Por este motivo, normalmente se utilizan computadores de reducido tamaño como las plataformas embebidas mencionadas en el capitulo 2.1.

Se han utilizado dos plataformas embebidas para el despliegue de Darknet ROS 3D:

Nvidia Jetson Nano

Consta de una GPU Nvidia Maxwell de 128 núcleos y un procesador ARM de 4 núcleos. Además cuenta con una memoria RAM de 4GB. El sistema operativo (Ubuntu 18.04) reside en una tarjeta SD y el tamaño de la placa es de 69.6 mm x 45 mm.

Tal y como se aprecia en la figura 5.3, cuenta con puertos USB para acceso a periféricos (sensores o actuadores) y conector de red RJ-45.

Este computador se ha utilizado sobre la plataforma robótica Turtlebot 2 previamente mencionada.

Figura 5.3: Nvidia Jetson Nano

Nvidia Jetson Tx2

Se trata de una GPU similar a la anterior aunque con mayor potencia de cómputo. Consta de 256 núcleos de GPU y una memoria RAM de 8GB. Además, presenta una diferencia con respecto a la anterior: el sistema operativo reside en una memoria flash integrada en la tarjeta, por lo que no se precisa de tarjeta SD.

Figura 5.4: Nvidia Jetson Tx2

En este caso, la GPU se encuentra montada sobre una placa base (carrier) que ofrece diferentes puestos de entrada/salida de igual modo que la Nvidia Jetson Nano (USB, RJ-45, etc).

La Nvidia Jetson Tx2 se utilizó conjuntamente con el robot Tiago.

El rendimiento de Darkent ROS 3D sobre ambas plataformas es muy satisfactorio, consiguiendo tasas de mensajes lo suficientemente elevadas como para llevar a cabo diversas pruebas en entornos reales como las que se describen en la próxima sección.

5.3. Entorno

Además de las plataformas mencionadas en el capítulo anterior, la herramienta Darknet ROS 3D ha sido utilizada en entornos reales.

La primera vez que se puso a prueba la herramienta fue en 2018 durante la **RoboCup** de Trieste (Italia).

La RoboCup se trata de una competición de robótica de ámbito internacional en la cual diferentes equipos, cada uno con su robot, deben abordar un conjunto de pruebas. Se trata de poner a prueba las capacidades del robot mediante retos que tienen que ver con navegación, visión, manipulación y diálogo, o incluso un conjunto de todas ellas.

Darknet ROS 3D fue de vital importancia, pues en diversas pruebas era necesario reconocer objetos o seguir a una persona.

Posteriormente, la herramienta volvió a ser utilizada durante la RoboCup de Milton Keynes (Reino Unido).

5.4. Repercusión

Adicionalmente a la utilización ya mencionada de la herramienta Darknet ROS 3D, este paquete ROS ha sido ampliamente utilizado por personas de diversos países. Tanto alumnos de robótica de diferentes lugares como desarrolladores y miembros de la comunidad de ROS han hecho uso del mismo.

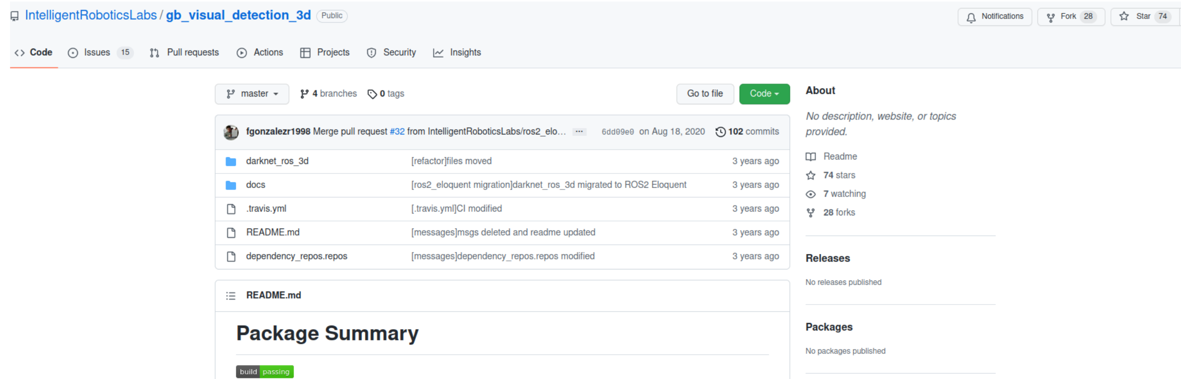


Figura 5.5: Repositorio Público de GitHub

Actualmente cuenta con más de 70 estrellas y 28 *forks* en Github, tal y como se aprecia en la figura 5.5 que muestra el repositorio de Github donde se encuentra disponible la herramienta Darknet ROS 3D.

6. Trabajos Futuros

A lo largo de este trabajo, se ha tratado de presentar y explicar una herramienta que fue desarrollada para dar solución a las diferentes pruebas de la competición *RoboCup* previamente mencionada. Sin embargo, se pretende que su utilidad sea mucho mayor en el futuro, ampliando las capacidades de este paquete ROS o desarrollando otros paquetes que sean capaz de integrarse con Darknet ROS 3D y que sean capaces de aportar valor a la robótica móvil.

En esta línea, lo que pretendo conseguir en un futuro son dos objetivos que describo a continuación:

6.1. Mapeo Semántico

A día de hoy, los mapas que suelen utilizarse para navegación autónoma están constituiodos por diversas celdas, tal y como se representa en la figura 1.10.

Esto permite que el robot sea capaz de desplazarse hacia unas coordenadas determinadas del mapa. Sin embargo, haciendo uso de Darknet ROS 3D, podríamos ser capaces de situar cada uno de los objetos detectados en el frame de referencia *map*, el cual se trata de un frame estático, es decir, que no varía nunca, permitiendo de ese modo situar los objetos dentro del mapa.

Una vez que los objetos de interés están situados en el mapa, podríamos ser capaces de hacer que el robot navegue hasta ellos. De este modo, a un robot se le podría decir algo como *acércate al frigorífico*, y que el robot (que ya conoce la posición exacta del frigorífico en el mapa) navegue hacia él de forma autónoma.

6.2. Estimación de Trayectorias

Se trata de poder estimar las trayectorias de cada uno de los objetos móviles detectados basándonos en las diferentes detecciones del mismo objeto obtenidas en cada instante de tiempo. Por ejemplo:

En el instante **t** se detecta una persona de dimensiones **w**, **h** y **d** (ancho, alto y profundidad), con centro en **c**.

En el instante **t + 1**, se detecta una persona de dimensiones similares y lo suficientemente cercana como para considerarla la misma persona que la de la detección anterior desplazada (se pueden proponer diversos algoritmos a tal efecto).

Sabiendo el tiempo transcurrido entre ambas detecciones y la distancia espacial entre una y otra, podrían calcularse las velocidades linear y angular del objeto móvil, lo que también permite realizar una estimación de la próxima posición donde se espera encontrar a dicho objeto. De este modo, se estimarían trayectorias.

El algoritmo que se usará para realizar todas las estimaciones es el denominado **Filtro Extendido de Kalman** (EKF), el cual se trata de un algoritmo probabilístico ampliamente conocido en diversos campos relacionados con la **teoría de control**. Hace uso de funciones de densidad de probabilidad Gaussianas para modelar el error cometido (bien sea debido a sensores ruidosos o a estimaciones imprecisas) con el fin de obtener predicciones con un alto grado de precisión.

En la figura 6.1 se representa un ejemplo de estimación de la posición de un objeto en movimiento.

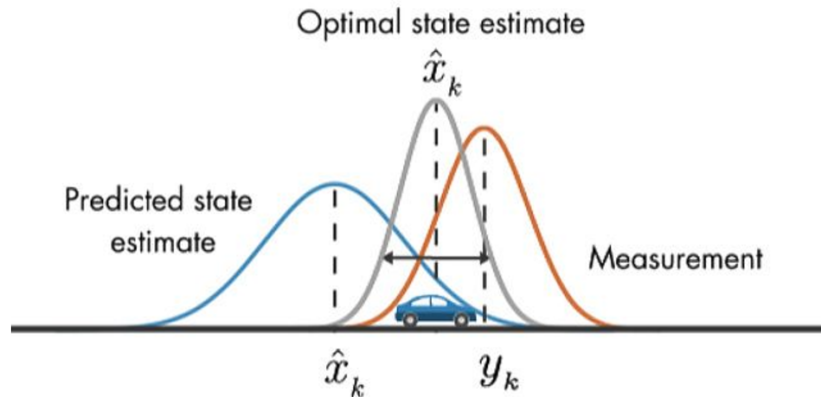


Figura 6.1: Ejemplo de uso Filtro Gaussiano

Tal y como se representa en la figura 6.1, cada posición viene caracterizada por una densidad de probabilidad Gaussiana, con un centro y una varianza determinadas. El centro determina el punto en el que está centrada la Gaussiana; La varianza, por el contrario, determina la anchura.

En el ejemplo se posee una predicción, que fue previamente calculada, y la cual estima que el objeto debía encontrarse en la posición X_k, con un margen de error. Sin embargo, el sensor calcula que la posición actual es Y_k, con otro margen de error (dependiendo de la precisión que proporcione el sensor). Combinando la medición y la predicción, se consigue una estimación notablemente certera de cuál es la posición del objeto. Así mismo, con cada iteración del algoritmo, el error en las predicciones va disminuyendo, haciendo que la precisión aumente hasta conseguir unas estimaciones óptimas.

A. Bibliografía

[1] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation, 2019.

[2] Gorka Guardiola Múzquiz Enrique Soriano Salvador. *Fundamentos de Sistemas Operativos: Una aproximación práctica usando Linux*. 2022. URL https://raw.githubusercontent.com/honecomp/honecomp.github.io/main/books/librossoo.pdf.

[3] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010. doi: 10.1109/TPAMI.2009.167.

[4] J. Jones and D. Roth. *Robot Programming: A Practical Guide to Behavior-Based Robotics*. McGraw-Hill Education, 2004. ISBN 9780071427784. URL https://books.google.es/books?id=cKL3zhGUC2YC.

[5] Ren.C. Luo, Shu-Ruei Chang, Chien-Chieh Huang, and Yee-Pien Yang. Human robot interactions using speech synthesis and recognition with lip synchronization. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, pages 171–176, 2011. doi: 10.1109/IECON.2011.6119307.

[6] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017. ISBN 0128122757.

[7] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. O'Reilly, Beijing, 2017. ISBN 978-1-4919-1425-0. URL https://www.safaribooksonline.com/library/view/deep-learning/9781491924570/.

[8] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger, 2016. URL https://arxiv.org/abs/1612.08242.

[9] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2015. URL https://arxiv.org/abs/1506.02640.

[11] Francisco Martín Rico. *A Concise Introduction to Robot Programming with ROS2*. CRC Press, 2023.

[12] 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. IEEE.

[13] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., USA, 2006. ISBN 0132392275.

[14] Dieter Fox Wolfram Burgard, Sebastian Thrun. *Probabilistic Robotics*. Ronald C. Arkin, 2005.