



UNIVERSIDADE DA CORUÑA



Escola Politécnica Superior

Trabajo Fin de Grado
CURSO 2019/20

*CREACIÓN DE UN MODELO DE SIMULACIÓN DE
ROBOT MÓVIL PARA CONDUCCIÓN AUTÓNOMA*

Grado en Ingeniería en Tecnologías Industriales

ALUMNO

Daniel Andrés Juanatey Hermo

TUTORES

Francisco Javier Bellas Bouza
Martín Naya Varela

FECHA

Septiembre 2020

Título

Creación de un modelo de simulación de robot móvil para conducción autónoma.

Resumen

Este Trabajo Fin de Grado tiene como objetivo implementar en simulación un modelo de vehículo capaz de circular de forma autónoma por un entorno urbano. Se requiere que este modelo de simulación pueda ser utilizado tanto en investigación como en docencia a distintos niveles. Para lograr la consecución de este objetivo se hará uso de herramientas software pertenecientes al campo de la visión por computación, destacando la detección de objetos en tiempo real, que serán empleadas en simulación y en un robot real.

La experimentación con vehículos de conducción autónoma en entornos reales supone diversos problemas en cuanto a coste e infraestructura. En cambio, el uso de software de simulación ahorra gran cantidad de costes en ensayos, material, reparaciones, etc. al mismo tiempo que ofrece distintas ventajas, como son la capacidad de realizar los experimentos tantas veces como sea necesario, y la posibilidad de documentar de forma fehaciente los resultados obtenidos. Además, este modelo de simulación ofrece otras ventajas durante su uso en docencia, ayudando a los centros educativos a proporcionar material didáctico al alumnado sin necesidad de que haya un robot por alumno, ofreciendo a mayores la posibilidad de continuar el curso de la materia de manera no presencial.

Título

Creación dun modelo de simulación de robot móbil para conducción autónoma.

Resumo

Este Traballo Fin de Grao ten como obxectivo levar a cabo en simulación un modelo de vehículo capaz de circular de forma autónoma por un entorno urbano. Requírese que este modelo de simulación poida ser utilizado tanto en investigación coma en docencia a distintos niveis. Para logra-la consecución de este obxectivo farase uso de ferramentas software pertencentes ao campo da visión por computación, destacando a detección de obxectos en tempo real, para a súa aplicación tanto en simulación coma nun entorno real.

A experimentación con vehículos de conducción autónoma en contornas reais pode supor diversos problemas en canto a custo e infraestrutura. Pola contra, o uso de software de simulación aforra gran cantidade de custos en ensaios, material, reparacións, etc. ao mesmo tempo que oferta outra serie de vantaxes, como son realiza-los experimentos tantas veces coma sexa necesario, e a posibilidade de documentar de forma fidedigna os resultados obtidos. Ademais, este modelo de simulación oferta outras vantaxes durante o seu uso en docencia, axudando a que os centros educativos poidan proporcionar material didáctico ao alumnado sen necesidade de que haxa un robot por alumno, ofrecendo a maiores a posibilidade de continuar o curso da materia de maneira non presencial.

Title

Development of a mobile robot simulation model for self-driving

Summary

This bachelor's degree project aims to simulate a vehicle model capable of driving autonomously through an urban environment. It is required that this simulation model can be used both in research and in different levels of teaching. To achieve this objective, software tools belonging to the field of computer vision will be used, highlighting the detection of objects in real time, for its implementation both in simulation and in a real robot.

Experimenting with self-driving vehicles in real-world environments can pose several cost and infrastructure issues. In return, the use of simulation software saves a lot of costs in tests, material, repairs, etc. at the same time that it allows, on the one hand, to carry out the experiments as many times as necessary, and on the other hand, to reliably document the results obtained. In addition, this simulation model takes into account the advantages of its use in teaching, helping educational centers to provide didactic material to students without the need for a robot per student, also offering the possibility of continuing the course of the matter in a non-presential way.



UNIVERSIDADE DA CORUÑA



Escola Politécnica Superior

**TRABAJO FIN DE GRADO
CURSO 2019/20**

*CREACIÓN DE UN MODELO DE SIMULACIÓN DE
ROBOT MÓVIL PARA CONDUCCIÓN AUTÓNOMA*

Grado en Ingeniería en Tecnologías Industriales

Documento

ÍNDICE

Índice General

1 Título y resumen.....	2
2 Índice.....	5
3 Memoria.....	7
4 Anexos.....	71
5 Presupuesto.....	88



Escola Politécnica Superior

**TRABAJO FIN DE GRADO
CURSO 2019/20**

*CREACIÓN DE UN MODELO DE SIMULACIÓN DE
ROBOT MÓVIL PARA CONDUCCIÓN AUTÓNOMA*

Grado en Ingeniería en Tecnologías Industriales

Documento

MEMORIA

Índice de contenido

1	Introducción	13
1.1	Robótica educativa	13
1.2	Software de simulación	14
2	Antecedentes y objetivos	16
2.1	Antecedentes	16
2.1.1	Conducción autónoma	16
2.1.2	Robótica educativa	19
2.2	Objetivos	24
2.2.1	Subobjetivos	24
3	Especificaciones de diseño	25
3.1	Robobo	25
3.1.1	Hardware	25
3.1.2	Software	27
3.2	Software de simulación	30
3.2.1	Interfaz de usuario	30
3.2.2	Escenas, modelos y objetos	32
3.2.3	Programación en CoppeliaSim	35
3.3	Software de visión	36
3.3.1	OpenCV	36
3.3.1	TensorFlow	37
4	Diseño del sistema	38
4.1	Modelo 3D Robobo	38
4.2	Modelado del entorno	39
4.2.1	Señalización horizontal	39
4.2.2	Señalización vertical	40
5	Implementación del sistema	42
5.1	Detección de carriles	42
5.1.1	Algoritmo de Canny	42
5.1.2	Transformada de Hough	44
5.1.3	Visualización	45
5.2	Detección de objetos	46
5.2.1	Selección del modelo	46
5.2.2	Entrenamiento del modelo	47
5.2.3	Ejecución en tiempo real	53
6	Protocolo de pruebas y validación	55

6.1 Validación de la detección de carriles	55
6.1.1 Análisis de los valores de las coordenadas en función de la velocidad.....	55
6.1.2 Análisis de los valores de las coordenadas en función de la curvatura.....	56
6.1.3 Conclusiones.....	58
6.2 Validación de la detección de objetos	58
6.2.1 Análisis de la puntuación con la distancia.....	58
6.2.2 Conclusiones.....	60
7 Resultados obtenidos	62
7.1 Desempeño en curvas	62
7.2 Frenado automático.....	63
7.3 Obediencia de las señales verticales	63
7.4 Obediencia de los semáforos.....	64
8 Estudio de aplicabilidad	66
8.1 Aplicación del modelo.....	66
8.2 Comparación con sistemas ya existentes	66
8.3 Trabajo futuro	67
9 Referencias	68

Índice de figuras

Figura 2-1 Detalle de las calles del futuro según la maqueta Futurama (1939).....	16
Figura 2-2 Los tres vehículos autónomos de la demostración en París	18
Figura 2-3 Interior de uno de los vehículos autónomos experimentales	18
Figura 2-4 LEGO Mindstorms EV3.....	19
Figura 2-5 Modelo de simulación del EV3.....	20
Figura 2-6 Modelos oficiales de TurtleBot 3	20
Figura 2-7 Robot Khepera IV	21
Figura 2-8 NAO en CoppeliaSim.....	22
Figura 2-9 Elementos robot humanoide NAO	22
Figura 2-10 Varias plataformas móviles Romo.....	23
Figura 2-11 Plataforma móvil Robobo.....	23
Figura 3-1 Vista en planta de la base Robobo	26
Figura 3-2 Ejes de rotación de la unidad Pan-Tilt.....	26
Figura 3-3 Vista posterior y alzado de la base Robobo	27
Figura 3-4 Framework del Robobo.....	28
Figura 3-5 Diagrama arquitectura Robobo framework.....	29
Figura 3-6 Ventanas de consola y aplicación de CoppeliaSim	31
Figura 3-7 Detalle de ventana de aplicación	32
Figura 3-8 Tipos de objetos de escena	32
Figura 3-9 Icono de las formas puras.....	33
Figura 3-10 Icono de las formas aleatorias	33
Figura 3-11 Icono de las formas convexas.....	33
Figura 3-12 Capa visible y capa oculta del Robobo	34
Figura 3-13 Par de revolución, prismático, helicoidal y rótula.....	35
Figura 3-14 Proyección ortogonal y proyección en perspectiva	35
Figura 4-1 Modelo de Robobo con cámara incorporada	38
Figura 4-2 Segmentos independientes del trazado	40
Figura 4-3 Señales de reglamentación en CoppeliaSim.....	40
Figura 4-4 Semáforos en CoppeliaSim	41
Figura 5-1 Efecto del algoritmo de Canny sobre una imagen.....	43
Figura 5-2 Aplicación de una máscara triangular a la imagen	43
Figura 5-3 Recta representada por un punto	44
Figura 5-4 Punto representado por una recta	44
Figura 5-5 Ecuación de la recta a partir de punto de corte	45
Figura 5-6 Líneas del carril detectadas en la imagen	45
Figura 5-7 Conjunto de datos para el entrenamiento del modelo	48

Figura 5-8 Ventana de aplicación de LabelImg	49
Figura 5-9 Generación de los datos de entrada	50
Figura 5-10 Mapa de etiquetas para detección	50
Figura 5-11 Entrenamiento del modelo	52
Figura 5-12 Gráfico de la pérdida en función del tiempo	53
Figura 6-1 Disposición de los elementos para la primera prueba	55
Figura 6-2 Disposición de los elementos para la segunda prueba	56
Figura 6-3 Línea fuera del área de detección.....	57
Figura 7-1 Robobo circulando por una curva	62
Figura 7-2 Frenado automático del Robobo.....	63
Figura 7-3 Reconocimiento de señales de tráfico	64
Figura 7-4 Reconocimiento de semáforos.....	65

Índice de tablas

Tabla 2-1 Niveles de autonomía definidos por la SAE	17
Tabla 5-1 Modelos de TensorFlow utilizados	47
Tabla 6-1 Valores de las coordenadas en función de la velocidad	56
Tabla 6-2 Valores de las coordenadas en función de la curvatura	57
Tabla 6-3 Valores de la confianza del Robobo en función de la distancia	59
Tabla 6-4 Valores de la confianza de los semáforos en función de la distancia	59
Tabla 6-5 Valores de la confianza de las señales en función de la distancia	60

1 INTRODUCCIÓN

Este Trabajo Fin de Grado (TFG) se enmarca dentro de un proyecto de investigación del Grupo Integrado de Ingeniería (GII) de la Universidade da Coruña (UDC) con el que se pretende avanzar en la investigación de la conducción autónoma mediante la incorporación de algoritmos de reconocimiento de objetos, toma de decisiones, comportamientos colaborativos, etc. a una plataforma robótica móvil.

Las tareas de estudio e implementación de los distintos algoritmos se realizarán con una plataforma robótica móvil controlada mediante un smartphone desarrollada por la spin-off de la UDC, Manufactura de Ingenios Tecnológicos (MINT). Esta plataforma consta tanto de elementos actuadores como son motores y leds, como de elementos sensores como son los sensores infrarrojos o los encoders de los motores, además de todos los que incorpora el propio smartphone.

Con objeto de emplear dicha plataforma robótica para el estudio de la conducción autónoma en entornos reales, se ha planteado la idea de recrear el comportamiento de un vehículo autónomo en un entorno de simulación que sea realista. Para que dicha recreación sea lo más fiel posible a la realidad se cree conveniente, por una parte, dotar al vehículo de aquellos comportamientos y algoritmos que se consideren necesarios para la realización de tareas de conducción autónoma (identificación de objetos, toma de decisiones, etc.). Por otra parte, crear un entorno de simulación que sea representativo de la casuística que los vehículos autónomos se pueden encontrar en la realidad, añadiendo objetos tales como señales, semáforos, cruces, etc. Con esto se pretende que el vehículo disponga de los comportamientos necesarios para desplazarse de forma autónoma y segura por la ciudad diseñada, a la vez que los comportamientos empleados en simulación sean aplicables a un robot y entorno reales.

1.1 Robótica educativa

En los últimos años la robótica ha experimentado un aumento notable de su incidencia en nuestras vidas. Esta no solo crece en el ámbito industrial sino también en el ámbito doméstico y nuestra vida privada, con los denominados robots de servicio [1].

Conscientes de que la robotización de la sociedad continuará aumentando, las autoridades competentes en educación procuran preparar lo mejor posible a los estudiantes a esta nueva realidad [2], propiciándose un auge de la robótica educativa. Esta disciplina se va abriendo camino en todos los niveles educativos, desde primaria hasta estudios de posgrado. En este sentido, en varios países europeos y algunas comunidades autónomas en nuestro país, la robótica ya forma parte del currículo educativo [3]. Por esto, pese a que esta disciplina no se ha extrapolado aún a todos los colegios españoles, no sería descabellado pensar que pueda llegar a hacerlo en un futuro cercano.

La robótica combina, entre otras, áreas de ciencia, tecnología, ingeniería y matemáticas [4]. Como área multidisciplinar que es, no existe un único camino para sumergirse en ella. Una persona que trabaje en este campo haberse introducido a través de otras áreas, tales como la electrónica, informática, biotecnología, ciencia cognitiva, etc. Existen múltiples caminos posibles para un trabajo en robótica. La UDC, en su afán de mantenerse a la vanguardia de la enseñanza, también se ha metido de lleno en el mundo de la robótica educativa. Para ello cuenta con un robot educativo de creación propia, la plataforma móvil desarrollada por MINT antes mencionada, llamada Robobo [5]. Este Trabajo Fin de Grado surge de la necesidad de la UDC de mantener su robot educativo actualizado, tratando de llegar a todos los niveles educativos y áreas posibles.

La singular forma del robot educativo de la universidad y sus características (una plataforma móvil con dos ruedas, con una alta sensorización) lo convierten en una gran herramienta para aplicación en el área de la robótica autónoma móvil. De este modo, es posible profundizar en un campo que actualmente está adquiriendo gran relevancia, como es la conducción autónoma de vehículos. El interés generado sobre este campo se debe a sus potenciales beneficios futuros a nivel de reducción de accidentes, atascos, contaminación, etc.

La mejora en la seguridad de un vehículo autónomo viene determinada por la eliminación del factor humano. Entre los factores humanos podemos destacar la conducción distraída, velocidad inadecuada, fatiga, conducción bajo los efectos del alcohol u otros estupefacientes, somnolencia o falta de experiencia.

Pero las mejoras de seguridad y tráfico no dependen sólo del nivel de automatización del propio vehículo, es de gran interés el estudio de comportamientos colaborativos. La capacidad de comunicarse con otros vehículos y con su entorno proporciona información relevante para la conducción, al poder notificar unos vehículos a otros la aparición de atascos, posibles peligros en la carretera o plazas de aparcamiento libres. De este modo, los comportamientos colaborativos permitirían reducir los atascos desviando a los vehículos a vías menos congestionadas, o reducir las vueltas necesarias para encontrar un aparcamiento. Estas mejoras en movilidad llevarían asociada una evidente disminución de la contaminación.

1.2 Software de simulación

Mientras que el mundo real es obviamente complejo y con múltiples variables aleatorias y difícilmente predecibles, un entorno de simulación ayuda a simplificar el problema que queramos abordar al proporcionarnos un control absoluto sobre el entorno y los experimentos que vayamos a realizar. Además, la simulación ahorra gran cantidad de costes, tanto materiales como de tiempo, en ensayos. Al mismo tiempo, permite reproducir los experimentos o prácticas que se realicen tantas veces como sea necesario, algo que resulta difícil con robots reales ya que las condiciones siempre cambian en un entorno real, como por ejemplo la iluminación, el estado de las baterías, etc. [6]

Como puede imaginarse, la experimentación en entornos reales de modelos de conducción autónoma está al alcance de muy pocas corporaciones y centros de investigación en todo el mundo. El elevado coste de este y otros tipos de proyectos puede ser tan alto que podría hacer inabordable su elaboración. Esto, unido a que los ordenadores personales son cada vez más potentes, ha popularizado el uso de software de simulación, contando estos en la actualidad con una potencia de procesamiento más que suficiente para hacer un modelado 3D y la renderización de un robot y su entorno, con un motor de física que genere un movimiento del robot completamente realista. Concretamente para este proyecto, el software de simulación escogido es CoppeliaSim, del que hablaremos extensamente en el apartado 3.2.

Para el ámbito de la robótica educativa, disponer de un modelo de simulación que represente al robot real de forma fidedigna y que sea programable de la misma forma que éste ofrece grandes ventajas desde el punto de vista didáctico. La primera ventaja que se puede destacar es el coste de adquisición. Puesto que, con los tiempos de incertidumbre económica actuales, los centros de enseñanza ven muy limitado su presupuesto para la adquisición de nuevo material educativo. Un modelo de simulación permite que los centros educativos puedan proporcionar a los alumnos acceso al material didáctico sin necesidad de que haya un robot para cada uno de ellos, lo cual en aulas con un gran número de estudiantes es muy significativo. El uso de software de simulación también es muy útil en casos de docencia no presencial y acceso restringido

a los recursos, como puede ser una universidad a distancia o una situación de confinamiento como la vivida recientemente. Además, al familiarizarse de esta forma con el robot, su programación y control, cuando se emplee el robot real los riesgos de averías derivadas de un mal uso o al desconocimiento se minimiza.

2 ANTECEDENTES Y OBJETIVOS

2.1 Antecedentes

El objetivo de este capítulo es proporcionar al lector un contexto histórico sobre los dos temas que suponen el núcleo de este proyecto: conducción autónoma y robótica educativa. Para ayudarnos en esta tarea se realizará un análisis de las investigaciones previas en estos ámbitos que se consideraron más relevantes.

2.1.1 Conducción autónoma

En el presente apartado se pretende dar una visión global del mundo de la conducción autónoma. Para ello, se dará un breve repaso a la evolución histórica de la conducción autónoma desde sus inicios hasta su contexto actual.

No tuvo que pasar mucho tiempo después del nacimiento del automóvil para que los inventores comenzaran a pensar en formas de convertir automóviles convencionales en autónomos. Ya en 1925, un ingeniero eléctrico llamado Francis P. Houdina fue el responsable de poner en marcha el primer vehículo que se podría considerar como mínimamente autónomo. El coche funcionaba gracias al uso de un control remoto que era manipulado por el inventor, seguía las órdenes que llegaban a su antena enviadas desde el coche que iba detrás. El vehículo era una máquina capaz de encender el motor, andar, esquivar otros coches y hasta hacer sonar el claxon sin un piloto a bordo.

El evento ocurrió en Manhattan. El coche, creado por la empresa Houdina Radio Control, maniobró durante varios kilómetros y estuvo en marcha hasta que dio a parar contra la parte posterior de otro vehículo.

En 1939, con motivo de la Exposición Mundial de Nueva York, Norman Bel Geddes presentó su exposición Futurama, una enorme maqueta que mostraba un posible modelo del mundo de 20 años en el futuro. Ahí aparecía su idea de coches eléctricos conducidos de forma autónoma, propulsados por campos electromagnéticos generados por circuitos insertados en el pavimento. En la Figura 2-1 se muestra una fotografía con un plano de detalle de la maqueta.



Figura 2-1 Detalle de las calles del futuro según la maqueta Futurama (1939)

Estos hitos históricos, aunque demuestran el interés que hubo desde un inicio en este método de transporte, no dejan de ser anecdóticos pues distan mucho de la idea de coche autónomo actual.

Un vehículo autónomo actual es un tipo de automóvil con los sensores, unidades de procesamiento, software y actuadores necesarios para conducirse por sí mismo, es capaz de percibir el medio que le rodea y navegar en consecuencia. Estos vehículos perciben el entorno mediante sistemas complejos como radar, LIDAR, GPS, navegación inercial, sensores de ultrasonido, sensores infrarrojos y visión computarizada, e interpretan la información de estos sistemas para identificar la ruta más adecuada para llegar a su destino, así como para evitar los obstáculos que se encuentre en su camino y cumplir con la normativa viaria vigente.

Los coches que vemos a diario no son completamente autónomos, en función de su grado de autonomía, la SAE (Sociedad de Ingenieros Automotrices) ha creado un estándar [7] que clasifica a todos los vehículos estableciendo seis niveles de conducción. Los niveles del 0 al 2 requieren de un mayor grado de atención humana, mientras que los niveles del 3 al 5 el coche ya es en su mayor parte autónomo y requiere de un menor grado de atención humana (excepto el último nivel, el nivel máximo de automatización que en teoría no requeriría de conductor).

Nivel	Definición	Resumen
0	Solo el conductor	El coche no tiene ningún sistema automático. Las tareas de conducción son realizadas completamente por el conductor (humano).
1	Asistente de conducción	El conductor humano sigue realizando la mayoría de las tareas, ayudado por sistemas como control de velocidad o mantenimiento en carril.
2	Semi autonomía	El vehículo es capaz de controlar completamente el movimiento, pudiendo el conductor humano apartar las manos del volante, pero siempre atento ante posibles eventualidades.
3	Autonomía controlada	El coche tiene sistemas de control de movimiento longitudinal, lateral, detección y respuesta ante objetos y en caso de no ser capaz debe informar al usuario con tiempo suficiente para que pueda reaccionar adecuadamente e intervenir.
4	Alto nivel de autonomía	Igual que el anterior, aunque si el conductor humano no responde adecuadamente en caso de fallo, el coche es capaz de conducir hasta una situación de riesgo mínimo.
5	Autonomía total	El coche es capaz de actuar en cualquier vía, momento o circunstancia como si fuese un conductor humano.

Tabla 2-1 Niveles de autonomía definidos por la SAE

El padre de este tipo de coche autónomo fue Ernst Dickmanns, ingeniero aeroespacial e investigador en la Universidad Bundeswehr de Múnich. En 1986 compró una furgoneta Mercedes e instaló en ella computadoras, cámaras y otros sensores, y comenzó a realizar pruebas en las instalaciones de la universidad. Para el año siguiente consiguió que esta furgoneta se condujese de manera autónoma por una autopista vacía [8].

Poco después de este hito, Dickmanns fue abordado por el fabricante de automóviles alemán Daimler. Juntos, obtuvieron fondos del proyecto Eureka “PROgraMme for a European Traffic of Highest Efficiency and Unprecedented Safety” (PROMETHEUS), el mayor proyecto de I+D jamás visto en este campo con una inversión de 749 millones de euros [9].

Tras años de investigación, en 1994 hicieron una demostración en una autopista francesa con dos Mercedes 500 SEL, siendo capaces de acelerar hasta 130 km/h, cambiar de carril y reaccionar ante otros coches de forma autónoma, con un sistema de ordenador de a bordo que controlaba el volante, el pedal del acelerador y los frenos.



Figura 2-2 Los tres vehículos autónomos de la demostración en París



Figura 2-3 Interior de uno de los vehículos autónomos experimentales

En la actualidad, los líderes en el ámbito del desarrollo de vehículos autónomos son, como cabía esperar, los principales fabricantes de automóviles y compañías tecnológicas (General Motors, Ford, Volkswagen, Tesla, Uber, etc.) siendo el más destacado hasta el momento [10] el proyecto de Waymo, perteneciente al conglomerado de empresas propietario de Google, con millones de km recorridos sin conductor en carreteras públicas, y miles de millones de km recorridos en simulación [11].

2.1.2 Robótica educativa

El comportamiento autónomo en el mundo de la robótica comenzó a finales de la década de 1940 con las “tortugas” de William Grey Walter [12] y por su parte, la robótica autónoma llegó al mundo de la educación a finales de los noventa, fruto de una colaboración entre LEGO y el Instituto de Tecnología de Massachusetts (MIT) [13].

En este apartado realizaremos un recorrido por los distintos robots autónomos usados en educación e investigación, desde los más conocidos, pasando por otros cuyas características se ajustan más a los objetivos planteados en este proyecto, y terminando por el propio Robobo y sus modelos realizados con anterioridad a este trabajo. La mayoría de los robots que veremos poseen su propio modelo de simulación en CoppeliaSim.

2.1.2.1 LEGO Mindstorms EV3

La opción más popular entre los robots educativos. La pieza fundamental del set es el ladrillo inteligente EV3, equipado con un procesador ARM9, un puerto USB para proporcionar WI-FI, lector de tarjetas Micro SD, altavoz, pantalla, botones retroiluminados y 4 puertos de motor, a los que se pueden conectar los sensores y actuadores incluidos. Estos son: 3 servomotores interactivos, sensor de color, sensor de contacto y sensor de infrarrojos. Además, cuenta con un control remoto y 550 piezas LEGO, con instrucciones para construir 5 robots diferentes [14].



Figura 2-4 LEGO Mindstorms EV3

Tiene una interfaz propia de programación en bloques, lo que lo vuelve intuitivo y fácil de usar, por lo que su uso es bastante extendido por centros de educación secundaria. Para niveles superiores es posible actualizar el firmware y programar el robot con Python [15], pero sigue bastante limitado al no contar con sensorización avanzada como cámaras o láser, ni soportar algoritmos complejos como reconocimiento de objetos al no contar con la capacidad de cómputo necesaria para ello.

En la Universidad de Málaga han realizado un modelo de simulación en CoppeliaSim de uno de los modelos de robot que se puede construir con el kit de LEGO Mindstorms EV3, además de la creación de una toolbox de Matlab basada en las funciones del lenguaje NXC que permite controlarlo en dicho entorno de simulación [16].

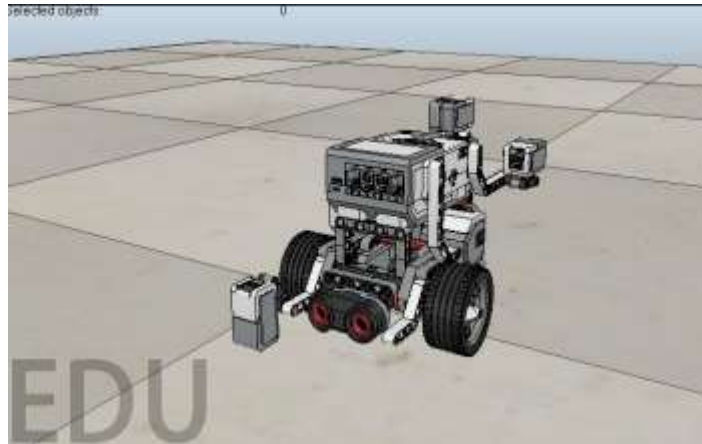


Figura 2-5 Modelo de simulación del EV3

La falta de sensorización avanzada o, en caso de haberla, la escasa potencia de procesamiento antes mencionada, no es un problema único del EV3 sino que es un mal compartido por la mayoría de robots educativos como el Thymio II [17] o robots basados en Arduino como el mBot [18]. Debemos por tanto enfocarnos en robots más avanzados como los que veremos a continuación.

2.1.2.2 TurtleBot 3

Es un kit robótico con software de código abierto y programable en ROS [19]. Su tercera versión fue lanzada en 2017 por Open Robotics y ROBOTIS, además de muchos otros colaboradores. Dicha versión está formada por piezas impresas en 3D y es modular, por lo que la base tiene un tamaño mucho más reducido que la versión anterior, pero sin perder funcionalidad al ser fácilmente expandible. Esto hace que haya muchas configuraciones posibles, pero centrándonos en los kits más populares a la venta estos incluyen: una base con controlador OpenCR, dos ruedas con motor propio, batería de 1800 mAh, una placa SBC que puede ser una Raspberry Pi o Intel Joule, cámara y LIDAR para navegación por mapeo y localización simultáneos (SLAM, por sus siglas en inglés).

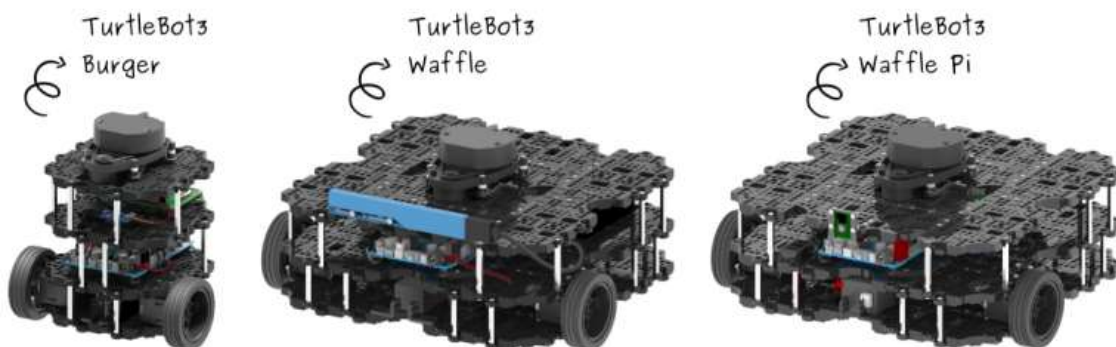


Figura 2-6 Modelos oficiales de TurtleBot 3

2.1.2.3 Khepera IV

Es un robot móvil especialmente compacto y sensorizado para cualquier actividad de laboratorio. Está dotado con 8 sensores infrarrojos horizontales para detección de obstáculos, 4 más en la parte inferior para evitar caídas o seguir líneas, 5 sensores de ultrasonidos, cámara a color, WI-FI, bluetooth, USB, acelerómetro, giroscopio, micrófono, altavoz y un bus de extensión para conectar complementos.

Sin embargo, lo más destacado de este robot es que cuenta con un sistema operativo Linux completo, lo que facilita el desarrollo de aplicaciones con C/C++ o importar librerías [20].



Figura 2-7 Robot Khepera IV

2.1.2.4 NAO

NAO es un robot humanoide de 58 centímetros, interactivo y totalmente programable. Nació en 2008, desarrollado por la empresa Softbank Robotics y ha pasado por 5 versiones hasta llegar al modelo actual *evolution v5*.

El robot es capaz de interactuar de forma natural con todo tipo de público. Escucha, ve, habla y se relaciona con el medio según se haya programado. Tiene aplicaciones útiles en educación (tanto para estudiantes como para profesores e investigadores), uso doméstico y empresas.

NAO es capaz de percibir el entorno a partir de sus múltiples sensores, entre los cuales se encuentran dos cámaras, cuatro micrófonos, nueve sensores táctiles, dos sensores de ultrasonidos, 8 sensores de presión un acelerómetro y un giroscopio. Además, es capaz de interactuar gracias a sus 53 LEDs RGB, un sintetizador de voz y dos altavoces. Incluye un software gráfico de programación llamado Choreographe, compatible con Windows, Linux y macOS, que permite programarlo sin tener conocimientos de un lenguaje de programación. Para usuarios más avanzados permite usar C++, Python, Java y Matlab.

Pese a ser extensamente empleado en docencia e investigación, y contar con un modelo de simulación realista, este robot no es válido para su uso en conducción autónoma.

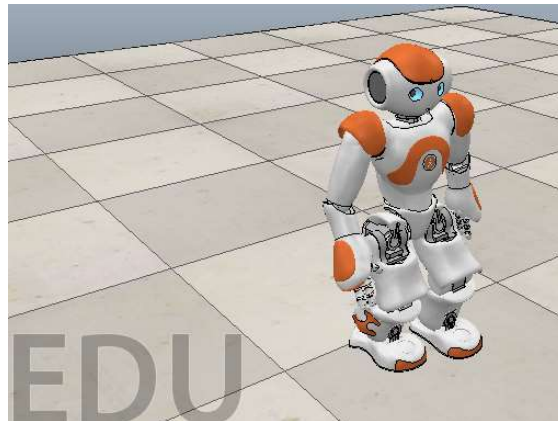


Figura 2-8 NAO en CoppeliaSim

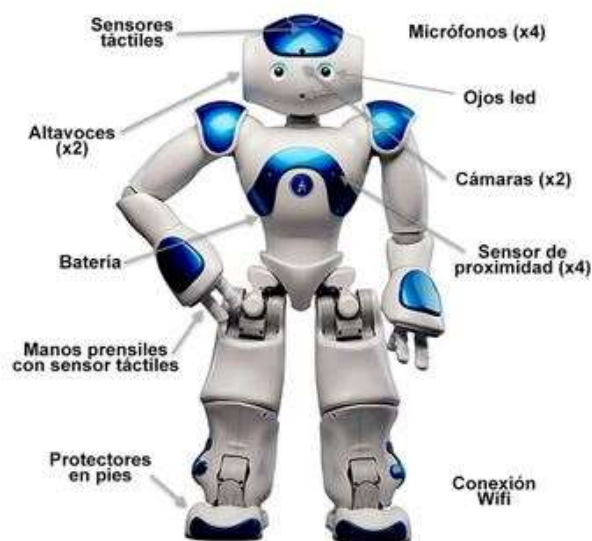


Figura 2-9 Elementos robot humanoide NAO

Como se puede observar, estos últimos robots que hemos comentado son muy completos, lo cual es directamente proporcional a su precio. Para encontrar una forma de conseguir un robot potente a un precio asequible, surgió la idea de emplear una base móvil a la que acoplar un smartphone para aprovechar sus sensores, su (relativamente) potente procesador y su enorme conectividad.

2.1.2.5 Romo

Fue una de las primeras plataformas móviles con smartphone, fabricada por Romotive y formada por una base con forma de tanque y compatible con dispositivos iOS (iPod 4, iPod 5, iPhone 4/4S y iPhone 5/5C/5S).

La aplicación del smartphone permitía al robot perseguir objetos, reconocer caras o ser controlado remotamente, entre otras cosas. El principal problema es que fue un proyecto financiado en una web de microfinanciación que dejó de actualizarse en 2013 [21], por lo que actualmente la aplicación se encuentra desfasada y sin soporte oficial. A este inconveniente hay que sumar la ausencia total de sensores en la plataforma y que los iPhone no son un tipo de smartphone barato. Además, Apple empleó su nuevo conector Lightning desde la quinta generación de sus dispositivos.

Debido a esto, por ejemplo, si tu Romo cuenta con conector de 30 pines, no será compatible con iPhone 5.



Figura 2-10 Varias plataformas móviles Romo

Las diferentes carencias anteriormente mencionadas de este y otro tipo de plataformas móviles como el Wheelphone [22] llevaron al Grupo Integrado de Ingeniería de la UDC a desarrollar su propia plataforma móvil con smartphone, el Robobo, que será finalmente el robot empleado para la realización del presente TFG.

2.1.2.6 Robobo

En relación con el Robobo, en los últimos años, alumnos de la Escuela Politécnica Superior (EPS) han contribuido al desarrollo del modelo de simulación del Robobo mediante trabajos fin de grado y fin de máster en los que se han ido mejorando progresivamente distintos aspectos del modelo de simulación. Partiremos del más reciente de ellos [23], donde los motores (de las ruedas y unidad Pan-Tilt) y sensores infrarrojos se comportan de forma análoga al Robobo real, y se dispone de una librería con las funciones correspondientes a los elementos mencionados.

Sin embargo, a diferencia del modelo real, este modelo en simulación no incluye ninguna funcionalidad que implique el uso de la cámara, por lo que se deberá trabajar en esta parte desde cero. Tampoco están disponibles otras funciones que emplean otros sensores del teléfono tales como el micrófono, acelerómetro, pulsación en pantalla, etc. que sí están disponibles en el robot real.



Figura 2-11 Plataforma móvil Robobo

2.2 Objetivos

El objetivo general del proyecto consiste en el desarrollo de un modelo en simulación de un robot móvil que pueda ser utilizado tanto en docencia como en investigación en el campo de la conducción autónoma.

2.2.1 Subobjetivos

La consecución de este objetivo general implica la realización de los siguientes objetivos específicos o subobjetivos:

- Diseño e implementación de un entorno de simulación de las características urbanas básicas existentes en una ciudad encargadas de permitir la circulación de vehículos.
- Implementación de diferentes tipos de señales existentes en la realidad para cumplir con la normativa de circulación.
- Estudio y análisis de las funcionalidades básicas necesarias para dotar a un vehículo de comportamiento autónomo.
- Implementación de las funcionalidades básicas del punto anterior para dotar a un dispositivo robótico móvil de comportamiento autónomo.
- Integración del vehículo autónomo dentro del entorno urbano creado.

3 ESPECIFICACIONES DE DISEÑO

A lo largo del presente capítulo se realizará una descripción detallada de todas las herramientas utilizadas para el diseño e implementación del modelo de simulación de robot móvil para conducción autónoma. Se explicarán fundamentos teóricos y tecnológicos que han sido necesarios para llevar a cabo este proyecto. Entre ellos, se detallará la configuración de la plataforma robótica empleada y se describirá el funcionamiento del software de simulación y de visión por computador. Estos tres elementos cuentan con una serie de requisitos impuestos por el GII, con objeto de continuar con la línea de trabajo establecida. Concretamente, las especificaciones de diseño o directrices a seguir son:

- **Plataforma robótica:** los algoritmos de conducción autónoma desarrollados se implementarán en el robot educativo propio de la UDC, el Robobo.
- **Software de simulación:** se hará uso de la última versión de CoppeliaSim.
- **Software de visión:** a escoger por el alumno, a condición de que deben emplearse herramientas que permitan su programación en Python.

3.1 Robobo

Puesto que este proyecto va a emplear un modelo en simulación del Robobo, para llevar a cabo dicha simulación será necesario conocer los componentes físicos del Robobo, así como su distribución [24] y las herramientas software empleadas para su conexión y control.

3.1.1 Hardware

Robobo cuenta con una base móvil a la que se le acopla un smartphone con el que se conecta de forma inalámbrica mediante Bluetooth. La base cuenta con dos ruedas, una unidad Pan-Tilt a la que se acopla el smartphone, cinco leds frontales y dos traseros, cinco sensores infrarrojos frontales y tres traseros, un conector micro USB tipo B y un botón de encendido. Dichos componentes aparecen indicados en las siguientes figuras.

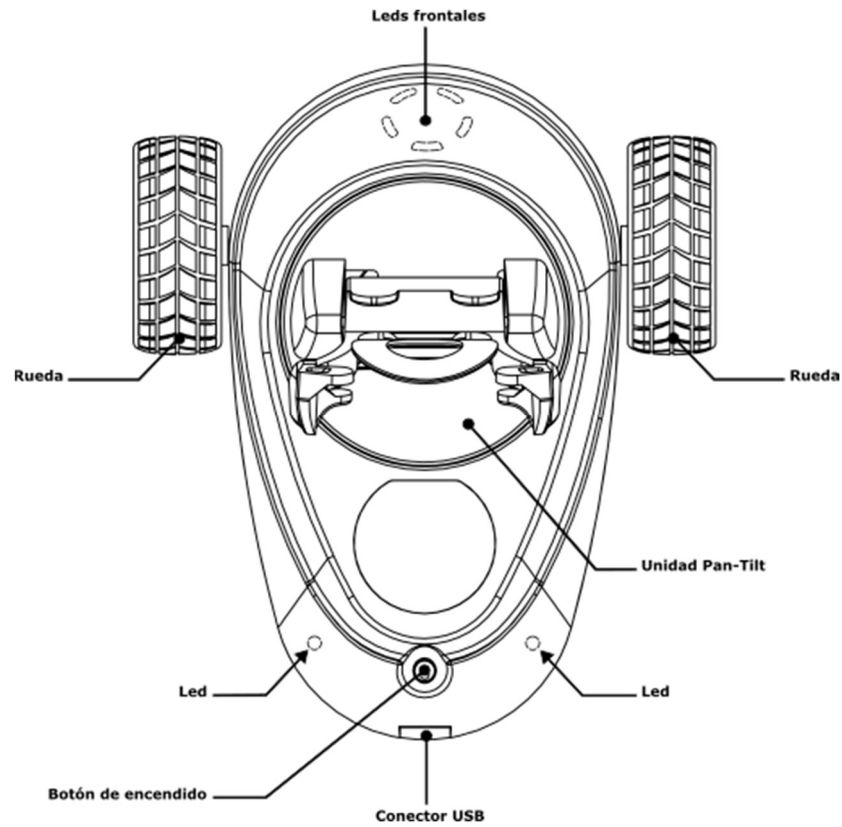


Figura 3-1 Vista en planta de la base Robobo

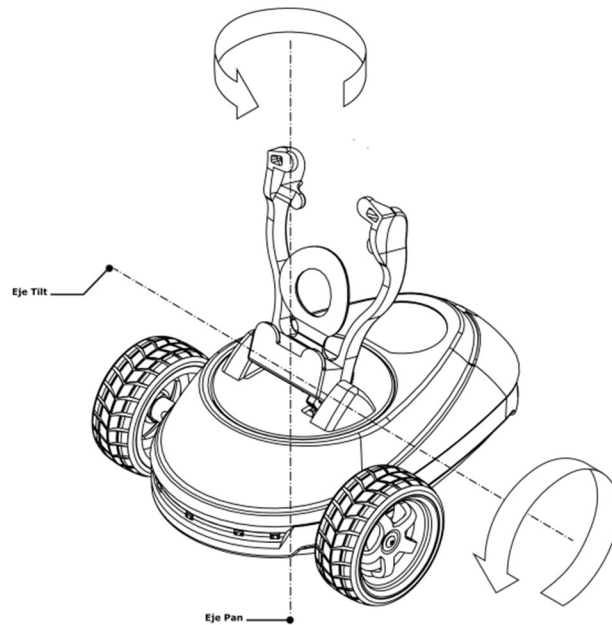


Figura 3-2 Ejes de rotación de la unidad Pan-Tilt

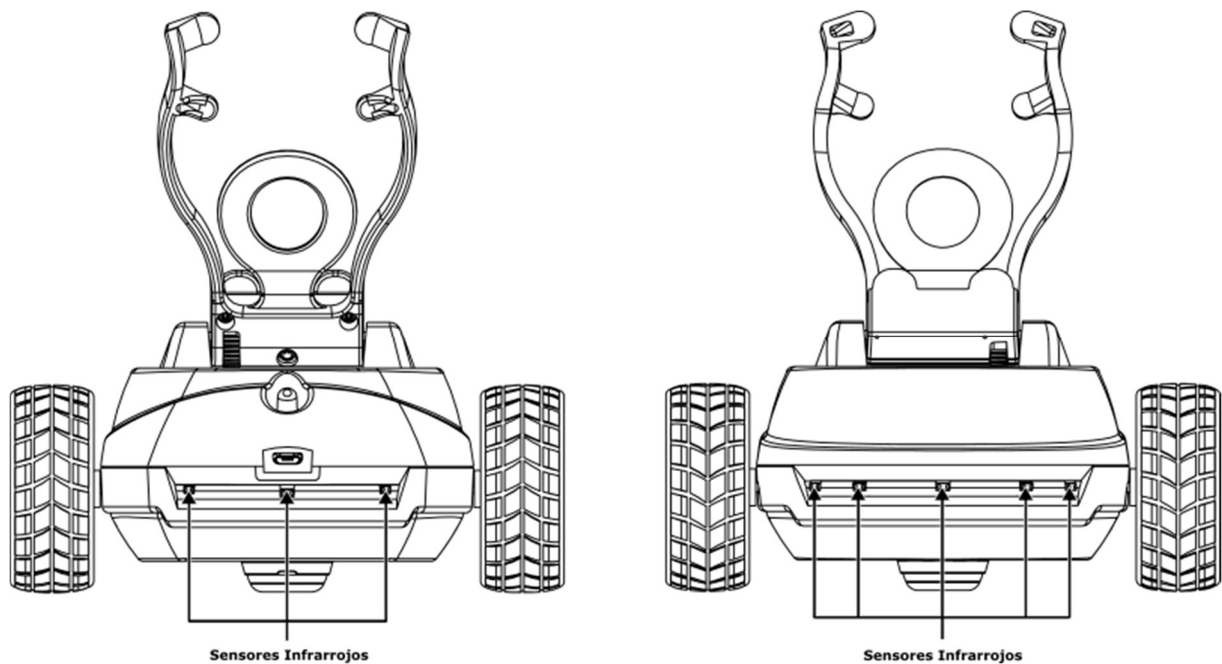


Figura 3-3 Vista posterior y alzado de la base Robobo

3.1.2 Software

El framework de Robobo permite programar el robot desde un ordenador o tablet con Windows, Linux o macOS. El único requisito es que el ordenador o tablet esté conectados a la misma red WI-FI a la que está conectado el smartphone del Robobo. El framework está organizado en tres niveles: para programadores principiantes, para intermedios, y para usuarios avanzados [25]. Todas estas librerías y herramientas son de código abierto.



Figura 3-4 Framework del Robobo

Como se observa en la Figura 4-4, el enfoque software del proyecto Robobo se basa en herramientas que los diseñadores y desarrolladores del framework han considerado adaptadas a las edades de los estudiantes, al no existir un paradigma de programación adecuado a todos los niveles educativos o de experiencia en robótica. Concretamente para el Robobo se han establecido:

- **Principiantes:**
 - *Entorno de programación:* Scratch 3
 - *Simulador:* propio, basado en Unity
- **Intermedios:**
 - *Entorno de programación:* librerías Python
 - *Simulador:* V-REP
- **Avanzados:**
 - *Entorno de programación:* librerías ROS
 - *Simulador:* Gazebo

El enfoque planteado en el Robobo framework parte de una programación sencilla basada en bloques para la primera etapa. Continuando en una segunda etapa con el uso del lenguaje Python, potente, pero con una sintaxis más intuitiva y fácil que otros lenguajes de programación, para llegar progresivamente en la tercera y última etapa a una programación de alto nivel basada en ROS para los estudiantes universitarios.

Además de la selección del lenguaje de programación, se estableció la necesidad de dotar al producto de un simulador que permita que los alumnos lo puedan programar sin depender totalmente del robot real. Disponer de un simulador posibilita, además de las ventajas mencionadas al principio de este documento, que los alumnos puedan trabajar de forma individualizada simplemente disponiendo de un ordenador y fuera de horas de clase. De esta forma cada alumno puede trabajar de forma independiente, adaptando su trabajo a sus necesidades y ritmo de estudio. Además, posibilita que los docentes avancen más en la materia, y dejen los robots reales como elemento de prueba para mostrar el funcionamiento real de los programas desarrollados.

Para dar soporte a los diferentes lenguajes de programación y simuladores de forma simple, se diseñó e implementó una arquitectura de software escalable y modular, cuya estructura funcional se muestra en el diagrama de la Figura 3-5:

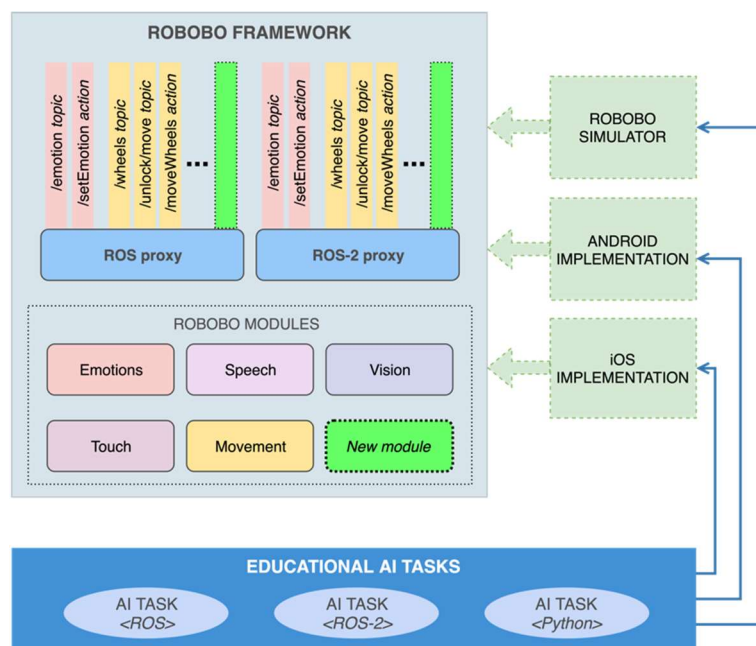


Figura 3-5 Diagrama arquitectura Robobo framework

Como se aprecia en el esquema anterior, el entorno de desarrollo está compuesto por una serie de módulos que implementan funcionalidades específicas tales como movimiento, habla, visión, etc. Existen diferentes implementaciones del entorno de desarrollo para iOS, Android y versiones simuladas del robot. Todas ellas comparten la misma API de comunicación, permitiendo que el mismo programa escrito en Python o ROS pueda ser ejecutado indistintamente usando un Robobo con Android, un Robobo con iOS, o un simulador.

Esta arquitectura proporciona dos formas diferentes de programar el robot: programación nativa usando el Robobo Framework en Java para dispositivos Android y Swift para los iOS, y remotamente usando la característica de control remoto que expone los comandos de actuación y el estado de los sensores. Estas capacidades de control remoto permiten la programación del robot con cualquier lenguaje que implemente el protocolo remoto de Robobo, basado en JSON, que puede ser interconectado a través de diferentes tecnologías de red. Actualmente se soportan websocket y ROS/ROS2, también se encuentran disponibles librerías en los lenguajes de programación Python y Javascript. De esta forma, Robobo ya se puede programar

en la mayoría de los lenguajes existentes, y queda preparado para añadir soporte a otros nuevos de manera sencilla.

3.2 Software de simulación

CoppeliaSim, anteriormente conocido como Virtual Robot Experimentation Platform (V-REP), es un software de simulación en 3D propiedad de Coppelia Robotics. Cuenta con un entorno de desarrollo integrado que permite modelar, editar, programar y simular cualquier robot o sistema robótico. Se basa en una arquitectura de control distribuido, es decir, cada objeto/modelo se puede controlar individualmente a través de un script incrustado, complemento (plugins y add-ons), nodo ROS o BlueZero, un cliente API remoto o una solución personalizada. Esto hace que CoppeliaSim sea muy versátil e ideal para aplicaciones de múltiples robots. Los controladores se pueden escribir en C/C++, Python, Java, Lua, Matlab u Octave.

Este software es compatible con Windows, Linux y macOS, además de que cuenta con tres versiones: CoppeliaSim Pro, que es la versión comercial; CoppeliaSim Player, versión gratuita para ejecutar simulaciones, pero capacidades limitadas de edición; y CoppeliaSim Edu, con todas las características y gratuita para uso educativo.

Coppelia Robotics ofrece en su página web una gran cantidad de información [26] acerca de este software para ayudar a los usuarios a iniciarse en el mismo. Se puede encontrar el manual de usuario, tutoriales, vídeos y modelos de robots realizados por otros usuarios o empresas que dejan acceder a ellos libremente. Además, existe un foro donde las dudas de los usuarios son contestadas bien por otros usuarios o por técnicos de la empresa.

En los subapartados siguientes se describirán los aspectos más importantes del software de simulación para dar una idea del funcionamiento e interfaz del mismo, así como de los elementos que componen un modelo y una escena. Estos subapartados servirán de introducción para una mejor comprensión de las partes que componen el modelo en simulación del Robobo, e igual de importante, del desarrollo del entorno urbano sobre el que circulará el robot.

3.2.1 Interfaz de usuario

La aplicación CoppeliaSim está compuesta por varias ventanas, las cuales son:

- 1) Ventana de consola: la ventana de consola o terminal muestra qué complementos se cargaron y si su procedimiento de inicialización fue exitoso. La ventana de la consola no es interactiva y solo se usa para generar información. El usuario puede enviar información a la ventana de la consola con el comando *print* de Lua (desde un script), *printf* en C u otros comandos desde un complemento. Además de eso, el usuario puede crear ventanas de consola auxiliares para mostrar información específica de una simulación.
- 2) Ventana de aplicación: es la ventana principal de la aplicación. Se utiliza para mostrar, editar, simular e interactuar con una escena. Los botones izquierdo y derecho del ratón, la rueda del ratón y el teclado tienen funciones específicas cuando se activan en la ventana de aplicación.
- 3) Cuadros de diálogo: al lado de la ventana de aplicación, el usuario también puede editar e interactuar con una escena ajustando la configuración o los parámetros de del cuadro de diálogo. Existen múltiples cuadros de diálogo, cada uno agrupa un conjunto de funciones relacionadas, o que se aplican a un mismo objeto.

Cuando se inicia la aplicación CoppeliaSim, esta abrirá una escena predeterminada. El usuario puede abrir varias escenas en paralelo. En la siguiente sección se dará una breve descripción de los elementos de la ventana de aplicación, los cuales son:

- 4) Barra de aplicación: muestra información del tipo de licencia de CoppeliaSim, el nombre de archivo de la escena que se muestra actualmente, el tiempo utilizado para un pase de renderizado y el estado actual del simulador (simulación o modo edición).
- 5) Barra de menú: permite acceder a casi todas las funcionalidades del simulador.
- 6) Barras de herramientas: presentan funciones a las que se accede con frecuencia (desplazar, rotar y acercar/alejar la cámara, desplazar y rotar objetos, seleccionar y configurar el motor físico, comenzar/pausar la simulación, etc.). Las barras de herramientas se pueden desacoplar.
- 7) Navegador de modelos: es un explorador de archivos que te permite acceder a modelos ya guardados, los cuales están organizados por carpetas y tienen miniaturas para facilitar la “navegación” y localización de los modelos. Las miniaturas se pueden arrastrar y soltar en la escena para cargar automáticamente el modelo relacionado. Las miniaturas arrastradas aparecen oscuras si el área de colocación no es compatible o no es adecuada.
- 8) Jerarquía de escena: muestra todo el contenido de una escena, es decir, todos los modelos, sus componentes y la relación entre ellos, o elementos independientes como suelo, luces, etc. Desde ella se pueden editar los distintos elementos la escena.

Existen otros elementos en pantalla en los cuales no nos vamos a parar a explicar por ser demasiado obvios o por no emplearse durante el transcurso de este proyecto, pero que pueden ser consultados en el manual de usuario. Estos elementos son: páginas, vistas, texto informativo, barra de estado, entrada de comandos (en Lua), interfaz de usuario personalizada y menú emergente.

Para mayor aclaración, los distintos elementos mencionados aparecen indicados en las Figuras 4-6 y 4-7:

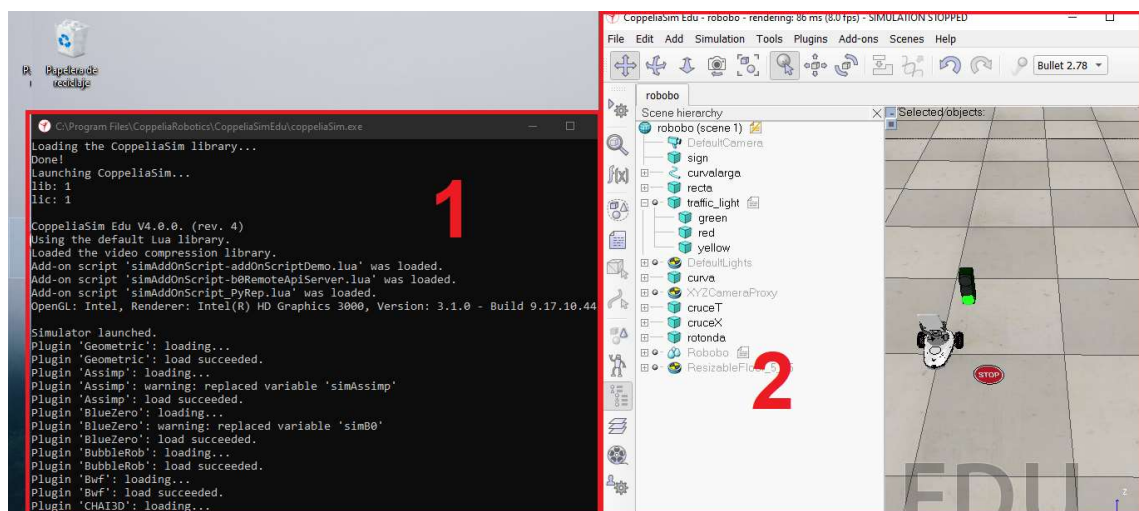


Figura 3-6 Ventanas de consola y aplicación de CoppeliaSim

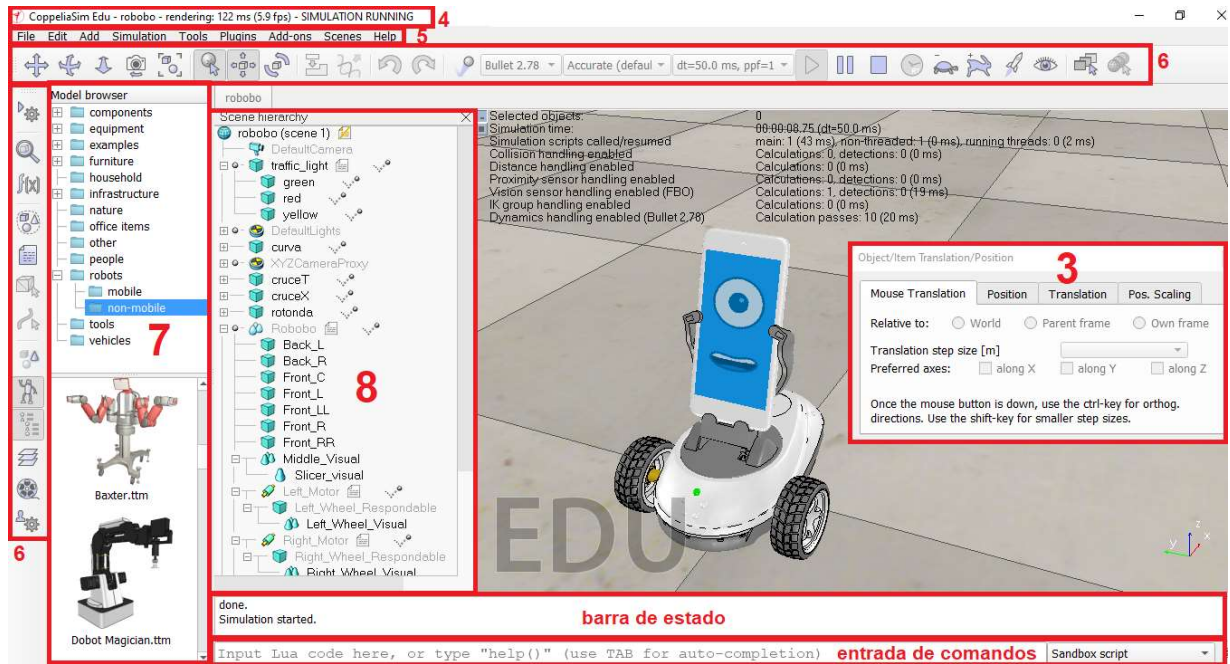


Figura 3-7 Detalle de ventana de aplicación

3.2.2 Escenas, modelos y objetos

Las escenas y los modelos son los elementos principales de una simulación. Un modelo es un elemento de una escena, y esta puede contener un número ilimitado de modelos. Ambos, escenas y modelos, están formados por uno o varios objetos y scripts. Además de eso, una escena siempre contendrá el entorno y un script principal.

Como se acaba de decir, los objetos son los elementos básicos necesarios para construir una escena. Encontraremos los diferentes tipos de objetos disponibles en la Figura 3-8:

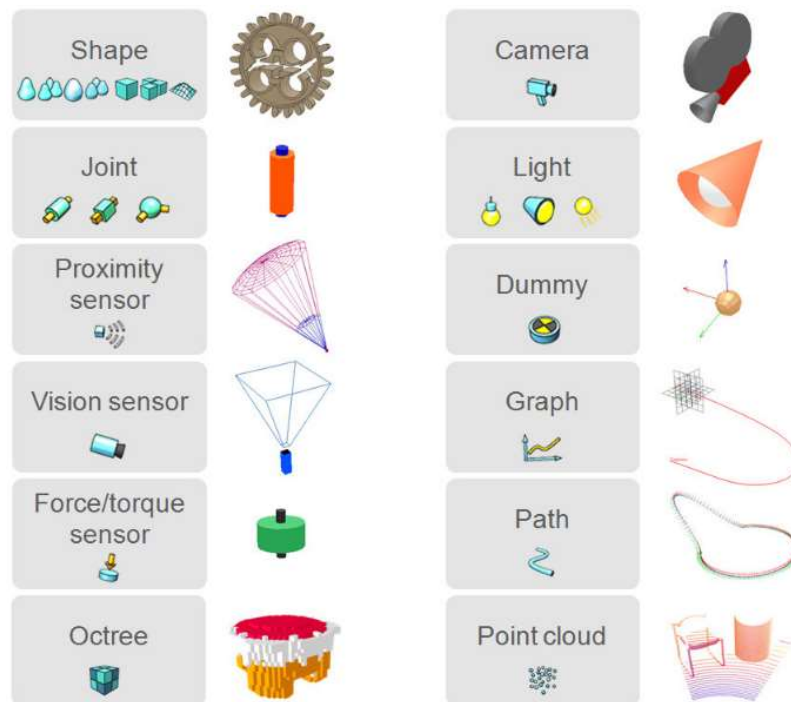


Figura 3-8 Tipos de objetos de escena

Por lo general (algunos objetos tienen propiedades especiales), los objetos se pueden editar para que puedan (o no) colisionar con otros objetos, se pueda medir la distancia mínima entre ellos, sean detectables por sensores de proximidad y/o de visión y ser visibles o invisibles para las cámaras de la simulación.

Los objetos más importantes para nosotros, por ser los empleados en la realización de la simulación de este proyecto serán las formas (*shapes*), articulaciones (*joints*), sensores de proximidad y sensores de visión, por eso se explicarán detalladamente a continuación.

3.2.2.1 Formas (Shapes)

Las *shapes* son mallas rígidas compuestas por caras triangulares y conforman la estructura física de cualquier modelo, es decir, su forma. Se pueden importar, exportar y editar. Existen cuatro tipos diferentes: formas puras, formas convexas, formas aleatorias y formas tipo *heighfield*, estas últimas no las explicaremos por no ser relevantes en el trabajo. Por defecto todas las formas importadas son simples, y estas se pueden agrupar o dividir.

Las formas puras representan formas primitivas (como un cubo, un cilindro o una esfera). Una forma pura es más adecuada para el cálculo dinámico de la respuesta ante colisión que el resto de las formas. Pueden ser simples o compuestas dependiendo de si el objeto está formado por una sola malla o por varias mallas agrupadas. El icono de este tipo de formas se muestra en la Figura 3-9.



Figura 3-9 Icono de las formas puras

Las formas aleatorias pueden representar cualquier malla, y son importadas desde un software de diseño 3D. No están optimizadas ni recomendadas para el cálculo dinámico de la respuesta ante colisión. También pueden ser simples o compuestas, y por tanto tener varios colores o atributos visuales. El icono de este tipo de formas se muestra en la Figura 3-10.



Figura 3-10 Icono de las formas aleatorias

El tercer tipo son las formas convexas, que son mallas importadas como las aleatorias pero optimizadas para el cálculo dinámico de la respuesta ante colisión (aunque se emplearán formas puras siempre que se pueda). También pueden ser simples o compuestas. El icono de este tipo de formas se muestra en la Figura 3-11.



Figura 3-11 Icono de las formas convexas

Cuando creamos un nuevo modelo, es muy recomendable el uso de dos “cuerpos” para el mismo. En una capa oculta se situará un primer cuerpo que será el empleado para el cálculo dinámico y de colisiones, y por tanto compuesto por formas puras y convexas. Mientras tanto, en la capa visible tendremos un cuerpo formado por *random shapes* o formas aleatorias que, al no estar optimizadas para este tipo de cálculos dinámicos, se relegan a una función únicamente visual, dotando de mayor realismo a la simulación. De vuelta a la capa oculta, debemos buscar una solución de compromiso, pues nos conviene aproximarnos lo máximo posible a la forma visual, pero cuanto más simplifiquemos estas formas usadas en los cálculos, menos lenta e inestable será la simulación.

Para ilustrar esto último, en la Figura 3-12 se muestran la capa visible y la capa oculta del modelo Robobo.

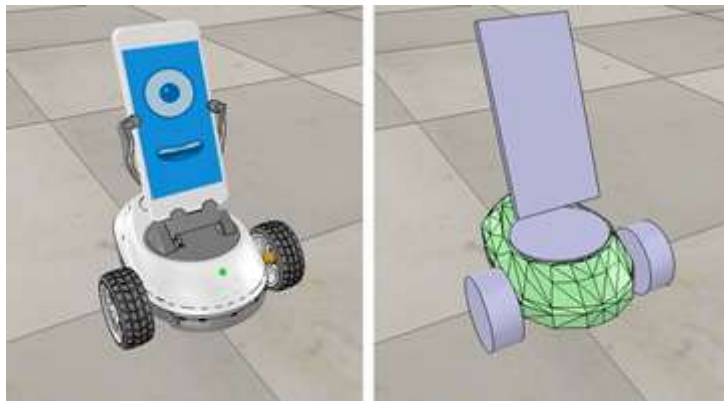


Figura 3-12 Capa visible y capa oculta del Robobo

3.2.2.2 Articulaciones (Joints)

Las articulaciones o *joints* son objetos que poseen como mínimo un grado de libertad, estos conectan a dos objetos entre sí permitiendo el movimiento relativo de uno respecto al otro. Esto es, básicamente, lo que en ingeniería mecánica se denomina par cinemático.

Existen cuatro tipos de pares en el programa:

- Par de revolución: posee un grado de libertad y permite el giro de dos objetos alrededor de un eje. Puede entenderse como una bisagra.
- Par prismático: posee un grado de libertad y permite la traslación entre dos objetos. Puede entenderse como una corredera rectangular.
- Par helicoidal: puede entenderse como un tornillo. Es una combinación del par de revolución y el prismático. Tiene un grado de libertad, es necesario definir un valor que represente la cantidad de traslación que genera una cantidad de rotación determinada.
- Par esférico o rótula: tiene tres grados de libertad y permite rotaciones entre dos objetos.

Las tres primeras pueden ser usadas como articulaciones pasivas o articulaciones activas (motores). Los pares esféricos son siempre pasivos y no pueden actuar como motores. Todos los tipos mencionados aparecen representados en la Figura 3-13.

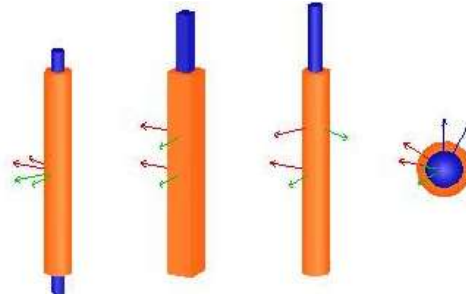


Figura 3-13 Par de revolución, prismático, helicoidal y rótula

3.2.2.3 Sensores (Sensors)

Tenemos disponibles tres tipos de sensores en el simulador: sensores de proximidad, sensores de visión y sensores de fuerza. El más importante para este trabajo será el sensor de visión, no debe confundirse con una cámara (que también es un tipo de objeto seleccionable en el simulador). Los sensores de visión se diferencian de las cámaras en que permiten seleccionar la resolución, el contenido de la imagen es accesible mediante una API y sólo muestra objetos que hemos marcado como representables. Como contrapartida, el sensor de visión tiene un mayor uso de CPU que una cámara.

Como vemos en la Figura 4-14, hay dos tipos de sensores de visión: de proyección ortogonal y de perspectiva. El tipo de proyección en perspectiva es el adecuado para simular cámaras y será el que emplearemos.

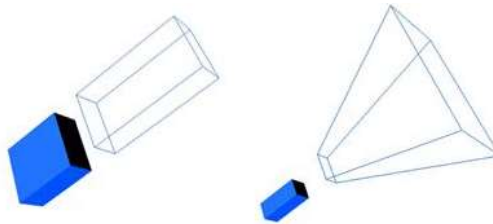


Figura 3-14 Proyección ortogonal y proyección en perspectiva

Una vez presentados todos los objetos de escena a utilizar, el modelo del Robobo simplemente es una representación digital formada por *shapes* para dar forma a los elementos, joints que determinen los movimientos entre elementos, sensores de proximidad, y sensores de visión para las cámaras del smartphone. Para que el modelo se comporte como un único objeto es necesario establecer una jerarquía, se escoge un elemento como base y los demás elementos siguen un criterio de subordinación.

3.2.3 Programación en Coppeliasim

Para programar el comportamiento de un modelo o escena, Coppeliasim nos ofrece seis alternativas distintas: mediante scripts, add-on, plugin, API externa, nodo de ROS o nodo de BlueZero. Cada uno tiene sus ventajas particulares, y evidentemente también desventajas. En este trabajo se han empleado scripts y una API externa.

Los scripts son programados en Lua [27], un lenguaje de programación libre y gratuito además de muy compacto, lo que lo vuelve ideal para scripts. Estos se pueden dividir en scripts de simulación, que como su propio nombre indica se ejecutan durante la simulación, y scripts de personalización, que se ejecutan con la simulación parada. En los scripts de simulación el más importante es el main script, que es el que se encarga de que la simulación funcione, por lo que ya viene por defecto y se recomienda

no modificar salvo si se es un usuario avanzado. Por lo tanto, lo recomendado es que todo lo que incluyamos no lo hagamos en el *main script* sino en *child scripts*, que son independientes y van asociados a realizar una parte específica de la simulación, bien del comportamiento de los robots, o bien del entorno.

Concretamente para nuestro caso, el Robobo tiene 4 scripts, uno asociado a la base y otro por cada motor (las dos ruedas y la unidad pan-tilt), los cuales contienen todo el código necesario para controlar remotamente el robot. El resto de los scripts añadidos durante el desarrollo del trabajo serán detalladamente comentados más adelante.

En cuanto a la API remota, CoppeliaSim nos ofrece dos versiones. La nueva versión B0 es más fácil de usar y permite mayor flexibilidad que la API remota antigua. Nosotros utilizaremos la más antigua por ser comparativamente más liviana y con menos dependencias. Esta API está disponible para varios lenguajes de programación: C/C++, Python, Java, Matlab/Octave o Lúa. La API remota nos permite tener dos entidades separadas, el lado del servidor (CoppeliaSim) y el lado del cliente (la aplicación que escojamos) que pueden interactuar de forma síncrona o asíncrona. El principal inconveniente es que hay que tener cuidado con el retardo de comunicación entre ambas entidades, este efecto no pasa cuando se emplean scripts, lo cual es una de sus mayores ventajas.

3.3 Software de visión

La visión artificial o visión por computación se puede abordar desde distintas perspectivas. Desde una perspectiva general, es un conjunto de métodos que se emplean para adquirir, procesar y analizar imágenes de forma que se extraiga de ellas información numérica que pueda ser tratada por un ordenador. Estos métodos llevan un largo recorrido en la industria tanto en control de calidad como en tareas que requieren suma precisión, como la medición y soldadura de piezas pequeñas. No obstante, lo más importante para este trabajo es la perspectiva de visión artificial como subcampo de la inteligencia artificial, donde estos métodos son empleados para permitir a un ordenador reconocer objetos, caras, extraer texto de una imagen, etc.

Para ello, en este TFG emplearemos dos herramientas: la librería OpenCV y la API de detección de objetos de Google, basada en TensorFlow.

3.3.1 OpenCV

Open Source Computer Vision Library (OpenCV), como su propio nombre indica, es una librería de código abierto sobre visión por computación, análisis de imagen y aprendizaje automático. Fue inicialmente desarrollada por Intel, que delegó el proyecto a una fundación sin ánimo de lucro. Gracias a la comunidad de usuarios continuó su desarrollo contando en la actualidad con más de 2500 algoritmos optimizados y extensa documentación explicada y actualizada, con ejemplos de uso de sus funciones, tutoriales, etc. Esto la convierte en la librería de visión artificial más popular del mundo [28].

Cuenta con interfaz para C++, Python, Java y MATLAB y soporta Windows, Linux, macOS y Android. Esto se traduce en que a la hora de emplear el Robobo real, las funciones de esta librería pueden ejecutarse directamente desde el smartphone o remotamente enviando la imagen del smartphone al ordenador/tablet y procesar la imagen desde este último.

En el momento de redactar este Trabajo Fin de Grado, OpenCV se encuentra en la versión 4.4.0 aunque para la realización de este se ha empleado la versión opencv-python 4.2.0.32.

3.3.1 *TensorFlow*

TensorFlow (TF) es una librería desarrollada por Google y publicada en 2015 bajo licencia de código abierto. Creada para facilitar el desarrollo y entrenamiento de modelos de aprendizaje automático [29].

Se puede ejecutar desde un ordenador, tablet, smartphone, microcontrolador o aplicación web, por lo que al igual que OpenCV, a la hora de probar los algoritmos con el robot real, estos se podrán adaptar para su ejecución tanto en remoto desde el ordenador como desde el mismo smartphone. Además, en caso de ejecutarla desde un PC, la librería puede aprovechar la potencia de procesamiento de la tarjeta gráfica aumentando significativamente el rendimiento de los modelos.

Conviene también mencionar dos herramientas íntimamente relacionadas con TF: TensorFlow Lite, framework que permite la ejecución de TF en dispositivos móviles y de IoT (previa conversión y optimizado del modelo); y Keras, la API de alto nivel de TensorFlow.

En el momento de redactar este Trabajo Fin de Grado, TensorFlow se encuentra en la versión 2.3.0, pero por razones de compatibilidad con la API de detección de objetos, a lo largo de este proyecto emplearemos la versión 1.15

4 DISEÑO DEL SISTEMA

4.1 Modelo 3D Robobo

Como ya se ha comentado, en este proyecto partimos de un modelo existente en CoppeliaSim del Robobo. La apariencia y el comportamiento de los motores y sensores infrarrojos es similar a la del robot real, pero no se había realizado ninguna actividad que implicase el uso de la cámara del smartphone, por lo que es necesaria una mejora del modelo.

También comentamos en el capítulo anterior que para simular una cámara en CoppeliaSim se emplea un sensor de visión. A este sensor se le ha establecido una resolución de 512x512p, suficiente para obtener una imagen nítida pero no tan grande como para ralentizar la simulación. A la hora de pasar del modelo en simulación al robot real habrá que tener en cuenta tres consideraciones respecto a lo anterior:

- La ubicación de la cámara: varía de un móvil a otro, sabemos que la cámara delantera se encuentra en la parte superior de la pantalla del teléfono, pero dependiendo del modelo puede estar situada a la derecha, izquierda o centro, por lo que puede variar la perspectiva de la imagen obtenida. Por defecto nosotros hemos colocado la cámara centrada.
- La resolución: como hemos dicho, para no sobrecargar la simulación hemos escogido una resolución más bien baja y que nos devuelve una imagen cuadrada. Actualmente un smartphone de gama media/baja suele contar con una cámara frontal de 8MP, es decir, podríamos encontrarnos con que la cámara nos devuelva una imagen de 3264x2448p y relación de aspecto 4:3.
- La iluminación: en un entorno real la iluminación y por tanto los colores de la imagen obtenida varían de un momento a otro del día y de una cámara a otra.

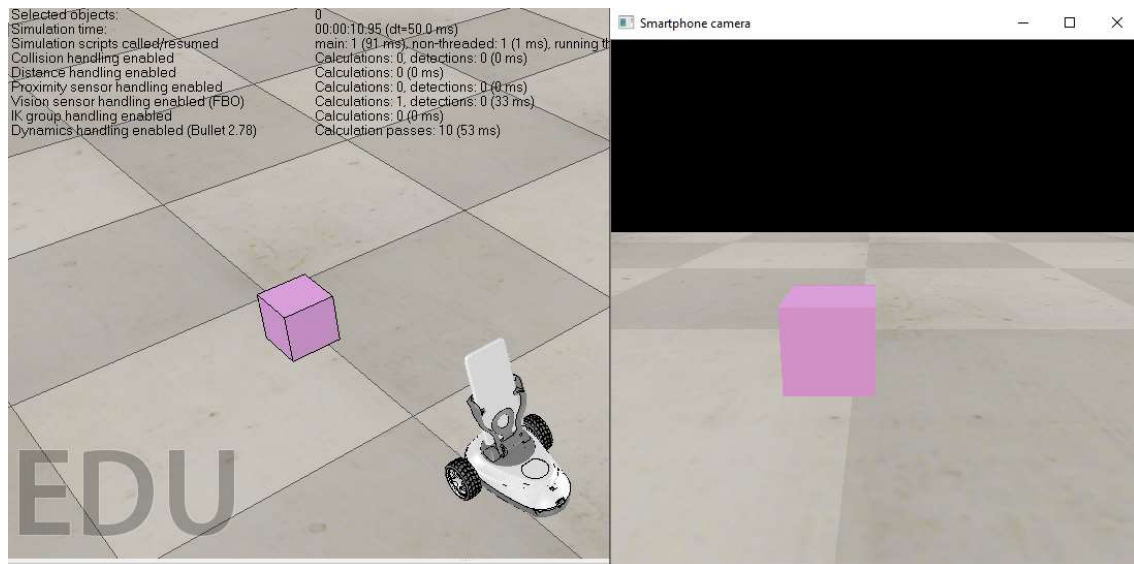


Figura 4-1 Modelo de Robobo con cámara incorporada

Para programar en Python desde la API externa de la misma manera que lo haríamos con un Robobo real necesitamos, entre otros, un fichero en nuestra carpeta de proyecto llamado *RoboboVREP.py* que contiene todas las definiciones de las funciones que se pueden emplear. A continuación de haber “instalado” nuestra cámara en el simulador, debemos añadir una nueva función que nos permita acceder a la imagen externamente.

El código del fichero actualizado con esta nueva funcionalidad se encuentra en el Anexo 1 de la memoria. Como se puede ver en el anexo finalmente son cuatro las funciones añadidas, así pues, se puede escoger entre extraer la imagen en color o en blanco y negro. Cada una de estas opciones requiere de dos funciones, una que se debe ejecutar una sola vez al comienzo del programa (*startColorImage* o *startGrayImage*) y otra que se ejecuta cada vez que se quiera obtener la imagen (*getColorImage* o *getGrayImage*).

La opción de extraer la imagen en blanco y negro del programa, aunque visualmente es menos atractiva, mejora notablemente el rendimiento de los scripts en los que utilicemos estas funciones. Esto se debe a que en informática las imágenes están compuestas por canales, esto es, una imagen del mismo tamaño que la original donde cada píxel contiene un valor numérico, entre 0 y 255 para el estándar de 8 bits por píxel (bpp). Una imagen RGB (color) tiene tres canales: rojo, verde y azul. Mientras tanto una imagen en blanco y negro solo emplea un canal (escala de grises) por lo que en comparación la misma imagen en color ocuparía el triple de memoria y sería más pesada de manejar por la CPU.

4.2 Modelado del entorno

Ahora que tenemos el modelo del Robobo preparado, necesitamos dar forma a un entorno donde el robot pueda desenvolverse y podamos desarrollar nuestros algoritmos de conducción autónoma. CoppeliaSim facilita la recreación de un entorno realista de un laboratorio, al ofrecernos un suelo de baldosas reescalable y modelos en 3D de mobiliario (armarios, estanterías, mesas, sillas, etc.), paredes, ventanas, puertas, objetos de oficina o personas. Sin embargo, debido a lo específico que es el enfoque de este proyecto, debemos crear nosotros mismos objetos que simulen un entorno urbano pero adaptados a las dimensiones de nuestro robot. A continuación, dejaremos de lado las partes del entorno que son meramente ornamentales como edificios, parques y demás que su única función es dotar de realismo a la simulación. En su lugar nos centraremos en las partes que afecten a la circulación del robot.

4.2.1 Señalización horizontal

La señalización horizontal corresponde a la aplicación de marcas viales, conformadas por líneas, flechas, símbolos y letras que se pintan sobre el pavimento. Se incluyen en este grupo las líneas longitudinales continuas y discontinuas (separación de carriles), líneas transversales continuas y discontinuas (líneas de detención y pasos de peatones), flechas, inscripciones y otras marcas (como el cebreado). Las marcas viales pueden ser de varios colores, siendo en general de color blanco, amarillas en obras o zonas con prohibición de parada, y azules o verdes en zonas de estacionamiento limitado.

Las líneas longitudinales que separan los carriles en nuestra simulación tienen una anchura de 2 cm, con una separación entre sí (es decir, con un ancho de carril) de 27,5 cm. Para facilitar las futuras ampliaciones o modificaciones en el trazado, se ha dividido el mismo en varios segmentos como puede observarse en la Figura 4-2. Contamos por tanto con una recta, una curva de 90°, un cruce en T y un cruce en X. Para formar el trazado deseado, estos elementos pueden ser fácilmente copiados, trasladados y rotados de forma que se colocan unos a continuación de otros y se añaden el resto de las marcas viales que se estimen oportunas.

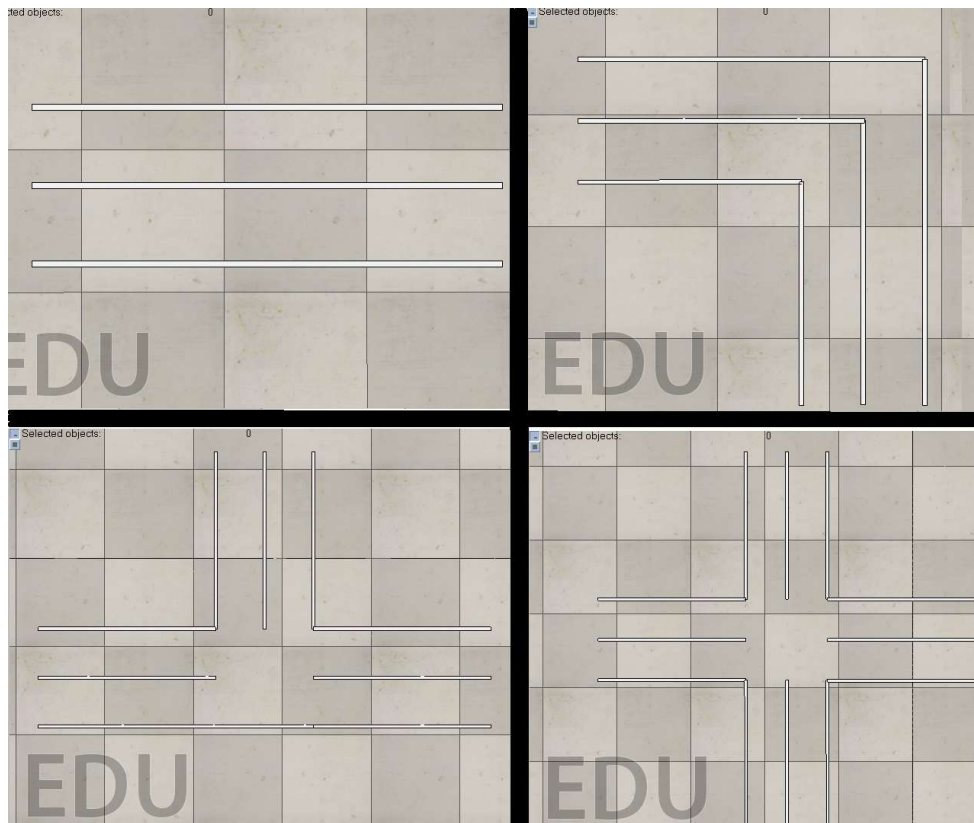


Figura 4-2 Segmentos independientes del trazado

4.2.2 Señalización vertical

Las señales verticales son tableros fijados en postes o estructuras colocados adyacentes a la vía, y su función es informar y advertir a los usuarios de esta. Debido a su importancia en la seguridad existe un acuerdo internacional para mantener el mismo formato de señalización vertical en toda la Unión Europea. Este formato, que será el que usemos en simulación, divide las señales en: señales de advertencia de peligro, se reconocen por su forma triangular; señales de reglamentación, son por lo general redondas e indican prohibiciones, obligaciones o restricciones sobre la vía; señales de indicación, son rectangulares y contienen diversa información de interés para el conductor.

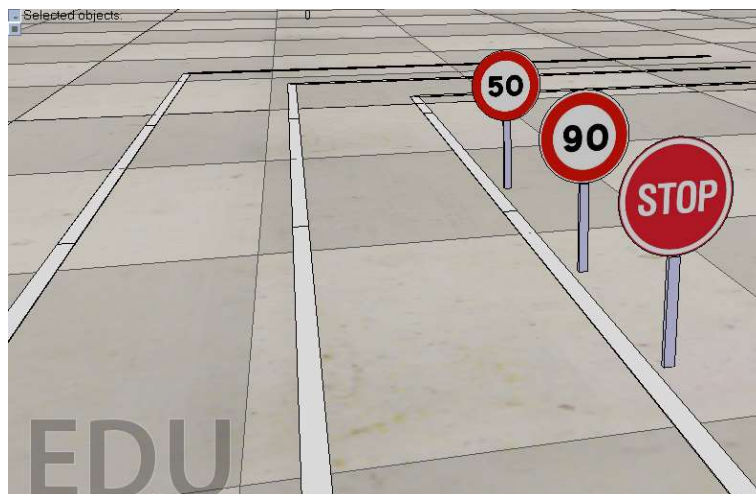


Figura 4-3 Señales de reglamentación en CoppeliaSim

Se puede representar de forma realista en CoppeliaSim cualquiera de las señales existentes en el código de circulación gracias a la opción de añadir texturas a objetos. En el caso de señales como las de la Figura 5-3 la textura se aplica sobre un disco plano de 10 cm de diámetro, con una altura total desde el pie hasta la parte más alta del disco de 20 cm.

En este apartado incluimos también los semáforos, que se puede ver en la Figura 5-4. Estos elementos son más complicados, comparados con las señales u otros objetos de la escena, debido a que su estado es variable con el tiempo. Por lo tanto, es necesario asociarles un script que defina su comportamiento durante la simulación. Este script, programado en Lua, se encuentra en el Anexo 2 de la memoria, por si se desea analizar los detalles del funcionamiento del semáforo.

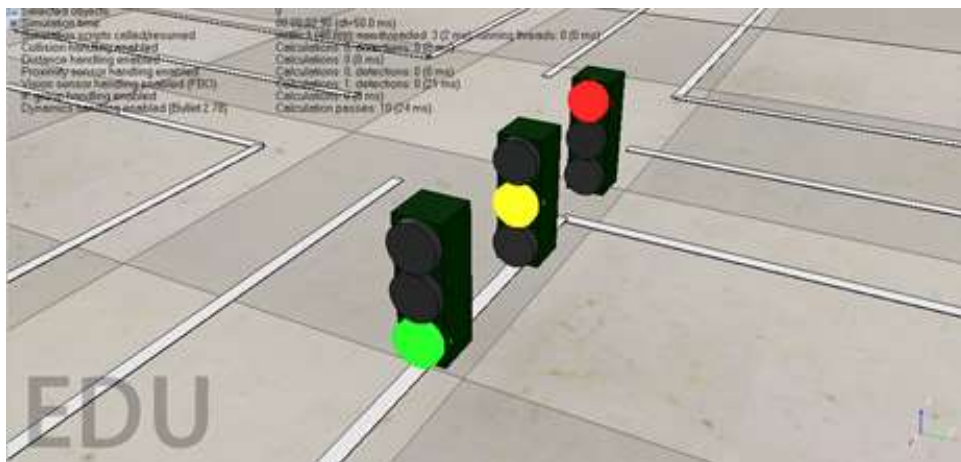


Figura 4-4 Semáforos en CoppeliaSim

5 IMPLEMENTACIÓN DEL SISTEMA

Una vez finalizado el diseño del modelo del robot y de su entorno, en este capítulo se explicarán los pasos necesarios para la implementación completa en el modelo de los algoritmos de conducción autónoma.

5.1 Detección de carriles

Las líneas trazadas en las carreteras indican a los conductores humanos la ubicación de los carriles y actúan como guía de referencia sobre la dirección que debe seguir el vehículo, acorde a las reglas de tráfico y con una interacción armoniosa con la carretera y los elementos presentes en ella. De esta manera, la capacidad de identificar y rastrear carriles es fundamental para desarrollar algoritmos en vehículos sin conductor [30].

La mayoría de los carriles del mundo real están diseñados para ser relativamente sencillos, con las mínimas perturbaciones para facilitar la conducción y mantener en la medida de lo posible una velocidad constante, tal y como indica la normativa [31]. El objetivo es poder detectar las líneas del suelo en la imagen de la cámara del Robobo y poder así mantener la plataforma robótica dentro de un carril especialmente diseñado para las dimensiones del robot. Por lo tanto, nuestro enfoque será el de, con la imagen obtenida por la cámara, detectar líneas rectas prominentes mediante técnicas de detección de bordes y extracción de características.

A continuación, explicaremos las técnicas empleadas, donde destacan el algoritmo de Canny y la transformada de Hough. Para aplicar estas técnicas emplearemos varias funciones de la librería OpenCV.

5.1.1 Algoritmo de Canny

El algoritmo de Canny es un operador desarrollado por John F. Canny en 1986 que utiliza un algoritmo de múltiples etapas para detectar bordes en imágenes [32]. El objetivo fundamental del algoritmo es detectar cambios bruscos en la luminosidad (grandes gradientes, como un cambio de negro a blanco) y los define como bordes o no en función de un conjunto de umbrales dado. Las cuatro etapas principales del algoritmo de Canny son:

- 1- Reducción de ruido digital. Como en gran cantidad de mediciones, el ruido es un problema crucial que a menudo conduce a una falsa detección. Para suavizar la imagen y así disminuir la sensibilidad del detector al ruido se aplica un filtro gaussiano de 5x5, configurando así el valor de cada píxel al promedio ponderado de sus píxeles vecinos y dando una apariencia de emborronado.
- 2- A la imagen suavizada se le aplica luego un operador Sobel, Roberts o Prewitt (OpenCV utiliza Sobel) a lo largo de los ejes x e y para detectar si los bordes son horizontales, verticales o diagonales. Para cada píxel de la imagen a procesar, el resultado del operador Sobel es tanto el vector gradiente correspondiente como la norma de este vector.
- 3- Para afinar los bordes de manera efectiva se comprueba para cada píxel si su valor es un máximo local en la dirección del gradiente calculado en el paso anterior.
- 4- Después del paso anterior, los píxeles “fuertes” (valores máximos locales) se encuentran en el mapa final de bordes. Sin embargo, los píxeles “débiles” deben analizarse más a fondo para determinar si se constituyen como borde o ruido. Aplicando dos valores umbral predefinidos (minVal y maxVal), establecemos que cualquier píxel con gradiente de intensidad superior a maxVal son bordes y

cualquier píxel con gradiente inferior a minVal se descarta. Los píxeles con gradiente de intensidad entre minVal y maxVal solo se consideran bordes si están conectados a un píxel con gradiente de intensidad por encima de maxVal .

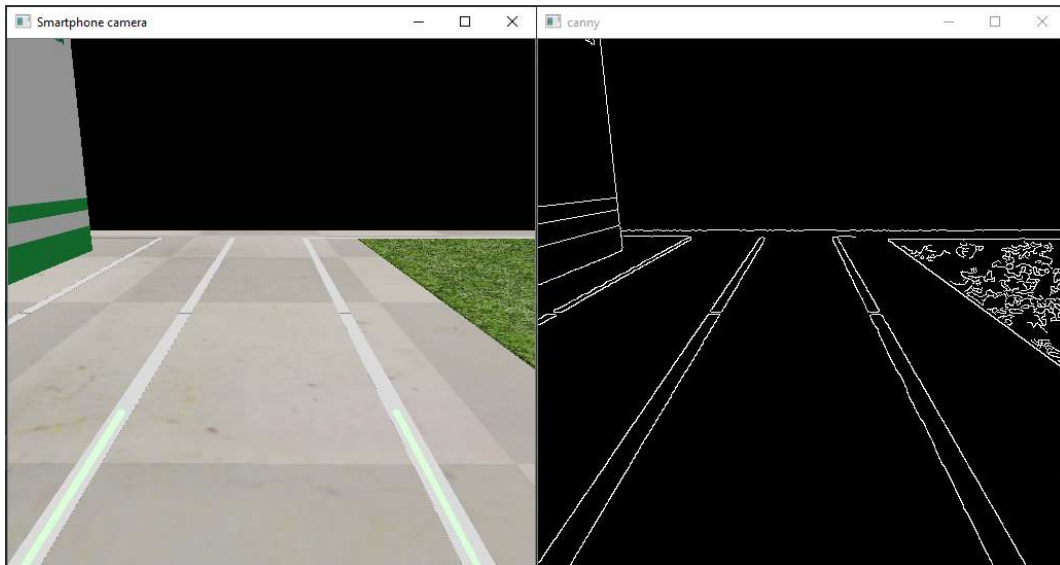


Figura 5-1 Efecto del algoritmo de Canny sobre una imagen

Como se puede observar en la Figura 5-5 el resultado es muy bueno, el filtro descarta los cambios de color de las baldosas del suelo, pero reconoce perfectamente las líneas relevantes. A continuación, vamos a crear una máscara triangular para segmentar el área del carril y descartar las áreas irrelevantes en la imagen para aumentar la efectividad de las etapas posteriores. Los vértices del triángulo vienen marcados por tres coordenadas, indicadas por los círculos rojos en la Figura 5-2.

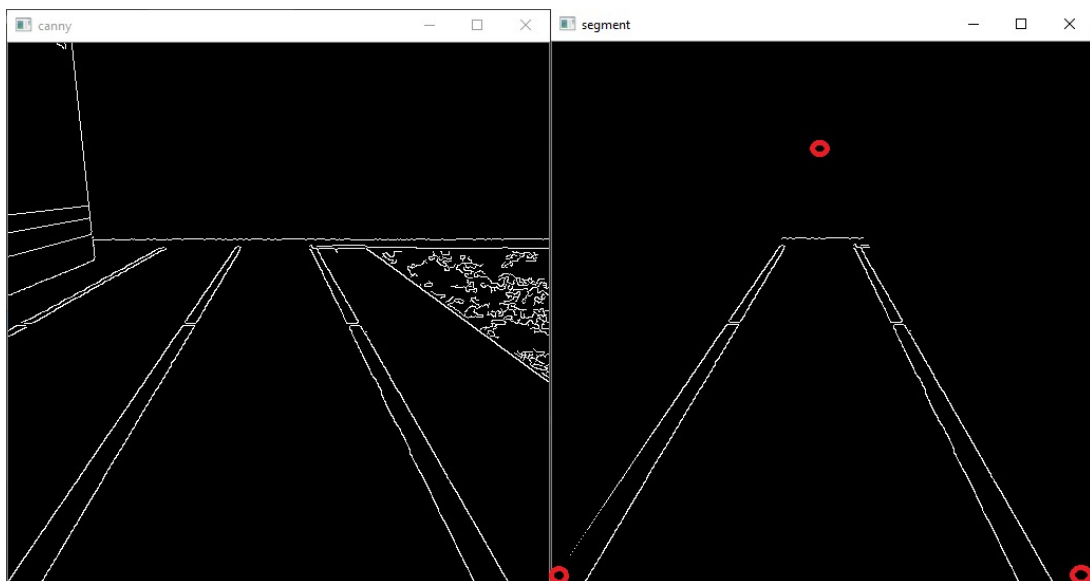


Figura 5-2 Aplicación de una máscara triangular a la imagen

5.1.2 Transformada de Hough

En el sistema de coordenadas cartesianas, se puede representar una línea recta como: $y = mx + b$ trazando y frente a x . Sin embargo, también se puede representar esta línea como un único punto en el espacio de Hough representando b frente a m . Por ejemplo, una recta $y = 2x + 1$ se representa en el espacio de Hough por el punto $(2, 1)$.

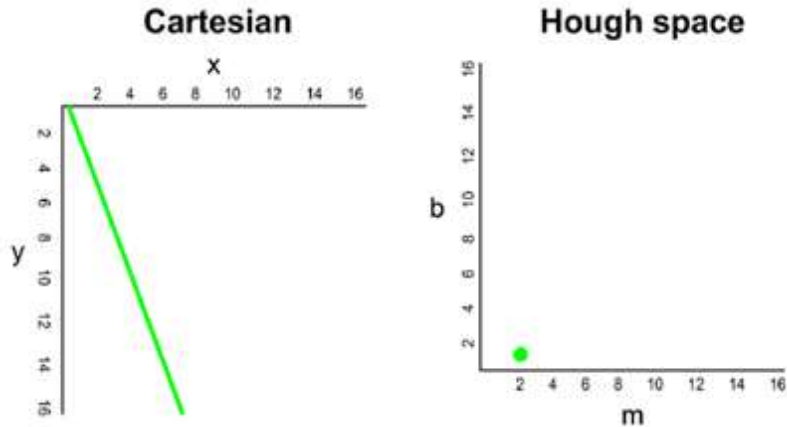


Figura 5-3 Recta representada por un punto

Ahora, ¿qué pasaría si en lugar de una recta tuviésemos que trazar un punto en el sistema de coordenadas cartesianas? Hay muchas rectas posibles que pueden pasar por este punto, cada recta con unos valores diferentes para los parámetros m y b . Por lo tanto, un punto en coordenadas cartesianas produce una recta en el espacio de Hough. Por ejemplo, el punto $(2, 12)$ en coordenadas cartesianas puede ser atravesado por las rectas $y = 2x + 8$, $y = 3x + 6$, $y = 4x + 4$, $y = 5x + 2$, $y = 6x$, etc. Estas rectas posibles se representan en el espacio de Hough por los puntos siguientes: $(2, 8)$, $(3, 6)$, $(4, 4)$, $(5, 2)$ y $(6, 0)$. La unión de estos puntos forma una recta, como se puede ver en la Figura 5-4:

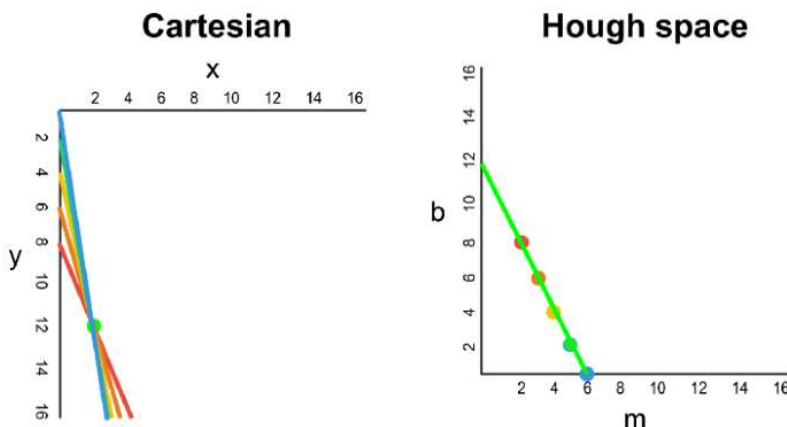


Figura 5-4 Punto representado por una recta

Cada vez que vemos una serie de puntos en un sistema de coordenadas cartesianas y sabemos que estos puntos están conectados por alguna línea, se puede encontrar la ecuación de esa línea trazando primero para cada punto del sistema de coordenadas cartesianas su línea correspondiente en el espacio de Hough. El punto de intersección de esas líneas en el espacio de Hough representa los valores de m y b de la recta que pasa consistentemente por todos los puntos de la serie.

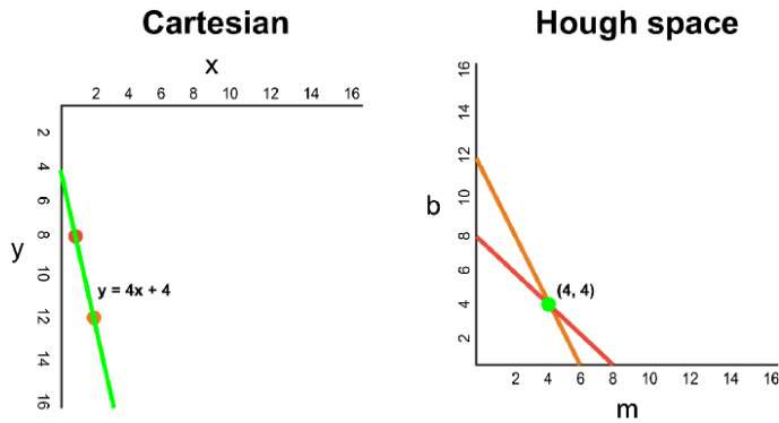


Figura 5-5 Ecuación de la recta a partir de punto de corte

Hemos utilizado coordenadas cartesianas para simplificar la explicación, sin embargo, hay un efecto matemático con este enfoque: cuando la línea es vertical, el gradiente es infinito y no se puede representar en el espacio de Hough. Para resolver este problema, se utilizan coordenadas polares en su lugar. Con este cambio, el proceso sigue siendo el mismo que se ha comentado, pero en vez de trazar m respecto a b , será r respecto a θ .

Generalmente, a mayor cantidad de rectas que se cruzan en el espacio de Hough significa que esa intersección corresponde a más puntos. Para nuestra implementación definiremos un umbral mínimo de intersecciones en el espacio de Hough para detectar una línea. Por lo tanto, la transformación de Hough básicamente realiza un seguimiento de las intersecciones en cada punto del espacio, si es número de intersecciones excede un umbral definido, identificamos esa línea con sus parámetros correspondientes.

5.1.3 Visualización

Por lo tanto, y gracias al sistema explicado apartado anterior, es posible obtener la ecuación de dos líneas rectas, que serán nuestros límites de carril izquierdo y derecho, y posteriormente representarlas (como se puede ver en la Figura 5-10) y tomar decisiones sobre la dirección del robot acorde a esta información.

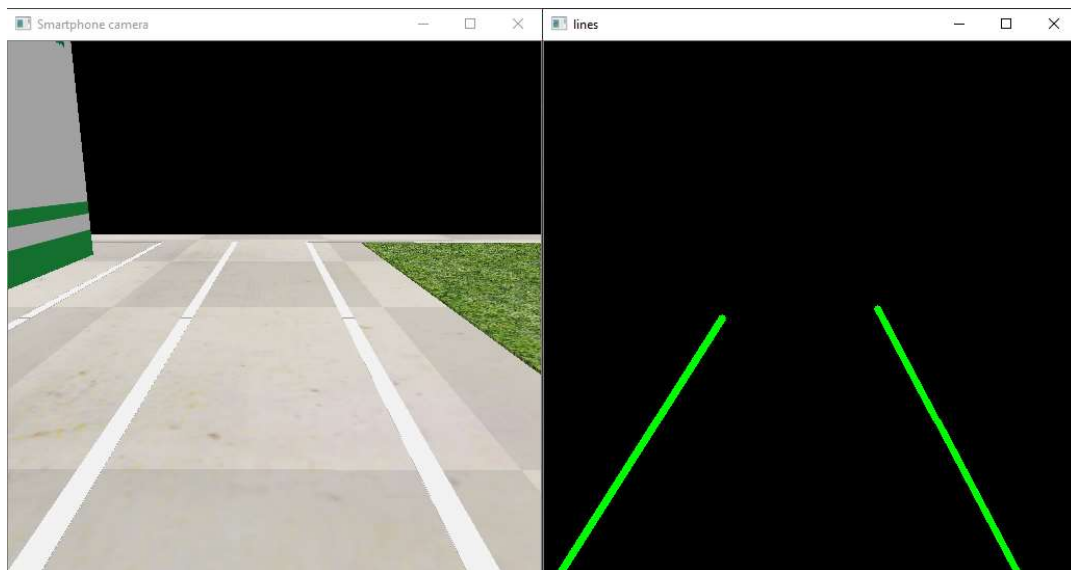


Figura 5-6 Líneas del carril detectadas en la imagen

Todo el código correspondiente al apartado 5.1 *Detección de carriles* se encuentra en el Anexo 3 de la memoria.

5.2 Detección de objetos

El resto de los elementos del entorno que un vehículo autónomo debe reconocer para poder operar correctamente, tales como señales de tráfico, semáforos y obstáculos como otros vehículos, se incluyen juntos en este apartado porque se reconocerán empleando la misma herramienta: la API de detección de objetos de TensorFlow. La misma dispone de un repositorio en GitHub desde la que puede descargarse [33].

Para usar esta herramienta primero debemos tenerla instalada y funcionando correctamente. Este paso, que puede parecer trivial y de hecho con muchos otros programas así es, se puede asegurar al lector que en este caso no lo es tanto, debido a las muchas dependencias que posee la API de detección de objetos. En el campo del software, se conoce como dependencia a una aplicación o una librería requerida por otro programa para poder funcionar correctamente. Pongamos un ejemplo práctico, una librería va a ejecutar una instrucción y para ello necesita un dato que aún no ha sido calculado y que es proporcionado por otra librería. Puede darse el caso de que esa segunda librería no esté instalada en nuestro ordenador (este sería el menor de nuestros problemas porque nos daríamos cuenta rápido del error), que la versión instalada no sea compatible, que la ruta seleccionada no sea correcta, y una larga lista de etcétera. Por experiencia personal durante la realización de este TFG, se recomienda para evitar los problemas mencionados crear un nuevo entorno virtual con Anaconda e instalar a mano todas las dependencias. Se ha encontrado un tutorial bastante completo (en inglés) para ayudarnos en esta tarea [34].

5.2.1 Selección del modelo

Aunque es posible crear desde cero una red neuronal convolucional para detección de objetos, para ahorrar tiempo y posibles complicaciones en el mismo repositorio de GitHub desde el que se descargamos la API nos ofrecen una colección de modelos de detección ya entrenados y testeados. Se probarán varios de estos modelos para escoger el más adecuado.

En la Tabla 5-1 enumeramos los modelos que se han probado, acompañados de dos parámetros: velocidad y precisión del modelo. La velocidad se refiere al tiempo de ejecución en milisegundos para evaluar una imagen de 600x600 píxeles (incluido todo el procesamiento previo y posterior). Se debe tener en cuenta que el tiempo de ejecución depende en gran medida de las características de hardware del que se disponga. Los tiempos marcados en la tabla junto con la puntuación de la precisión fueron proporcionados por el desarrollador de los modelos, empleando una tarjeta gráfica NVIDIA GeForce GTX TITAN X, por lo que es de esperar que durante el uso del simulador y en la posterior implementación de los algoritmos en el Robobo real, el tiempo de ejecución sea bastante superior. De todas formas, los datos de velocidad sirven para dar una idea de qué modelos son más rápidos y ayudarnos en la elección del modelo, cuanto menor sea el tiempo de procesamiento, mejor.

Para cuantificar el rendimiento del detector de objetos los desarrolladores han realizado numerosas pruebas [35] y emplean su propia unidad de medida [36] para expresar los resultados: *precisión promedio* o en inglés *mean Average Precision* (mAP). No entraremos a definir los experimentos ni como calculan la precisión promedio para no eternizar el apartado, lo que sí debemos saber es que al contrario de lo que ocurre con la columna de tiempos, aquí cuanto mayor sea el número, mejor.

Nombre del modelo	Velocidad	Precisión
	ms	mAP
ssd_mobilenet_v1_coco	30	21
ssd_mobilenet_v2_coco	31	22
ssdlite_mobilenet_v2_coco	27	22
ssd_mobilenet_v3_large_coco	119*	22.6
faster_rcnn_inception_v2_coco	58	28
faster_rcnn_resnet50_coco	89	30

*medido en smartphone Google Pixel 1

Tabla 5-1 Modelos de TensorFlow utilizados

Como se puede comprobar, los modelos más precisos lo son a costa de un mayor tiempo de procesamiento. Hay muchos más modelos que no se han probado porque, pese a ser muy precisos son demasiado lentos para un uso práctico en nuestro proyecto. En aplicaciones de detección en tiempo real, como es nuestro caso, se prioriza en cierta medida la velocidad sobre la precisión. Para ello, presentamos el siguiente ejemplo:

Imaginemos que vamos conduciendo y de pronto un árbol cae sobre el carril. Se ha estimado que el tiempo de reacción medio desde que el conductor detecta el obstáculo hasta que se comienza a frenar es de 0.75 segundos, oscilando entre 0.5 y 1 segundo debido a los múltiples factores que pueden influir en el estado psíquico y físico del conductor, como son: su edad, años de experiencia al volante, sueño, fatiga, estado de ánimo, consumo de alcohol y/o drogas, etc. Necesitamos por tanto que nuestro algoritmo sea capaz de procesar un mínimo de dos imágenes por segundo para lograr una mejora en la seguridad con respecto a un conductor humano.

Finalmente, nos decantaremos por el modelo *ssd_mobilenet_v3_large_coco*. Se ha elegido este modelo por ser la versión más reciente y con mayor precisión de los modelos *Single Shot Multibox Detector (SSD)* de *MobileNet*, que son los más rápidos, y porque está especialmente diseñado para ejecutarse desde un smartphone, por lo que no será necesario un nuevo modelo en futuros proyectos, donde se prueben los algoritmos de conducción autónoma ya en un Robobo real.

5.2.2 Entrenamiento del modelo

El modelo escogido está entrenado para reconocer objetos cotidianos tales como bicicletas, motos, coches, gatos, perros, personas o incluso tostadoras. Debemos por lo tanto deshacernos de las etiquetas que no necesitamos y añadir otras necesarias para conducción autónoma como señales y semáforos. Se puede “enseñar” al modelo a reconocer cualquier cosa, la única limitación importante como vamos a comprobar a lo largo de este apartado va a ser el tiempo.

5.2.2.1 Recopilar conjunto de datos

Una vez elegido el objeto u objetos que queremos reconocer, necesitamos recopilar una gran cantidad de imágenes por cada objeto deseado. Las imágenes deben representar un buen grado de variaciones, de modo que el detector final sea lo más robusto posible a la hora de detectar el objeto. Ahora bien, ¿Cuántas imágenes son “una gran cantidad de imágenes”? No es que se pretendiese dar una explicación vaga al lector, sino que no hay manera de saberlo con exactitud. Para las clases de objetos en las que viene entrenado el modelo descargado, sus desarrolladores emplearon un

conjunto de datos con varios miles de imágenes por clase [37], pero veremos una serie de simplificaciones por las cuales es posible reducir considerablemente esa cifra, hasta el orden de unas 100 imágenes por clase.

Como se ha dicho, los modelos que descargamos vienen entrenados con objetos cotidianos, que se pueden encontrar en cualquier parte y que por tanto tienen una alta variabilidad. Por ejemplo, un perro puede ser de una alta variedad de razas, tamaños, colores y estar colocado en infinidad de poses distintas. Para entrenar un detector robusto las imágenes de entrenamiento deberían tener objetos aleatorios en la imagen junto con los objetos deseados, y deberían tener una variedad de fondos y condiciones de iluminación. También es conveniente emplear imágenes donde el objeto deseado esté parcialmente oscurecido, superpuesto con otra cosa o que se salga por un lado de la imagen. Por suerte para nosotros, los semáforos que coloquemos en nuestra simulación van a ser siempre idénticos, misma forma, mismo entorno y misma iluminación, lo único que va a variar será su posición. Por esta razón se reduce el número de imágenes necesarias y se facilita el aprendizaje del modelo, aunque será necesario un reentrenamiento a la hora de emplear este modelo con el Robobo real.

Para obtener el conjunto de imágenes que se puede observar en la Figura 5-7 se ha creado un script, que se puede encontrar en el Anexo 4 de la memoria, el cual guarda una captura de la imagen de la cámara de la simulación cada tres segundos, tiempo suficiente para que el usuario varíe la posición y/o ángulo del objeto a capturar.

Fuera de la simulación, recopilar imágenes a mano es un trabajo arduo y molesto, habría que hacer trabajo de campo con una cámara o buscar el objeto deseado en internet. Para esto último existen herramientas que nos facilitan el trabajo permitiéndonos descargar de golpe todas las imágenes encontradas por el motor de búsqueda del explorador web [38]. Debe tenerse en cuenta que, a la hora de descargar imágenes o tomarlas con una cámara de alta resolución, estas no sean muy grandes o pesadas, porque ralentizará el proceso de aprendizaje del detector. Este problema nos lo hemos ahorrado al guardar imágenes directamente desde la simulación, ya que entrenaremos con un tamaño de imagen de 512x512 píxeles y menos de 150 KB. Otra forma de aumentar la cantidad de imágenes para el entrenamiento es el uso de herramientas de aumento de datos [39], las cuales modifican un conjunto de imágenes existente mediante transformaciones geométricas, cambios de tono o saturación, etc. obteniendo un nuevo conjunto de imágenes ligeramente alteradas mucho más grande que el original.

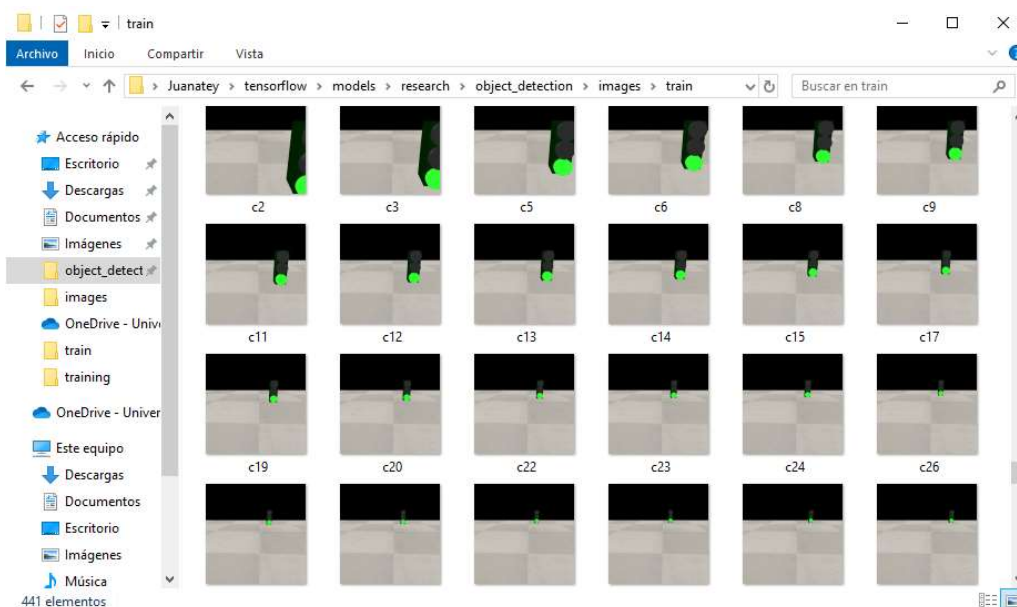


Figura 5-7 Conjunto de datos para el entrenamiento del modelo

5.2.2.2 Etiquetado

Una vez recopiladas todas las imágenes que se utilizarán para mejorar el modelo, debemos etiquetarlas, para lo que usaremos la herramienta LabelImg [40] cuya interfaz gráfica se puede ver en la Figura 5-8. LabelImg es muy liviano y fácil de usar, para ello abriremos el directorio donde hemos guardado nuestras imágenes y con ayuda del ratón iremos individualmente marcando con un recuadro las zonas de la imagen que contengan un objeto de interés, y etiquetándolo. Este proceso generará en nuestra carpeta un documento XML para cada imagen.

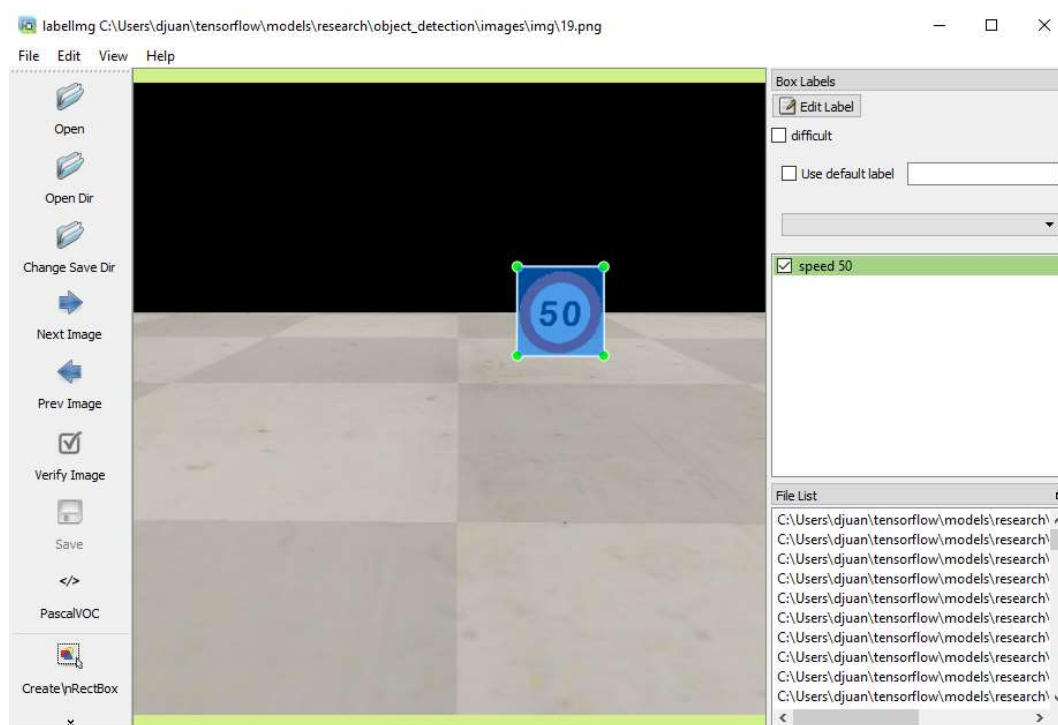
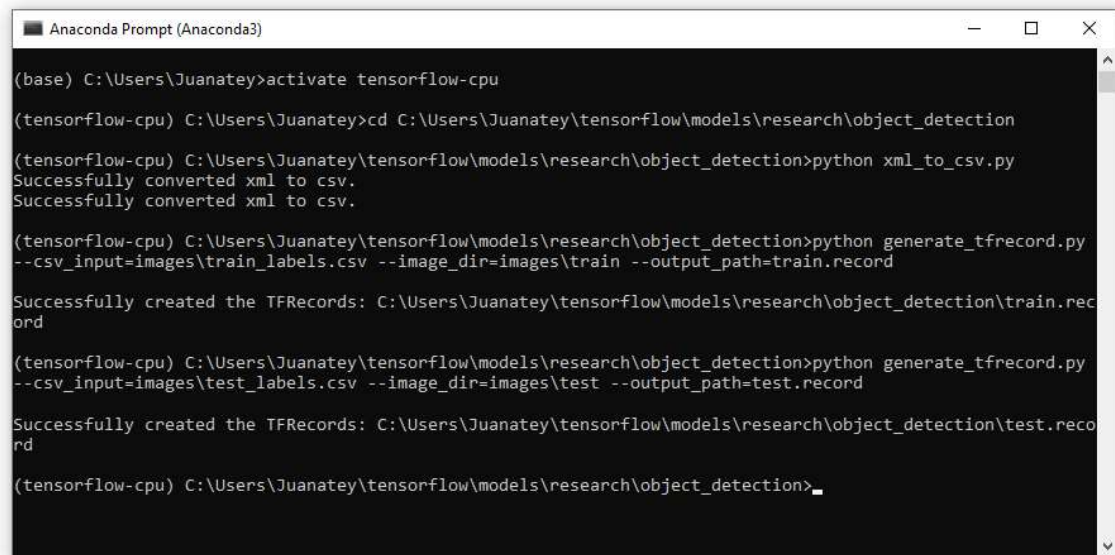


Figura 5-8 Ventana de aplicación de LabelImg

Una vez concluido el etiquetado de las imágenes dividiremos el conjunto en dos carpetas: una primera carpeta de entrenamiento, que corresponderá a la mayor parte del conjunto (entre un 70% y un 90% del total); y el resto en una carpeta para fines de evaluación.

Con las imágenes separadas en la proporción que se haya escogido, es el momento de generar los archivos que sirven como datos de entrada para el entrenamiento del modelo. Debemos abrir con un editor de texto los scripts *xml_to_csv.py* y *generate_tfrecord.py* que ya vienen incluidos en la carpeta de la API de detección de objetos cuando la descargamos. Nos aseguraremos de que la ruta del directorio de imágenes coincida con el nuestro y actualizaremos en *generate_tfrecord.py* a partir de la línea 31 la lista de objetos, añadiendo los que acabamos de etiquetar.

Tras este paso, ejecutaremos ambos scripts con ayuda de la consola. El resultado si todo ha salido bien será como el de la Figura 5-9 y se habrán creado en nuestra carpeta dos archivos *train.record* y *test.record*.



```
(base) C:\Users\Juanatey>activate tensorflow-cpu

(tensorflow-cpu) C:\Users\Juanatey>cd C:\Users\Juanatey\tensorflow\models\research\object_detection

(tensorflow-cpu) C:\Users\Juanatey\tensorflow\models\research\object_detection>python xml_to_csv.py
Successfully converted xml to csv.
Successfully converted xml to csv.

(tensorflow-cpu) C:\Users\Juanatey\tensorflow\models\research\object_detection>python generate_tfrecord.py
--csv_input=images\train_labels.csv --image_dir=images\train --output_path=train.record

Successfully created the TFRecords: C:\Users\Juanatey\tensorflow\models\research\object_detection\ttrain.record

(tensorflow-cpu) C:\Users\Juanatey\tensorflow\models\research\object_detection>python generate_tfrecord.py
--csv_input=images\test_labels.csv --image_dir=images\test --output_path=test.record


Successfully created the TFRecords: C:\Users\Juanatey\tensorflow\models\research\object_detection\ttest.record

(tensorflow-cpu) C:\Users\Juanatey\tensorflow\models\research\object_detection>_
```

Figura 5-9 Generación de los datos de entrada

5.2.2.3 Crear mapa de etiquetas

El modelo de TensorFlow trabaja con números enteros, por lo que debemos asignar cada una de las etiquetas anteriores a un valor, de la manera que vemos en la Figura 5-10. Es importante que los números de ID del mapa de etiquetas sean los mismos que los definidos en el archivo *generate_tfrecord.py*. Este mapa de etiquetas es utilizado tanto por los procesos de entrenamiento como de detección, por lo que cuando estemos usando el modelo debemos asegurarnos de usar el mismo mapa de etiquetas con el que lo hemos entrenado previamente.



```
robobo_label_map: Bloc de notas
Archivo Edición Formato Ver Ayuda
{
  "item": {
    "id": 1,
    "name": "speed 50"
  },
  "item": {
    "id": 2,
    "name": "speed 120"
  },
  "item": {
    "id": 3,
    "name": "stop"
  }
}
```

Figura 5-10 Mapa de etiquetas para detección

5.2.2.4 Comienzo del entrenamiento

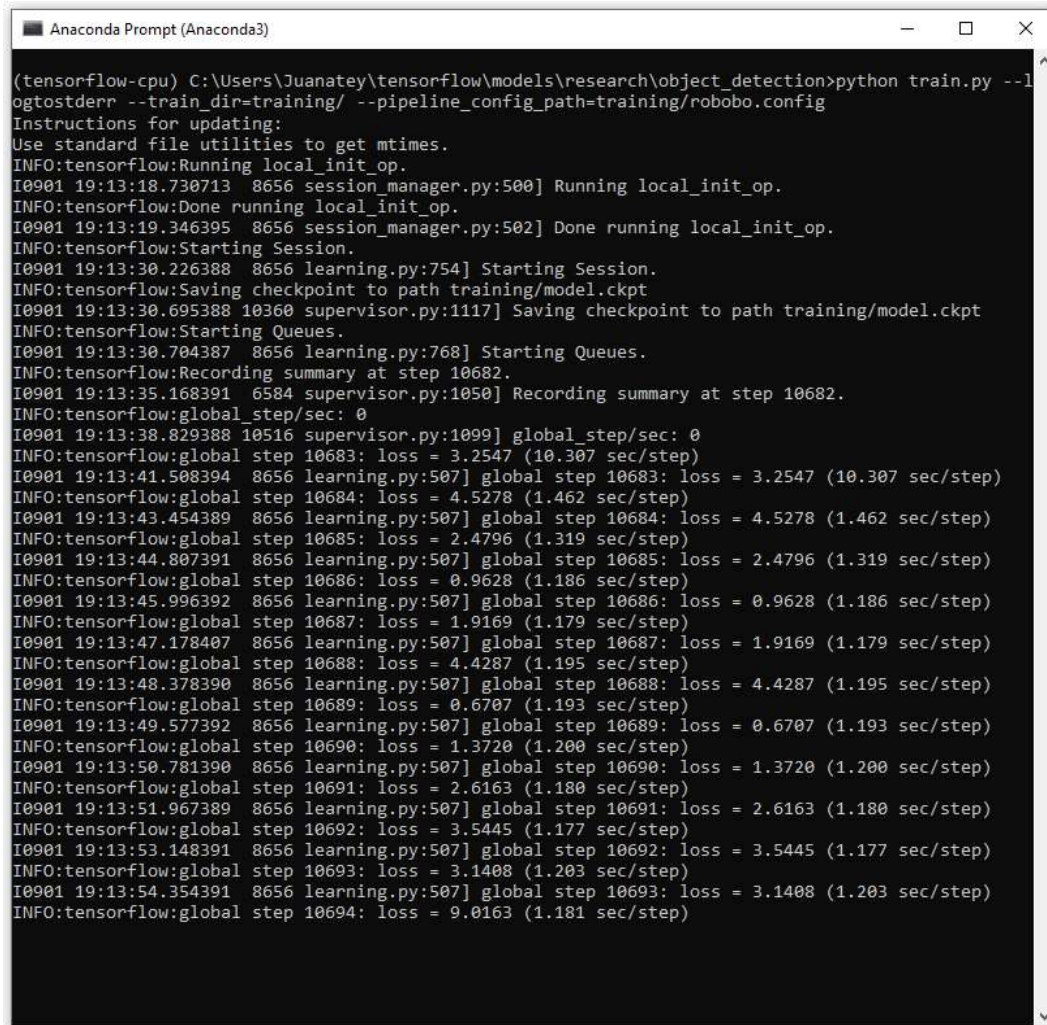
Como último paso antes de poder comenzar el entrenamiento debemos crear un archivo donde se defina qué modelo vamos a entrenar y los parámetros que se utilizarán para el entrenamiento. Si hemos descargado uno de los modelos del repositorio oficial ya tendremos en la carpeta descargada un archivo de configuración que podemos editar. Lo abriremos con un editor de texto y cambiaremos los siguientes parámetros acorde a nuestras necesidades:

- Número clases, es decir, el número de objetos diferentes que se desea identificar. En nuestro caso serán 33.
- Las rutas de acceso al directorio con los datos de entrenamiento.
- Tamaño del lote, que será mayor o menor en función de la memoria RAM que tengamos disponible. En nuestro caso 8 GB.
- Número de pasos o iteraciones. Si no se establece serán 20000 por defecto.
- El punto desde el que comenzará el entrenamiento.

Este último punto se debe a que no tenemos por qué entrenar desde cero el modelo de cada vez. La rutina de entrenamiento guarda periódicamente puntos de control aproximadamente cada cinco minutos. Es posible terminar el entrenamiento en cualquier momento y luego retomarlo desde el punto de control. Esto es muy útil porque si tenemos un modelo entrenado es posible añadir a posteriori nuevas clases de objetos (siguiendo todos los pasos anteriores). En el Anexo 5 encontramos el código correspondiente al archivo de configuración empleado en este proyecto.

Ahora sí, se puede comenzar el entrenamiento desde la consola. Si todo se ha configurado correctamente, el resultado debe ser parecido al que vemos en la Figura 5-11. El proceso de aprendizaje se lleva a cabo mediante el método de propagación hacia atrás o retropropagación. La propagación hacia atrás detecta las ponderaciones correctas que se deben aplicar a los nodos de una red neuronal mediante la comparación de las salidas actuales de la red con los resultados correctos o deseados [41]. Cada paso o iteración del entrenamiento nos proporciona la siguiente información:

- La función de costo o pérdida: es la diferencia entre el resultado deseado y el resultado en esa iteración, medida como la suma de los cuadrados de las diferencias.
- *Global step*: es el número total de pasos o iteraciones, que a su vez es el número de lotes empleados. Cada vez que se proporciona un lote, las ponderaciones se actualizan en la dirección que minimiza la pérdida.
- El tiempo transcurrido en dicha iteración.



```
(tensorflow-cpu) C:\Users\Juanatey\tensorflow\models\research\object_detection>python train.py --logtostderr --train_dir=training/ --pipeline_config_path=training/robobo.config
Instructions for updating:
Use standard file utilities to get mtimes.
INFO:tensorflow:Running local_init_op.
I0901 19:13:18.730713 8656 session_manager.py:500] Running local_init_op.
INFO:tensorflow:Done running local_init_op.
I0901 19:13:19.346395 8656 session_manager.py:502] Done running local_init_op.
INFO:tensorflow:Starting Session.
I0901 19:13:30.226388 8656 learning.py:754] Starting Session.
INFO:tensorflow:Saving checkpoint to path training/model.ckpt
I0901 19:13:30.695388 10360 supervisor.py:1117] Saving checkpoint to path training/model.ckpt
INFO:tensorflow:Starting Queues.
I0901 19:13:30.704387 8656 learning.py:768] Starting Queues.
INFO:tensorflow:Recording summary at step 10682.
I0901 19:13:35.168391 6584 supervisor.py:1050] Recording summary at step 10682.
INFO:tensorflow:global_step/sec: 0
I0901 19:13:38.829388 10516 supervisor.py:1099] global_step/sec: 0
INFO:tensorflow:global step 10683: loss = 3.2547 (10.307 sec/step)
I0901 19:13:41.508394 8656 learning.py:507] global step 10683: loss = 3.2547 (10.307 sec/step)
INFO:tensorflow:global step 10684: loss = 4.5278 (1.462 sec/step)
I0901 19:13:43.454389 8656 learning.py:507] global step 10684: loss = 4.5278 (1.462 sec/step)
INFO:tensorflow:global step 10685: loss = 2.4796 (1.319 sec/step)
I0901 19:13:44.807391 8656 learning.py:507] global step 10685: loss = 2.4796 (1.319 sec/step)
INFO:tensorflow:global step 10686: loss = 0.9628 (1.186 sec/step)
I0901 19:13:45.996392 8656 learning.py:507] global step 10686: loss = 0.9628 (1.186 sec/step)
INFO:tensorflow:global step 10687: loss = 1.9169 (1.179 sec/step)
I0901 19:13:47.178407 8656 learning.py:507] global step 10687: loss = 1.9169 (1.179 sec/step)
INFO:tensorflow:global step 10688: loss = 4.4287 (1.195 sec/step)
I0901 19:13:48.378390 8656 learning.py:507] global step 10688: loss = 4.4287 (1.195 sec/step)
INFO:tensorflow:global step 10689: loss = 0.6707 (1.193 sec/step)
I0901 19:13:49.577392 8656 learning.py:507] global step 10689: loss = 0.6707 (1.193 sec/step)
INFO:tensorflow:global step 10690: loss = 1.3720 (1.200 sec/step)
I0901 19:13:50.781390 8656 learning.py:507] global step 10690: loss = 1.3720 (1.200 sec/step)
INFO:tensorflow:global step 10691: loss = 2.6163 (1.180 sec/step)
I0901 19:13:51.967389 8656 learning.py:507] global step 10691: loss = 2.6163 (1.180 sec/step)
INFO:tensorflow:global step 10692: loss = 3.5445 (1.177 sec/step)
I0901 19:13:53.148391 8656 learning.py:507] global step 10692: loss = 3.5445 (1.177 sec/step)
INFO:tensorflow:global step 10693: loss = 3.1408 (1.203 sec/step)
I0901 19:13:54.354391 8656 learning.py:507] global step 10693: loss = 3.1408 (1.203 sec/step)
INFO:tensorflow:global step 10694: loss = 9.0163 (1.181 sec/step)
```

Figura 5-11 Entrenamiento del modelo

5.2.2.5 Monitorización del proceso de entrenamiento

Es posible supervisar el entrenamiento del modelo de forma más visual gracias a la herramienta TensorBoard. Para usarla es necesario abrir una nueva consola de Python, acceder a la carpeta desde la que iniciamos el entrenamiento en el paso anterior y ejecutar TensorBoard. Esto nos permite abrir a través del navegador web una página con información y multitud de gráficos sobre los distintos parámetros del entrenamiento. Un gráfico importante es el que muestra la pérdida a lo largo del tiempo, en la

Figura 5-12 encontramos el gráfico correspondiente al entrenamiento efectuado durante este proyecto en particular.

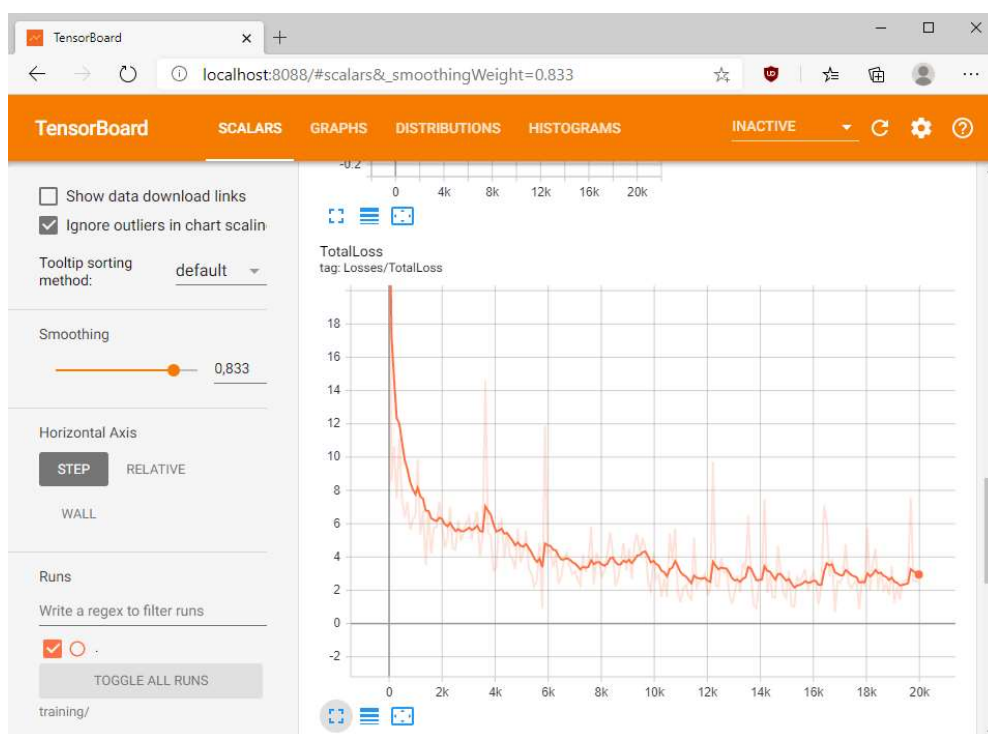


Figura 5-12 Gráfico de la pérdida en función del tiempo

Inicialmente, las ponderaciones y sesgos de los nodos (neuronas) de la red se suelen establecer en valores aleatorios que a menudo producen un valor alto de la pérdida en los primeros pasos de entrenamiento. El objetivo es que el valor de la pérdida sea lo más pequeño posible. Sin embargo, debe tenerse cuidado de no sobreajustar el modelo, lo que significa que el modelo funcionará mal cuando se aplique a imágenes que no pertenezcan al conjunto de datos.

Más concretamente para este proyecto, se puede ver en la gráfica que la pérdida decrece rápidamente hasta un valor de 6, sigue disminuyendo poco a poco hasta aproximadamente 2 y luego se mantiene más o menos estable en ese valor, por lo que puede ser considerado como el mínimo. Se ha terminado el entrenamiento a los 20000 pasos, pero vemos en el gráfico que habría sido suficiente en torno a los 16000. Por suerte para nosotros, como se ha comentado el entrenamiento guarda periódicamente puntos de control, por lo que descartamos los últimos 4000 pasos en los que el modelo se ha sobreajustado y empeorado sus estadísticas.

5.2.3 Ejecución en tiempo real

Se ha creado un script, cuyo código se encuentra en el Anexo 6 de la memoria, que permite emplear el modelo entrenado en el paso anterior analizando en tiempo real la imagen de la cámara de nuestro robot simulado. Este script funciona con otros modelos sin necesidad de hacer cambios mayores en el código, para ello basta con cambiar la ruta de la carpeta.

Su funcionamiento consiste en sobrescribir la imagen de vídeo, encuadrando los objetos detectados junto con una puntuación, la cual indica el grado de confianza en que el objeto se detectó correctamente. La puntuación es un número entre 0 y 1 (es decir, entre 0% y 100%). Es necesario definir previamente un umbral de corte por debajo del cual el modelo descartará los resultados obtenidos. Por defecto, este umbral está

colocado en 0,7 (lo que significa una probabilidad del 70% de que la detección sea correcta). Utilizar un umbral bajo aumentará la aparición de falsos positivos, es decir áreas de la imagen que se identifican como un objeto cuando en realidad no lo son u objetos nombrados incorrectamente. Por el contrario, un umbral demasiado alto es proclive a los falsos negativos, objetos que se encuentran en escena pero que no se identifican por no haber una confianza alta en su detección.

La información proporcionada por la detección de objetos (como el número de objetos en la escena, la clase de cada objeto, la confianza en la detección y las dimensiones y posición de los recuadros) será fundamental para el posterior desarrollo de algoritmos de toma de decisiones.

6 PROTOCOLO DE PRUEBAS Y VALIDACIÓN

En este capítulo veremos algunas de las pruebas realizadas para validar el trabajo que hemos descrito a lo largo del presente documento. Estas pruebas están enfocadas a detectar los límites de funcionamiento de los algoritmos desarrollados.

6.1 Validación de la detección de carriles

Como se ha visto en el capítulo anterior y se puede ver en el Anexo 3 con el código correspondiente, el algoritmo de detección de carriles devuelve las coordenadas de dos puntos pertenecientes a cada una de las líneas detectadas en la imagen. Por lo tanto, nos devolverá dos vectores de coordenadas con la forma: $[x1 \ y1 \ x2 \ y2]$. Estos vectores de coordenadas los usaremos posteriormente para tomar decisiones sobre la dirección del vehículo y representar las líneas dibujándolas sobre la imagen. Esto último ayuda al usuario a comprobar visualmente si el programa está detectando correctamente el carril (o si no lo detecta), acelerando así el proceso de supervisión y corrección de errores.

El procedimiento que seguirá para validar el algoritmo de detección de carriles consistirá en analizar los valores de las coordenadas devueltos por el mismo en diferentes situaciones y comprobar que se ajusten correctamente a las líneas de la imagen. El desempeño de los distintos algoritmos varía considerablemente en función del hardware disponible y los parámetros del Robobo. Estas pruebas se han realizado en un ordenador con un procesador Intel Core i5-2500 con 8 GB de memoria RAM y una tarjeta gráfica NVIDIA GeForce GTX 1050 Ti. Las velocidades del Robobo serán las indicadas en cada prueba o en la tabla correspondiente, y se ha establecido una inclinación del TILT de 105.

6.1.1 Análisis de los valores de las coordenadas en función de la velocidad

Para la ejecución de esta prueba se hace circular al Robobo a través de un carril recto a distintas velocidades, siguiendo la disposición de la Figura 6-1, y se observan los valores que devuelve el algoritmo a lo largo de todo el recorrido. Posteriormente se realizará una interpretación de los datos, prestando especial atención a la dispersión de los valores. También comprobaremos cuales son los resultados de la detección con el robot parado. Para ello lo colocaremos en mitad de la recta y tomaremos 10 mediciones para hacer la media.

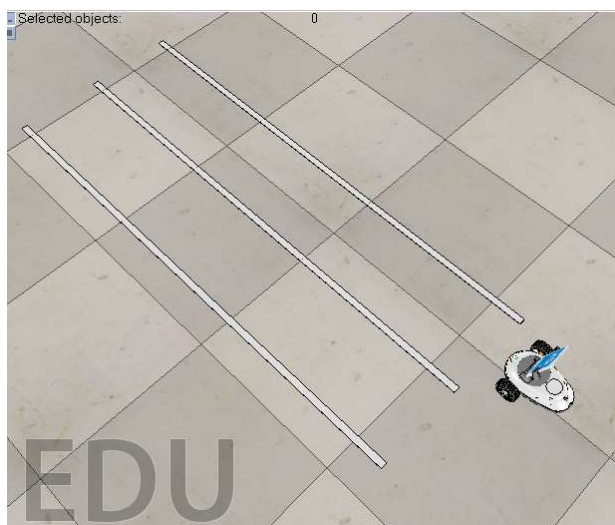


Figura 6-1 Disposición de los elementos para la primera prueba

Durante la prueba hay un valor de cada coordenada que se repite con mucha más frecuencia que los demás (la moda), por lo que tomaremos este valor como el verdadero. No obstante, estos valores no se mantienen estáticos en el tiempo y se sufre una dispersión en las medidas, por lo que anotaremos para cada caso el valor más alejado respecto al que hemos considerado como verdadero, es decir, su desviación absoluta máxima. Por lo tanto, los datos ya procesados se muestran en la siguiente tabla:

Velocidad	Línea	Valor moda				Desviación absoluta máx.			
		x1	y1	x2	y2	x1	y1	x2	y2
0	izq	20	512	113	362	3	0	3	0
	der	455	512	376	362	6	0	4	0
20	izq	26	512	115	362	1	0	1	0
	der	453	512	374	362	15	0	9	0
50	izq	61	512	143	362	6	0	6	0
	der	488	512	400	362	4	0	4	0
90	izq	71	512	150	362	7	0	5	0
	der	499	512	409	362	7	0	6	0

Tabla 6-1 Valores de las coordenadas en función de la velocidad

Se puede comprobar en la tabla que en este experimento las líneas de los carriles son detectadas en todo momento, a cualquier velocidad, y que los valores de las coordenadas en el eje y son siempre iguales. Se han obtenido también otras conclusiones, pero las comentaremos más adelante después de haberlas contrastado con la siguiente prueba.

6.1.2 Análisis de los valores de las coordenadas en función de la curvatura

En esta prueba se deja el robot parado ante una serie de curvas como la que se puede ver en la Figura 6-2. Analizaremos por tanto las diferencias en los valores de las coordenadas que devuelve el algoritmo para cada caso.

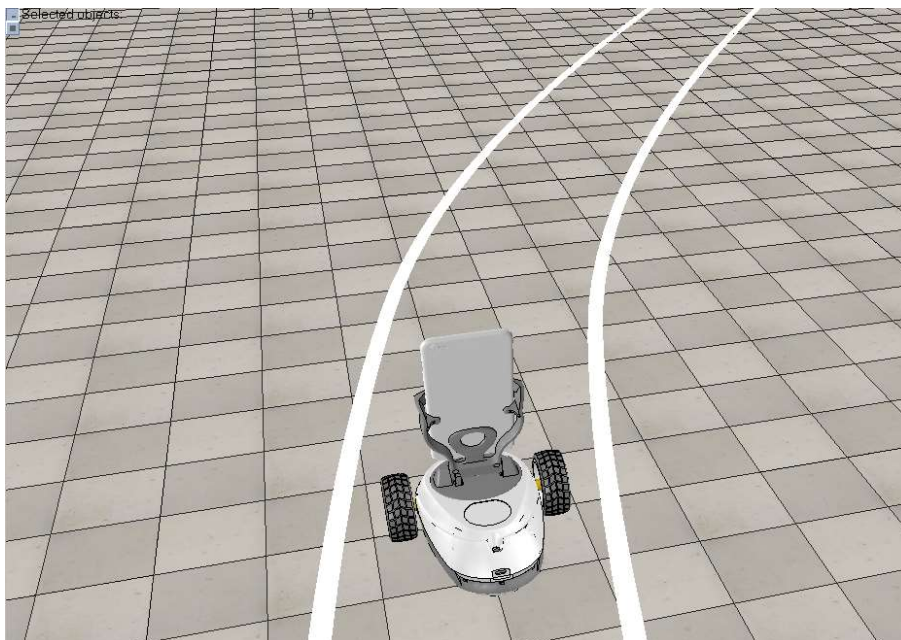


Figura 6-2 Disposición de los elementos para la segunda prueba

Los valores se han tomado de forma análoga a la prueba anterior con el robot parado, tomando 10 muestras y calculando el valor moda y la desviación absoluta máxima. Los resultados de la segunda prueba se muestran en la Tabla 6-2 a continuación:

Tipo de tramo	Línea	Valor moda				Desviación absoluta máx.			
		x1	y1	x2	y2	x1	y1	x2	y2
curva suave derecha	izq	6	512	116	362	6	0	7	0
	der	462	512	390	362	1	0	1	0
curva pronunciada derecha	izq	58	512	165	362	6	0	3	0
	der	0	0	0	0	0	0	0	0
curva suave izquierda	izq	38	512	122	362	9	0	4	0
	der	643	512	464	362	6	0	4	0
curva pronunciada izquierda	izq	0	0	0	0	0	0	0	0
	der	473	512	357	362	9	0	3	0

Tabla 6-2 Valores de las coordenadas en función de la curvatura

Como se puede observar, cuando nos encontramos una curva pronunciada hacia cualquiera de los dos lados, la línea del carril correspondiente a ese lado deja de detectarse. Esto se debe a que dicha línea deja de estar dentro del área de detección, como se puede comprobar en la Figura 6-3. Cuando esto ocurre, el algoritmo devuelve cero en todas las coordenadas.

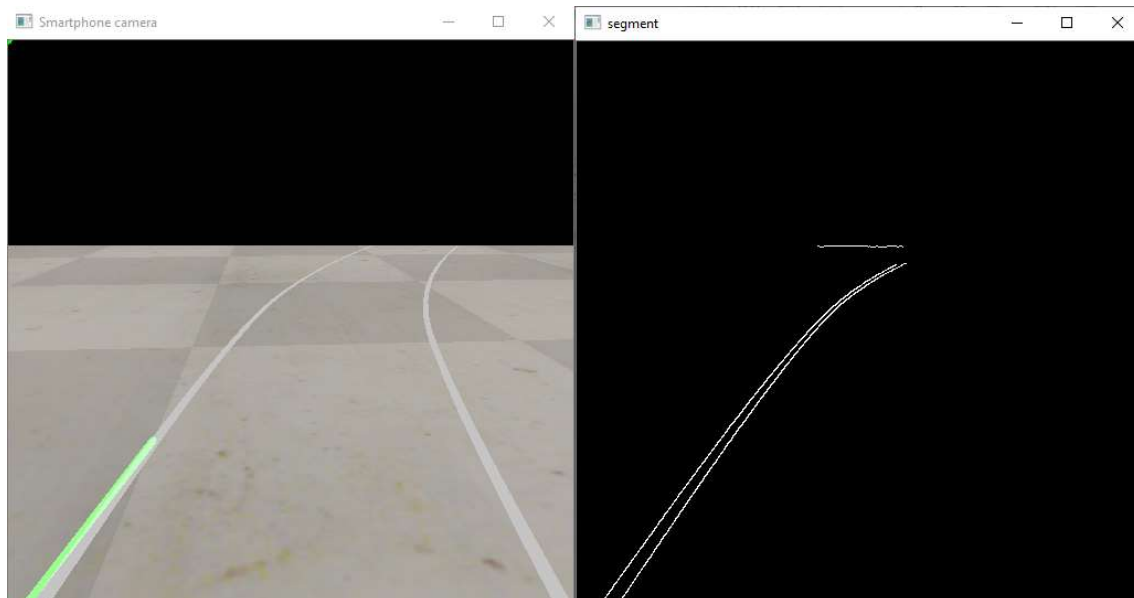


Figura 6-3 Línea fuera del área de detección

6.1.3 Conclusiones

Una vez analizado el comportamiento del algoritmo de detección, se pueden inferir las siguientes conclusiones:

- La detección ocurre sin problemas en todo momento, a cualquier velocidad y tanto en líneas rectas como ligeramente curvadas o incluso discontinuas, siempre que estas se encuentren dentro del área de detección.
- La ausencia de una línea nos permite saber en qué dirección debe girar el Robobo, como se ha explicado. Desafortunadamente, no se ha encontrado una correlación suficiente entre las coordenadas devueltas y la dirección del trazado como para elaborar un control de la dirección más sofisticado. La continua desviación en los valores, incluso cuando el robot se encuentra parado, tampoco ayuda en esta tarea.

6.2 Validación de la detección de objetos

En este apartado vamos a probar el modelo de red neuronal entrenado en el capítulo anterior con el objetivo de determinar si el entrenamiento ha sido satisfactorio para las clases de objetos que se han definido.

6.2.1 Análisis de la puntuación con la distancia

Para esta prueba se ha dejado al Robobo estático, colocando a su frente los objetos a detectar y alejándolos paulatinamente hasta una distancia de dos metros de la cámara (aproximadamente 195 centímetros de la parte frontal del Robobo). Se anota para cada distancia la clase de objeto que el modelo cree detectar y una puntuación que indica la confianza de que la detección sea correcta. El hardware es el mismo que en las pruebas anteriores, la inclinación del TILT se mantiene en 105 y el umbral mínimo de detección se ha establecido en un 30% de confianza.

Lo primero que analizaremos es la capacidad de detectar otros Robobos que se encuentren circulando por nuestro entorno. Analizaremos distintos ángulos de detección de Robobos, porque no solo es importante detectarlos de frente, sino que también nos interesa que la detección sea fiable desde un costado, que será como lo vemos en los cruces, o desde la parte trasera, para evitar colisiones por alcance. Además, analizaremos los semáforos tanto en rojo como en amarillo o verde, y una muestra de señales que se pueden encontrar frecuentemente en la carretera.

Se han realizado pruebas complementarias como identificación de varios objetos en la escena y detección con el Robobo en movimiento, siendo los resultados similares a los obtenidos en la prueba principal. A continuación, se muestran las tablas con los resultados de la prueba para todos los objetos mencionados:

Robobo						
Distancia	Vista frontal		Vista lateral		Vista posterior	
cm	Clase	Confianza	Clase	Confianza	Clase	Confianza
15	robobo	85%	robobo	47%	robobo	86%
25	robobo	90%	robobo	88%	robobo	89%
35	robobo	90%	robobo	86%	robobo	89%
45	robobo	87%	robobo	83%	robobo	84%
55	robobo	86%	robobo	81%	robobo	82%
65	robobo	85%	robobo	64%	robobo	40%
75	robobo	72%	robobo	63%	robobo	37%
85	robobo	67%	robobo	65%	robobo	45%
95	robobo	69%	robobo	58%	robobo	39%
105	robobo	74%	robobo	49%	robobo	69%
115	robobo	74%	robobo	40%	robobo	63%
125	robobo	65%	robobo	40%	robobo	62%
135	robobo	58%	robobo	35%	robobo	45%
145	robobo	54%	robobo	30%	robobo	44%
155	robobo	38%	robobo	32%	robobo	38%
165	robobo	32%	robobo	32%	robobo	31%
175	robobo	30%	bicicleta	36%	robobo	30%
185	-	-	bicicleta	40%	robobo	30%
195	-	-	bicicleta	50%	botella	31%

Tabla 6-3 Valores de la confianza del Robobo en función de la distancia

Semáforo						
Distancia	Rojo		Amarillo		Verde	
cm	Clase	Confianza	Clase	Confianza	Clase	Confianza
15	sem. rojo	99%	sem. amarill	99%	sem. verde	99%
25	sem. rojo	98%	sem. amarill	98%	sem. verde	98%
35	sem. rojo	99%	sem. amarill	99%	sem. verde	98%
45	sem. rojo	99%	sem. amarill	99%	sem. verde	99%
55	sem. rojo	77%	sem. amarill	54%	sem. verde	84%
65	sem. rojo	31%	sem. amarill	30%	sem. verde	76%
75	-	-	sem. amarill	78%	sem. verde	87%
85	-	-	sem. amarill	77%	sem. verde	95%
95	STOP	40%	sem. amarill	45%	sem. verde	92%
105	STOP	51%	-	-	sem. verde	55%
115	STOP	69%	sem. amarill	30%	sem. verde	75%
125	STOP	81%	sem. amarill	30%	sem. verde	85%
135	STOP	75%	-	-	sem. verde	88%
145	STOP	58%	-	-	sem. verde	86%
155	-	-	-	-	sem. verde	72%
165	-	-	-	-	sem. verde	50%
175	velocidad 50	57%	velocidad 50	31%	-	-
185	velocidad 50	34%	-	-	-	-
195	velocidad 50	32%	-	-	-	-

Tabla 6-4 Valores de la confianza de los semáforos en función de la distancia

Señales						
Distancia	Velocidad 50		Velocidad 120		STOP	
cm	Clase	Confianza	Clase	Confianza	Clase	Confianza
15	velocidad 50	89%	velocidad 120	87%	STOP	99%
25	velocidad 50	88%	velocidad 120	36%	STOP	99%
35	velocidad 50	85%	velocidad 50	92%	STOP	99%
45	velocidad 50	82%	velocidad 50	89%	STOP	94%
55	velocidad 50	31%	velocidad 50	81%	STOP	91%
65	velocidad 50	50%	velocidad 50	31%	STOP	85%
75	velocidad 50	42%	velocidad 50	62%	STOP	99%
85	velocidad 50	46%	velocidad 120	31%	STOP	97%
95	-	-	velocidad 50	52%	STOP	40%
105	velocidad 50	30%	-	-	STOP	71%
115	velocidad 50	67%	-	-	STOP	32%
125	velocidad 50	74%	velocidad 50	34%	velocidad 50	51%
135	velocidad 50	76%	velocidad 50	32%	velocidad 50	34%
145	velocidad 50	68%	velocidad 50	31%	velocidad 50	40%
155	velocidad 50	63%	velocidad 50	33%	velocidad 50	51%
165	velocidad 50	61%	velocidad 50	67%	velocidad 50	84%
175	velocidad 50	58%	velocidad 50	60%	velocidad 50	81%
185	velocidad 50	55%	velocidad 50	52%	velocidad 50	76%
195	velocidad 50	55%	velocidad 50	50%	velocidad 50	74%

Tabla 6-5 Valores de la confianza de las señales en función de la distancia

6.2.2 Conclusiones

Una vez obtenidos y ordenados conforme a las tablas anteriores los datos de la prueba de detección de objetos, se pueden inferir de dichos datos las siguientes conclusiones:

- En mediciones que se encuentran por debajo de los 50 cm la precisión es muy satisfactoria, superando el 80% de confianza e incluso rozando el 100% en algunos casos.
- Tanto la precisión como la distancia máxima de detección depende en gran medida del tipo de objeto a detectar, destacando positivamente la identificación del Robobo desde todas sus perspectivas, la identificación del semáforo cuando este está en verde y la señal de límite de velocidad 50. La nota negativa la ponen la señal de límite de velocidad 120 y el semáforo en rojo.
- La baja efectividad en la detección de la señal de límite de velocidad 120 se explica debido al gran parecido que guarda con la señal de límite de velocidad 50, confundiéndola con esta última. Esto puede solucionarse aumentando el entrenamiento del modelo con imágenes en las que ambas señales estén presentes simultáneamente.
- Los problemas a la hora de identificar el semáforo en rojo podrían explicarse debido a que, a una cierta distancia, lo más llamativo en la escena es el círculo rojo de la luz del semáforo, confundiéndose este con un STOP.
- La oclusión parcial de objetos afecta a su detección. Con varios objetos en pantalla, si estos se superponen, aunque sea parcialmente, disminuye la confianza o directamente dejan de detectarse.

- Por el contrario, no existe una diferencia apreciable en la confianza de detección entre que un objeto esté sólo o haya varios objetos en la escena, siempre y cuando no haya oclusiones.

7 RESULTADOS OBTENIDOS

En el presente capítulo se llevará a cabo un análisis de los resultados obtenidos a la hora de poner en práctica los algoritmos de detección. Para ello, se efectuarán distintas demostraciones donde el Robobo deba tomar decisiones de forma autónoma. Esto último implica que en estas demostraciones en ningún momento interviene el usuario, sino que el robot solo dependerá de comportamientos preprogramados y la información que le proporcionan los algoritmos de detección de carriles y objetos. Concretamente, se verán cuatro demostraciones diferentes:

- Corrección de la dirección en una curva para mantenerse dentro de los límites del carril.
- Frenado automático al percibir riesgo de colisión.
- Adecuación de la velocidad ante la presencia de una señal de limitación de la velocidad.
- Atravesar un cruce señalizado por semáforos.

7.1 Desempeño en curvas

Inspirándose en los sistemas reales de asistencia de mantenimiento de carril, se programa un control básico de la dirección del Robobo. Como se ha explicado anteriormente, cuando en una curva la línea del carril se sale del área de detección el algoritmo devuelve un vector de ceros. Cuando esto ocurre, el Robobo girará hacia el mismo lado de la línea que ha perdido, hasta volver a detectarla de nuevo, como se ve en la Figura 7-1.

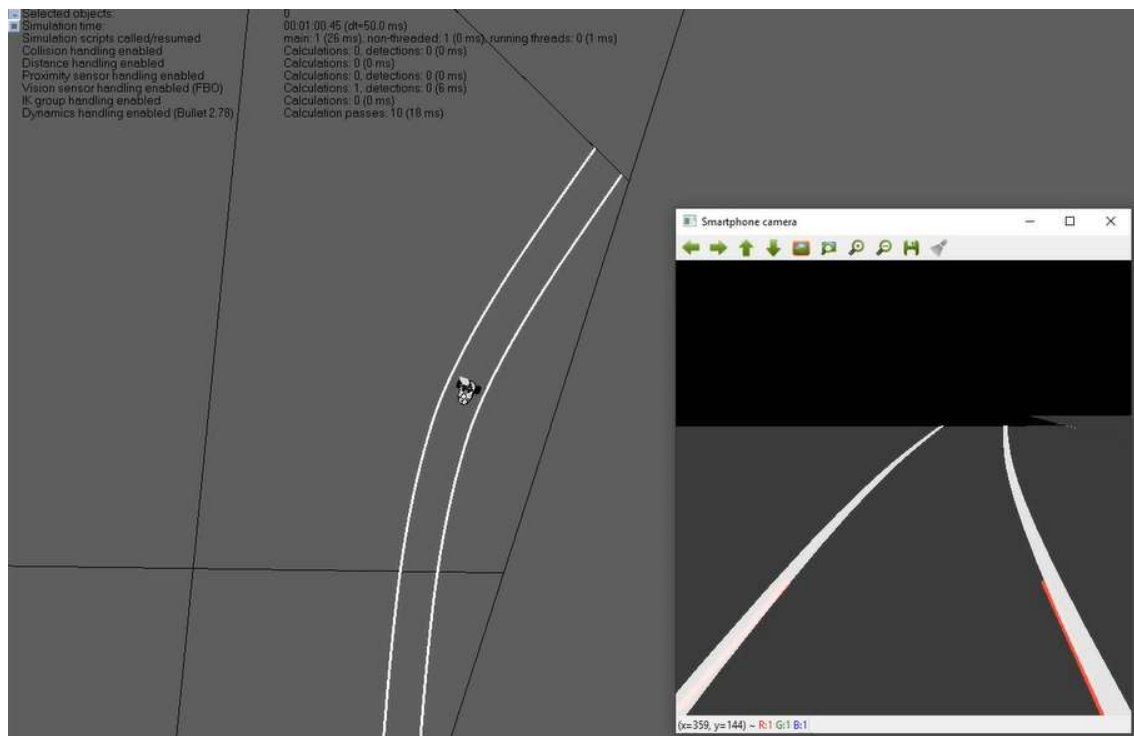


Figura 7-1 Robobo circulando por una curva

7.2 Frenado automático

En esta ocasión se pretende emular los sistemas avanzados de frenado de emergencia (AEBS, por sus siglas en inglés). Para ello se coloca un Robobo incorrectamente detenido en medio de la carretera. Como se puede observar en la Figura 7-2, el Robobo que está circulando detecta al Robobo detenido delante de él y se frena. Valiéndose de información como el tamaño del recuadro y la confianza de detección, si el recuadro se vuelve más pequeño y la confianza descende significa que el obstáculo está alejándose, por lo que el segundo Robobo reanudará la marcha.

Nótese en la figura que los algoritmos de detección de objetos y detección de líneas se están ejecutando a la vez sin problemas.

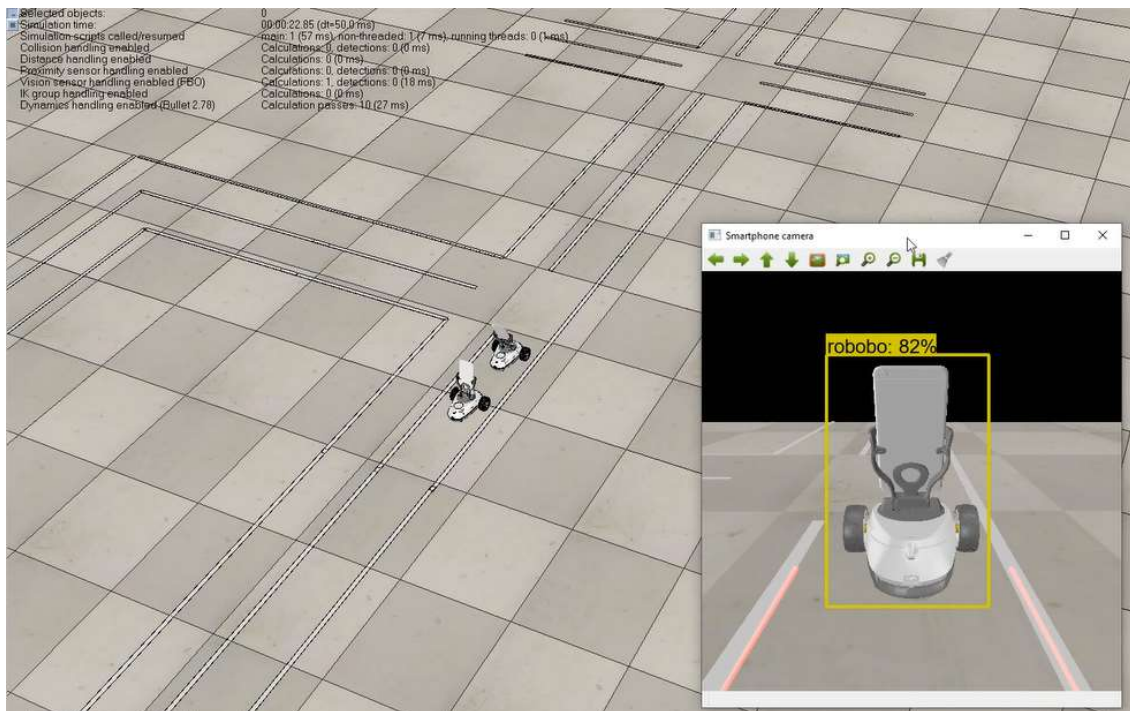


Figura 7-2 Frenado automático del Robobo

7.3 Obediencia de las señales verticales

También basado en tecnología actual, el Robobo puede emular los sistemas de reconocimiento de señales de tráfico (TSR, por sus siglas en inglés) más avanzados y no sólo informar al usuario de la detección de una señal sino actuar en consecuencia con ella. Para ilustrar esto, en la Figura 7-3 se muestra al Robobo circulando y detectando una señal de límite de velocidad. El Robobo guarda en memoria la última clase de objeto detectada. Cuando rebasa la señal la obedecerá en base a un comportamiento preestablecido. Para este caso en particular, se ha multiplicado la velocidad de las ruedas por un coeficiente concreto, cada señal de velocidad tiene asociado un valor para ese coeficiente, aumentando o reduciendo la velocidad del Robobo cuando este las detecta.

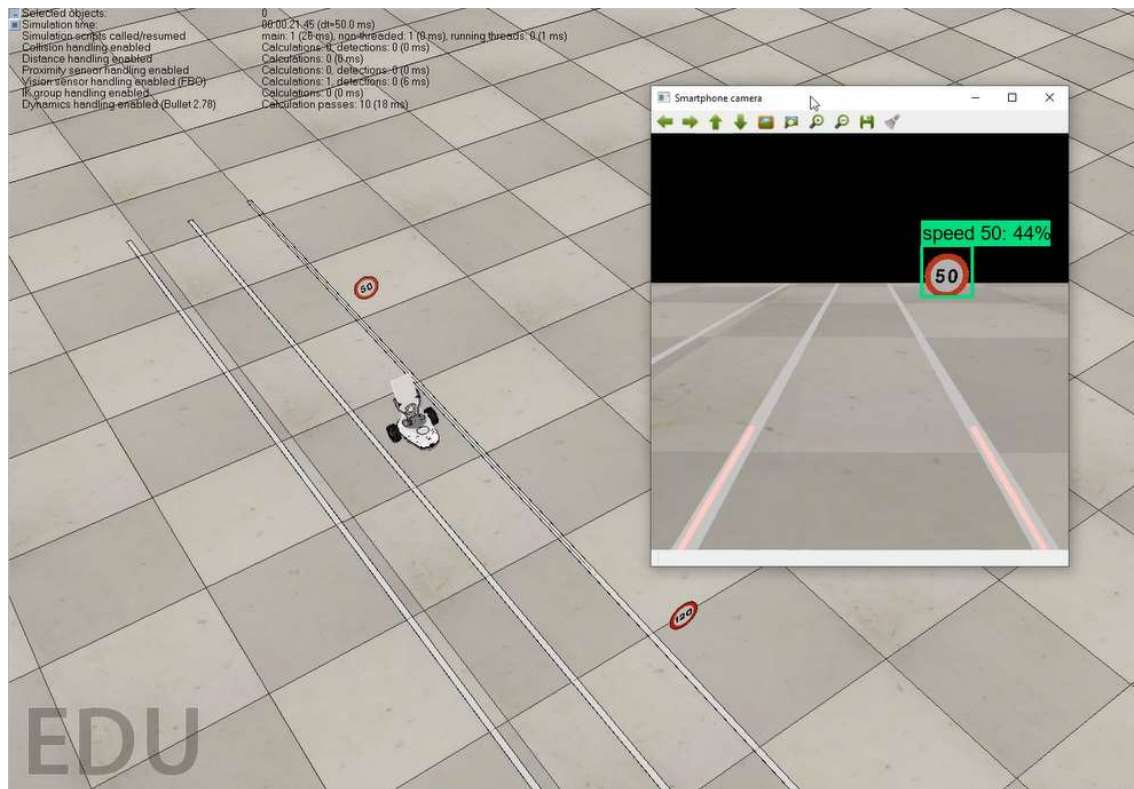


Figura 7-3 Reconocimiento de señales de tráfico

7.4 Obediencia de los semáforos

Es una evolución del sistema del apartado anterior, solo que en este caso se requiere que la acción se ejecute antes de rebasar el semáforo y no después como en caso de las señales. De lo contrario, cuando el Robobo se parase ya no se vería el semáforo en la cámara, por lo que no sabría cuando cambia de color.

Nótese en la Figura 7-4 que el Robobo reconoce también el semáforo situado al otro lado del cruce. Como este semáforo está girado, el modelo cree erróneamente que es un semáforo en verde. Pese a esto, el Robobo no reanuda la marcha, pues en caso de varios objetos en pantalla este sólo obedece al más próximo y con mayor puntuación de confianza.



Figura 7-4 Reconocimiento de semáforos

8 ESTUDIO DE APLICABILIDAD

8.1 Aplicación del modelo

Una vez analizados los resultados obtenidos y comprobado el correcto funcionamiento del modelo de simulación, se pueden destacar los siguientes aspectos acerca de los subobjetivos que se plantearon al principio del proyecto. Así pues:

- Se ha diseñado e implementado un entorno de simulación de las características urbanas básicas existentes en una ciudad encargadas de permitir la circulación de vehículos, incluyendo tanto señalización horizontal como vertical y semáforos.
- Se han implementado diferentes tipos de señales existentes en la realidad para cumplir con la normativa de circulación, incluyendo tanto señales de límite de velocidad como de restricción de paso (STOP). Además, el diseño de estas señales es completamente realista, está totalmente optimizado e integrado en el entorno de simulación y se puede crear de la misma forma cualquier otra señal existente en la realidad.
- Se ha realizado el estudio y análisis de las funcionalidades básicas necesarias para dotar a un vehículo de comportamiento autónomo. Dentro de estas funcionalidades básicas, se ha creído que las más necesarias para un vehículo autónomo son: la detección de las líneas del suelo para mantenerse en todo momento dentro del carril; la identificación de señales y semáforos para cumplir las normas de circulación y la detección de obstáculos en la carretera para poder circular de manera segura.
- Se han implementado las funcionalidades básicas del punto anterior siguiendo además todas las especificaciones de diseño impuestas desde el GII. Estas funcionalidades también se han documentado, optimizado y testeado satisfactoriamente.
- La integración del vehículo autónomo dentro del entorno urbano creado se ha demostrado en el capítulo anterior a este, mostrando que el modelo de simulación del robot es capaz de detectar el entorno mencionado e interactuar convenientemente con él.

Como conclusión, con la realización de todas las tareas anteriores, se considera cumplido el objetivo principal de este Trabajo Fin de Grado de creación de un modelo de simulación de robot móvil para conducción autónoma. Este modelo de simulación puede ser utilizado en sustitución de la plataforma robótica real para labores de docencia e investigación llevadas a cabo en la UDC. Esto significa que, gracias al trabajo realizado en este proyecto, es posible efectuar pruebas de conducción autónoma ahorrando gran cantidad de costes en material, reparaciones y tiempo invertido en ensayos. Mientras que, en docencia, hace posible a los centros educativos proporcionar material didáctico al alumnado sin necesidad de que haya un robot por alumno, ofreciendo a mayores la posibilidad de continuar el curso de la materia de manera no presencial.

8.2 Comparación con sistemas ya existentes

Es complicada la comparación de este modelo de simulación con respecto a otros sistemas ya existentes, pues se trata de un modelo diseñado para unas especificaciones muy concretas. Con respecto a otras simulaciones de vehículos autónomos, puede destacarse que este modelo está basado en un robot real y educativo, por lo que todo el trabajo que se realice puede continuar su desarrollo y estudiar sus aplicaciones en el mundo real, sin incurrir en costes desorbitados. En comparación precisamente con el modelo real del robot, además de las ventajas en cuanto a coste e infraestructura

comentadas anteriormente, puede destacarse la posibilidad de evitar la gran variabilidad del mundo real, eliminando gran parte del ruido de la escena como pueden ser brillos o destellos que afecten a la detección de la cámara.

8.3 Trabajo futuro

A pesar de haber finalizado este Trabajo Fin de Grado cumpliendo con el objetivo principal planteado, el modelo de simulación elaborado aún se encuentra en una fase temprana de su desarrollo. Queda abierta por tanto la posibilidad de cara al futuro de mejorar la toma de decisiones del modelo (mediante el estudio de alternativas, validación en diversas pruebas y comparación de los resultados obtenidos, tal y como se ha hecho en este proyecto con los algoritmos de detección), implementar comportamientos colaborativos o aprovechar las nuevas funciones de extraer la imagen de la cámara para ampliar las capacidades de visión artificial del Robobo. Por ejemplo, mediante el reconocimiento de gestos, que se añadiría al reconocimiento de rostros y de objetos.

9 REFERENCIAS

- [1] International Federation of Robotics, «Executive Summary World Robotics 2019 Service Robots,» [En línea]. Available: Disponible en: https://ifr.org/downloads/press2018/Executive_Summary_WR_Service_Robots_2019.pdf.
- [2] Ministerio de Educación y Formación Profesional, «Isabel Celaá apuesta por la tecnología y la transformación digital para la educación del siglo XXI,» 04 03 2019. [En línea]. Available: Disponible en: <http://www.educacionyfp.gob.es/prensa/actualidad/2019/03/20190304-deusto.html>. [Último acceso: 25 Junio 2020].
- [3] Instituto Nacional de Tecnologías Educativas y de Formación del Profesorado, «Code INTEF,» [En línea]. Available: Disponible en: <http://code.intef.es/situacion-en-espana/>. [Último acceso: 25 Junio 2020].
- [4] Wikipedia, «Educación STEM,» [En línea]. Available: Disponible en: https://es.wikipedia.org/wiki/Educaci%C3%B3n_STEM.
- [5] F. P. A. D. R. J. Bellas, *Robobo: la siguiente generación de robot educativo*, Vols. %1 de %2Cufie. Universidade da Coruña (pág. 13-30), 2018.
- [6] M. Á. C. Q. V. M. Jose María Cañas Plaza, «Uso de simuladores en docencia de robótica móvil,» vol. 4, nº 4, Noviembre 2009.
- [7] S. International, «Levels of driving automation». Patente J3016, 2016.
- [8] J. Delcker, «Politico,» 2018. [En línea]. Available: Disponible en: <https://www.politico.eu/article/delf-driving-car-born-1986-ernst-dickmanns-mercedes/>. [Último acceso: 22 Junio 2020].
- [9] Eureka, «Programme for a european traffic system with highest efficiency and unprecedented safety,» [En línea]. Available: Disponible en: <https://www.eurekanetwork.org/project/id/45>. [Último acceso: 22 Junio 2020].
- [10] Guidehouse Insights, *Guidehouse Insights Leaderboard: Automated Driving Vehicles*, 2019.
- [11] Waymo LLC, «Technology,» [En línea]. Available: Disponible en: <https://waymo.com/tech/>. [Último acceso: 10 Julio 2020].
- [12] Wikipedia, «Wikipedia, the free encyclopedia,» [En línea]. Available: Disponible en: [https://en.wikipedia.org/wiki/Elmer_and_Elsie_\(robots\)](https://en.wikipedia.org/wiki/Elmer_and_Elsie_(robots)). [Último acceso: 15 Julio 2020].
- [13] Wikipedia, «Wikipedia, the free encyclopedia,» [En línea]. Available: Disponible en: https://es.wikipedia.org/wiki/Lego_Mindstorms. [Último acceso: 16 Julio 2020].
- [14] LEGO, «Mindstorms EV3,» [En línea]. Available: Disponible en: <https://www.lego.com/es-es/product/lego-mindstorms-ev3-31313>. [Último acceso: 2020 Julio 28].

- [15] LEGO, «Program in Python with EV3,» [En línea]. Available: Disponible en: <https://education.lego.com/en-us/support/mindstorms-ev3/python-for-ev3>. [Último acceso: 2020 Julio 28].
- [16] A. M. Domínguez, Modelado y simulación de un robot LEGO Mindstorms EV3 mediante V-REP y Matlab, Málaga: Universidad de Málaga, 2016.
- [17] Thymio Wiki, «Thymio II,» [En línea]. Available: Disponible en: <http://wiki.thymio.org/en:thymio>. [Último acceso: 3 Agosto 2020].
- [18] Makeblock, «mBot,» [En línea]. Available: Disponible en: https://www.makeblock.es/productos/robot_educativo_mbot/. [Último acceso: 3 Agosto 2020].
- [19] Robotis, «TurtleBot 3,» [En línea]. Available: Disponible en: <https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>. [Último acceso: 3 Agosto 2020].
- [20] K-Team, «Products,» [En línea]. Available: Disponible en: <https://www.k-team.com/khepera-iv>. [Último acceso: 3 Agosto 2020].
- [21] Kickstarter, «Romo - The smartphone robot for everyone,» [En línea]. Available: Disponible en: <https://www.kickstarter.com/projects/peterseid/romo-the-smartphone-robot-for-everyone/description>. [Último acceso: 3 Agosto 2020].
- [22] GCtronic, «Wheelphone,» [En línea]. Available: Disponible en: <http://www.wheelphone.com/wiki.html>. [Último acceso: 3 Agosto 2020].
- [23] R. Boado de la Fuente, Desarrollo de un modelo en simulación en V-REP de un robot móvil basado en smartphone y soporte al lenguaje Python, Ferrol: RUC (Repositorio institucional da Universidade da Coruña), 2019.
- [24] MINT, «Presentación de la base Robobo,» [En línea]. Available: Disponible en: <http://education.theroboboproject.com/presentacion-de-robobo/presentacion-de-la-base-robobo>. [Último acceso: 20 Abril 2020].
- [25] MINT , «Robobo programming wiki,» [En línea]. Available: Disponible en: <https://github.com/mintforpeople/robobo-programming/wiki>. [Último acceso: 18 Junio 2020].
- [26] Coppelia Robotics, «Recursos,» [En línea]. Available: Disponible en: <https://www.coppeliarobotics.com/resources>. [Último acceso: 21 Abril 2020].
- [27] PUC, «What is Lua?,» [En línea]. Available: Disponible en: <https://www.lua.org/about.html>. [Último acceso: 17 Junio 2020].
- [28] Intel, «OpenCV,» [En línea]. Available: Disponible en: <https://opencv.org/about/>. [Último acceso: 21 Junio 2020].
- [29] Google LLC, «TensorFlow,» [En línea]. Available: Disponible en: <https://www.tensorflow.org/>. [Último acceso: 7 Agosto 2020].
- [30] C.-e. Lin, «Tutorial: Build a lane detector,» 17 Diciembre 2018. [En línea]. Available: Disponible en: <https://towardsdatascience.com/tutorial-build-a-lane-detector-679fd8953132>. [Último acceso: 21 Junio 2020].
- [31] Ministerio de Fomento - Dirección General de Carreteras, *Instrucción de carreteras*.

- [32] J. F. Canny, «A Computational Approach To Edge Detection,» vol. 8(6):679–698, 1986.
- [33] GitHub, Inc., «TF Object Detection API,» [En línea]. Available: Disponible en: https://github.com/tensorflow/models/tree/master/research/object_detection. [Último acceso: 15 Agosto 2020].
- [34] I. & c. Read the Docs, «TensorFlow Object Detection API Tutorial,» [En línea]. Available: Disponible en: <https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/index.html>. [Último acceso: 28 Agosto 2020].
- [35] R. V. S. C. Z. M. K. A. F. A. F. I. W. Z. Huang J, «Speed/accuracy trade-offs for modern convolutional object detectors.,» 2017.
- [36] J. Hui, «mAP (mean Average Precision) for Object Detection,» [En línea]. Available: Disponible en: https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173. [Último acceso: 20 Agosto 2020].
- [37] C. O. i. C. COCO, «COCO Dataset Explorer,» [En línea]. Available: Disponible en: <https://cocodataset.org/#explore>. [Último acceso: 31 Agosto 2020].
- [38] H. Vasa, «Google Images Download,» [En línea]. Available: Disponible en: <https://github.com/hardikvasa/google-images-download>. [Último acceso: 31 Agosto 2020].
- [39] A. Jung, «imgaug,» [En línea]. Available: Disponible en: <https://github.com/aleju/imgaug>. [Último acceso: 31 Agosto 2020].
- [40] T. Lin, «Labellmg,» [En línea]. Available: Disponible en: <https://github.com/tzutalin/labellmg>. [Último acceso: 31 Agosto 2020].
- [41] F. L. Vigne, «Inteligencia artificial: ¿Cómo aprenden las redes neuronales?,» Microsoft Docs, 2 Mayo 2019. [En línea]. Available: Disponible en: <https://docs.microsoft.com/es-es/archive/msdn-magazine/2019/april/artificially-intelligent-how-do-neural-networks-learn#:~:text=La%20funci%C3%B3n%20de%20p%C3%A9rdida%20reduce%20toda%20la%20complejidad,permite%20pensar%20en%20su%20rendimient>. [Último acceso: 3 Septiembre 2020].



UNIVERSIDADE DA CORUÑA



Escola Politécnica Superior

**TRABAJO FIN DE GRADO
CURSO 2019/20**

*CREACIÓN DE UN MODELO DE SIMULACIÓN DE
ROBOT MÓVIL PARA CONDUCCIÓN AUTÓNOMA*

Grado en Ingeniería en Tecnologías Industriales

Documento

ANEXOS A LA MEMORIA

Índice de Anexos

1 RoboboVREP.py.....	73
2 Script asociado a traffic_light (Lua).....	77
3 line_detection.py.....	78
4 capture.py.....	81
5 robobo.config.....	82
6 object_detection.py.....	86

Anexo 1 – RoboboVREP.py

```
001 import time
002 import cv2
003 import numpy as np
004 try:
005     import vrep
006 except:
007     print('-----')
008     print('"vrep.py" could not be imported. This means very probably that')
009     print('either "vrep.py" or the remoteApi library could not be found.')
010     print('Make sure both are in the same folder as this file,')
011     print('or appropriately adjust the file "vrep.py"')
012     print('-----')
013     print('')
014
015
016 class RoboboVREP:
017
018     def __init__(self):
019         vrep.simxFinish(-1) # just in case, close all opened connections
020         self.clientID = vrep.simxStart('127.0.0.1', 19999, True, True, 5000, 5)
021         if self.clientID != -1:
022             print('Connected to remote API server')
023         else:
024             print('Failed connecting to remote API server')
025
026         self.cameraHandler = 0
027
028         # aqui ya le llegan los datos con los tipos que requiere
029     def moveWheelsByTime(self, rspeed, lspeed, duration):
030
031         inputIntegers = [lspeed, rspeed]
032         inputFloats = [duration]
033         inputStrings = []
034         inputBuffer = bytearray()
035         vrep.simxCallScriptFunction(self.clientID, 'Robobo', vrep.sim_scripttype_childscript,
036                                     'moveWheelsByTime', inputIntegers, inputFloats,
037                                     inputStrings, inputBuffer, vrep.simx_opmode_blocking)
038         resultado = vrep.simxGetIntegerSignal(self.clientID,
039                                               'Bloqueado', vrep.simx_opmode_blocking)
040         while resultado[1]:
041             time.sleep(0.1)
042             resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
043                                                   vrep.simx_opmode_blocking)
044
045     def moveWheels(self, rspeed, lspeed):
046
047         inputIntegers = [lspeed, rspeed]
048         inputFloats = []
049         inputStrings = []
050         inputBuffer = bytearray()
051         vrep.simxCallScriptFunction(self.clientID, 'Robobo', vrep.sim_scripttype_childscript,
052                                     'moveWheels', inputIntegers, inputFloats,
053                                     inputStrings, inputBuffer, vrep.simx_opmode_blocking)
054
055     def moveWheelsByDegrees(self, wheel, degrees, speed):
```

```
056
057     inputIntegers=[degrees, speed]
058     inputFloats=[]
059     inputStrings = [wheel]
060     inputBuffer = bytearray()
061     vrep.simxCallScriptFunction(self.clientID, 'Robobo', vrep.sim_scripttype_childscript,
062                               'moveWheelsByDegrees', inputIntegers, inputFloats,
063                               inputStrings, inputBuffer, vrep.simx_opmode_blocking)
064     resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
065                                          vrep.simx_opmode_blocking)
066     while resultado[1]:
067         time.sleep(0.1)
068         resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
069                                              vrep.simx_opmode_blocking)
070
071     def movePanTo(self, degrees, speed):
072         inputIntegers=[degrees,speed]
073         inputFloats=[]
074         inputStrings = []
075         inputBuffer = bytearray()
076         vrep.simxCallScriptFunction(self.clientID, 'Robobo', vrep.sim_scripttype_childscript,
077                                   'movePanTo', inputIntegers, inputFloats,
078                                   inputStrings, inputBuffer, vrep.simx_opmode_blocking)
079         resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
080                                              vrep.simx_opmode_blocking)
081         while resultado[1]:
082             time.sleep(0.1)
083             resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
084                                                  vrep.simx_opmode_blocking)
085
086     def moveTiltTo(self, degrees, speed):
087         inputIntegers=[degrees,speed]
088         inputFloats=[]
089         inputStrings = []
090         inputBuffer = bytearray()
091         vrep.simxCallScriptFunction(self.clientID, 'Robobo', vrep.sim_scripttype_childscript,
092                                   'moveTiltTo', inputIntegers, inputFloats,
093                                   inputStrings, inputBuffer, vrep.simx_opmode_blocking)
094         resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
095                                              vrep.simx_opmode_blocking)
096         while resultado[1]:
097             time.sleep(0.1)
098             resultado = vrep.simxGetIntegerSignal(self.clientID, 'Bloqueado',
099                                                  vrep.simx_opmode_blocking)
100
101     def readAllIRSensor(self):
102         inputIntegers=[]
103         inputFloats=[]
104         inputStrings = []
105         inputBuffer = bytearray()
106         array=vrep.simxCallScriptFunction(self.clientID, 'Robobo',
107                                          vrep.sim_scripttype_childscript,
108                                          'readAllIRSensor', inputIntegers, inputFloats,
109                                          inputStrings, inputBuffer, vrep.simx_opmode_blocking)
110         return array[1]
111         # (IR_Front_C, IR_Front_L, IR_Front_R, IR_Front_L_FLOOR, IR_Front_R_FLOOR, IR_Back_C,
112         IR_Back_L, IR_Back_R)
113
114     def readPanPosition(self):
```

```

114     inputIntegers=[]
115     inputFloats=[]
116     inputStrings = []
117     inputBuffer = bytearray()
118     array=vrep.simxCallScriptFunction(self.clientID, 'Robobo',
vrep.sim_scripttype_childscript,
119         'readPanPosition', inputIntegers, inputFloats,
120         inputStrings, inputBuffer, vrep.simx_opmode_blocking)
121     return array[1][0]
122
123 def readTiltPosition(self):
124     inputIntegers=[]
125     inputFloats=[]
126     inputStrings = []
127     inputBuffer = bytearray()
128     array=vrep.simxCallScriptFunction(self.clientID, 'Robobo',
129         vrep.sim_scripttype_childscript,
130         'readTiltPosition', inputIntegers, inputFloats,
131         inputStrings, inputBuffer, vrep.simx_opmode_blocking)
132     return array[1][0]
133
134 def readWheels(self):
135     inputIntegers=[]
136     inputFloats=[]
137     inputStrings = []
138     inputBuffer = bytearray()
139     array=vrep.simxCallScriptFunction(self.clientID, 'Robobo',
140         vrep.sim_scripttype_childscript,
141         'readWheel', inputIntegers, inputFloats,
142         inputStrings, inputBuffer, vrep.simx_opmode_blocking)
143     return array[1] #(posicion_Rueda_I, posicion_Rueda_D, velocidad_Rueda_I,
velocidad_Rueda_D)
144
145 def resetWheelEncoders(self):
146     inputIntegers=[]
147     inputFloats=[]
148     inputStrings = []
149     inputBuffer = bytearray()
150     vrep.simxCallScriptFunction(self.clientID, 'Robobo', vrep.sim_scripttype_childscript,
151         'resetWheelEncoders', inputIntegers, inputFloats,
152         inputStrings, inputBuffer, vrep.simx_opmode_blocking)
153
154 def startColorImage(self):
155     res, self.cameraHandler = vrep.simxGetObjectHandle(self.clientID,
156         'Smartphone_camera',
157         vrep.simx_opmode_oneshot_wait)
158     vrep.simxGetVisionSensorImage(self.clientID, self.cameraHandler, 0,
vrep.simx_opmode_streaming)
159
160 def startGrayImage(self):
161     res, self.cameraHandler = vrep.simxGetObjectHandle(self.clientID,
162         'Smartphone_camera',
163         vrep.simx_opmode_oneshot_wait)
164     vrep.simxGetVisionSensorImage(self.clientID, self.cameraHandler, 1,
vrep.simx_opmode_streaming)
165
166 def getColorImage(self):
167     err, resolution, image = vrep.simxGetVisionSensorImage(self.clientID,
168         self.cameraHandler,

```

```
170         0, vrep.simx_opmode_buffer)
171     if err == vrep.simx_return_ok:
172         img = np.array(image, dtype=np.uint8)
173         img.resize([resolution[1], resolution[0], 3])
174         img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
175         return cv2.flip(img, 0)
176     elif err == vrep.simx_return_novalue_flag:
177         print("no image yet")
178         pass
179     else:
180         print(err)
181
182     def getGrayImage(self):
183         err, resolution, image = vrep.simxGetVisionSensorImage(self.clientID,
184             self.cameraHandler,
185             1, vrep.simx_opmode_buffer)
186         if err == vrep.simx_return_ok:
187             img = np.array(image, dtype=np.uint8)
188             img.resize([resolution[1], resolution[0]])
189             return cv2.flip(img, 0)
190         elif err == vrep.simx_return_novalue_flag:
191             print("no image yet")
192             pass
193         else:
194             print(err)
195
```

Anexo 2 – Script asociado a traffic_light (Lua)

```
01 function sysCall_init()
02   red = sim.getObjectHandle("red")
03   yellow = sim.getObjectHandle("yellow")
04   green = sim.getObjectHandle("green")
05   sim.setShapeColor(green,"",sim.colorcomponent_emission,{0,0,0})
06   sim.setShapeColor(yellow,"",sim.colorcomponent_emission,{0,0,0})
07   sim.setShapeColor(red,"",sim.colorcomponent_emission,{1,0,0})
08   t=0
09 end
10
11 function sysCall_actuation()
12   if t==500 then
13     sim.setShapeColor(green,"",sim.colorcomponent_emission,{0,1,0})
14     sim.setShapeColor(red,"",sim.colorcomponent_emission,{0,0,0})
15   elseif t==900 then
16     sim.setShapeColor(yellow,"",sim.colorcomponent_emission,{1,1,0})
17     sim.setShapeColor(green,"",sim.colorcomponent_emission,{0,0,0})
18   elseif t==1000 then
19     sim.setShapeColor(red,"",sim.colorcomponent_emission,{1,0,0})
20     sim.setShapeColor(yellow,"",sim.colorcomponent_emission,{0,0,0})
21     t=0
22   end
23   t=t+1
24 end
25
26 function sysCall_sensing()
27   -- put your sensing code here
28 end
29
30 function sysCall_cleanup()
31   -- do some clean-up here
32 end
```

Anexo 3 – line_detection.py

```
001 # Import packages
002 import numpy as np
003 import cv2 as cv
004 import time
005
006 # Import vrep/coppeliasim sync
007 import RoboboVREP
008 robobo = RoboboVREP.RoboboVREP()
009
010
011 def do_canny(gray):
012     # Applies a 5x5 gaussian blur with deviation of 0 to frame
013     blur = cv.GaussianBlur(gray, (5, 5), 0)
014     # Applies Canny edge detector with minVal of 50 and maxVal of 150
015     canny = cv.Canny(blur, 50, 150)
016     return canny
017
018
019 def do_segment(frame):
020     # Since an image is a multi-directional array containing the relative intensities of
021     # each pixel in the image, we can use frame.shape to return a tuple:
022     # [number of rows, number of columns, number of channels] of the dimensions of the frame
023     # Since height begins from 0 at the top, the y-coordinate of the bottom of the frame is its
024     height
025     # Creates a triangular polygon for the mask defined by three (x,y) coordinates
026     polygons = np.array([
027         [(0, 512), (512, 512), (256, 128)]
028     ])
029     # Creates an image filled with zero intensities with the same dimensions as the frame
030     mask = np.zeros_like(frame)
031     # Allows the mask to be filled with values of 1 and the other areas to be filled with values of 0
032     cv.fillPoly(mask, polygons, 255)
033     # A bitwise and operation between the mask and frame keeps only the triangular area of the
034     frame
035     segment = cv.bitwise_and(frame, mask)
036     return segment
037
038
039 def calculate_lines(frame, lines):
040     # Empty arrays to store the coordinates of the left and right lines
041     left = []
042     right = []
043     # Loops through every detected line
044     if lines is not None:
045         for line in lines:
046             # Reshapes line from 2D array to 1D array
047             x1, y1, x2, y2 = line.reshape(4)
048             # Fits a linear polynomial to the x and y coordinates and
049             # returns a vector of coefficients which describe the slope and y-intercept
050             parameters = np.polyfit((x1, x2), (y1, y2), 1)
051             slope = parameters[0]
052             y_intercept = parameters[1]
053             # If slope is negative, the line is to the left of the lane, otherwise, the line is to the right of
054             the lane
055             if slope < 0:
056                 left.append((slope, y_intercept))
057             else:
058                 right.append((slope, y_intercept))
059     return left, right
```

```
054     else:
055         right.append((slope, y_intercept))
056     # Averages out all the values for left and right into a single slope and y-intercept value for
each line
057     left_avg = np.average(left, axis=0)
058     right_avg = np.average(right, axis=0)
059     # Calculates the x1, y1, x2, y2 coordinates for the left and right lines
060     left_line = calculate_coordinates(frame, left_avg)
061     right_line = calculate_coordinates(frame, right_avg)
062     return np.array([left_line, right_line])
063
064
065 def calculate_coordinates(frame, parameters):
066     if type(parameters) is np.float64 or type(parameters) is None:
067         x1 = y1 = x2 = y2 = 0
068     else:
069         slope, intercept = parameters
070         # Sets initial y-coordinate as height from top down (bottom of the frame)
071         y1 = frame.shape[0]
072         # Sets final y-coordinate as 150 above the bottom of the frame
073         y2 = int(y1 - 150)
074         # Sets initial x-coordinate as (y1 - b) / m since y1 = mx1 + b
075         x1 = int((y1 - intercept) / slope)
076         # Sets final x-coordinate as (y2 - b) / m since y2 = mx2 + b
077         x2 = int((y2 - intercept) / slope)
078     return np.array([x1, y1, x2, y2])
079
080
081 def visualize_lines(frame, lines):
082     # Creates an image filled with zero intensities with the same dimensions as the frame
083     lines_visualize = np.zeros_like(frame)
084     # Checks if any lines are detected
085     if lines is not None:
086         for x1, y1, x2, y2 in lines:
087             # Draws lines between two coordinates with green color and 5 thickness
088             cv.line(lines_visualize, (x1, y1), (x2, y2), (0, 255, 0), 5)
089     return lines_visualize
090
091
092 def do_lines(frame):
093     gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
094     canny = do_canny(gray)
095     segment = do_segment(canny)
096     hough = cv.HoughLinesP(segment, 2, np.pi / 180, 100, np.array([]),
097                             minLineLength=100, maxLineGap=50)
098     # Averages multiple detected lines from hough into one line for left border of lane and one line
for right border
099     lines = calculate_lines(frame, hough)
100     # Visualizes the lines
101     lines_visualize = visualize_lines(frame, lines)
102     # Overlays lines on frame by taking their weighted sums and
103     # adding an arbitrary scalar value of 1 as the gamma argument
104     frame = cv.addWeighted(frame, 0.9, lines_visualize, 1, 1)
105     return frame
106
107
108 # Abrimos el acceso a la cámara y ajustamos la inclinación del smartphone
109 # para obtener una buena perspectiva de la carretera
110 robobo.startColorImage()
```

```
111 robobo.moveTiltTo(105, 70)
112 while True:
113     start_time = time.time() # start time of the loop
114     frame = robobo.getColorImage()
115     frame = do_lines(frame)
116     cv.imshow('Smartphone camera', frame)
117     print("FPS: ", 1.0 / (time.time() - start_time)) # FPS = 1 / time to process loop
118     # El programa se termina cuando el usuario presiona la tecla 'q'
119     if cv.waitKey(1) & 0xFF == ord('q'):
120         cv.destroyAllWindows()
121         break
122
```


Anexo 4 – capture.py

```
01 import RoboboVREP
02 import cv2 as cv
03
04 robobo = RoboboVREP.RoboboVREP()
05
06 robobo.startColorImage()
07 robobo.moveTiltTo(105, 70)
08 i = 1
09 while True:
10     frame = robobo.getColorImage()
11     # Opens a new window and displays the output frame
12     cv.imshow("smartphone camera", frame)
13     cv.imwrite('images/e{}.png'.format(i), frame)
14     print(i)
15     i = i + 1
16     # The program breaks out of the while loop when the user presses the 'q' key
17     # The program also waits and takes a capture every 3 seconds
18     if cv.waitKey(3000) & 0xFF == ord('q'):
19         cv.destroyAllWindows()
20         break
21
```

Anexo 5 – robobo.config

```
001 model {
002   ssd {
003     num_classes: 6
004     image_resizer {
005       fixed_shape_resizer {
006         height: 300
007         width: 300
008       }
009     }
010     feature_extractor {
011       type: "ssd_mobilenet_v3"
012       depth_multiplier: 1.0
013       min_depth: 16
014       conv_hyperparams {
015         regularizer {
016           l2_regularizer {
017             weight: 3.99999989895e-05
018           }
019         }
020         initializer {
021           truncated_normal_initializer {
022             mean: 0.0
023             stddev: 0.0299999993294
024           }
025         }
026         activation: RELU_6
027         batch_norm {
028           decay: 0.999700009823
029           center: true
030           scale: true
031           epsilon: 0.0010000000475
032           train: true
033         }
034       }
035       use_depthwise: true
036     }
037     box_coder {
038       faster_rcnn_box_coder {
039         y_scale: 10.0
040         x_scale: 10.0
041         height_scale: 5.0
042         width_scale: 5.0
043       }
044     }
045     matcher {
046       argmax_matcher {
047         matched_threshold: 0.5
048         unmatched_threshold: 0.5
049         ignore_thresholds: false
050         negatives_lower_than_unmatched: true
051         force_match_for_each_row: true
052       }
053     }
054     similarity_calculator {
055       iou_similarity {
```

```
056     }
057   }
058   box_predictor {
059     convolutional_box_predictor {
060       conv_hyperparams {
061         regularizer {
062           l2_regularizer {
063             weight: 3.99999989895e-05
064           }
065         }
066         initializer {
067           truncated_normal_initializer {
068             mean: 0.0
069             stddev: 0.0299999993294
070           }
071         }
072         activation: RELU_6
073         batch_norm {
074           decay: 0.999700009823
075           center: true
076           scale: true
077           epsilon: 0.0010000000475
078           train: true
079         }
080       }
081       min_depth: 0
082       max_depth: 0
083       num_layers_before_predictor: 0
084       use_dropout: false
085       dropout_keep_probability: 0.800000011921
086       kernel_size: 3
087       box_code_size: 4
088       apply_sigmoid_to_scores: false
089       use_depthwise: true
090     }
091   }
092   anchor_generator {
093     ssd_anchor_generator {
094       num_layers: 6
095       min_scale: 0.20000000298
096       max_scale: 0.949999988079
097       aspect_ratios: 1.0
098       aspect_ratios: 2.0
099       aspect_ratios: 0.5
100       aspect_ratios: 3.0
101       aspect_ratios: 0.333299994469
102     }
103   }
104   post_processing {
105     batch_non_max_suppression {
106       score_threshold: 0.300000011921
107       iou_threshold: 0.600000023842
108       max_detections_per_class: 100
109       max_total_detections: 100
110     }
111     score_converter: SIGMOID
112   }
113   normalize_loss_by_num_matches: true
114   loss {
```

```
115     localization_loss {
116         weighted_smooth_l1 {
117         }
118     }
119     classification_loss {
120         weighted_sigmoid {
121         }
122     }
123     hard_example_miner {
124         num_hard_examples: 3000
125         iou_threshold: 0.990000009537
126         loss_type: CLASSIFICATION
127         max_negatives_per_positive: 3
128         min_negatives_per_image: 3
129     }
130     classification_weight: 1.0
131     localization_weight: 1.0
132 }
133 }
134 }
135 train_config {
136     batch_size: 2
137     data_augmentation_options {
138         random_horizontal_flip {
139         }
140     }
141     data_augmentation_options {
142         ssd_random_crop {
143         }
144     }
145     optimizer {
146         rms_prop_optimizer {
147             learning_rate {
148                 exponential_decay_learning_rate {
149                     initial_learning_rate: 0.00400000018999
150                     decay_steps: 800720
151                     decay_factor: 0.9499999988079
152                 }
153             }
154             momentum_optimizer_value: 0.899999976158
155             decay: 0.899999976158
156             epsilon: 1.0
157         }
158     }
159     #fine_tune_checkpoint: "PATH_TO_BE_CONFIGURED/model.ckpt"
160     num_steps: 20000
161     #fine_tune_checkpoint_type: "detection"
162 }
163 train_input_reader {
164     label_map_path:
"C:/Users/Juanatey/tensorflow/models/research/object_detection/training/robobo_label_map.pb
txt"
165     tf_record_input_reader {
166         input_path:
"C:/Users/Juanatey/tensorflow/models/research/object_detection/train.record"
167     }
168 }
169 eval_config {
170     num_examples: 50
```

```
171  max_evals: 10
172  use_moving_averages: false
173  }
174  eval_input_reader {
175    label_map_path:
"C:/Users/Juanatey/tensorflow/models/research/object_detection/training/robobo_label_map.pb
txt"
176    shuffle: false
177    num_readers: 1
178    tf_record_input_reader {
179      input_path:
"C:/Users/Juanatey/tensorflow/models/research/object_detection/test.record"
180    }
181  }
```

Anexo 6 – object_detection.py

```
01 # Import packages
02 import numpy as np
03 import tensorflow as tf
04 import cv2 as cv
05 import time
06
07 # Import utilities
08 from utils import label_map_util
09 from utils import visualization_utils as vis_util
10
11 # Import vrep/coppeliasim sync
12 import RoboboVREP
13 robobo = RoboboVREP.RoboboVREP()
14
15 ## Model preparation
16 # Any model exported using the `export_inference_graph.py` tool
17 # can be loaded here simply by changing the path.
18 # By default we use an "SSDLite Mobile net" based model.
19 # See the detection model zoo for a list of models that can be
20 # run out-of-the-box with varying speeds and accuracies.
21
22 # Name of the directory containing the object detection module we're using
23 # model_name = 'inference_graph' by default
24 model_name = 'robobo'
25 # Path to frozen detection graph .pb file,
26 # which contains the model that is used for object detection.
27 PATH_TO_CKPT = '{}/frozen_inference_graph.pb'.format(model_name)
28
29 ## Loading label map
30 # Label maps map indices to category names, so that when our convolution network predicts `5`,
31 # we know that this corresponds to `airplane`. Here we use internal utility functions,
32 # but anything that returns a dictionary mapping integers to appropriate string labels would be
33 fine.
34 # List of the strings that is used to add correct label for each box.
35 # PATH_TO_LABELS = 'data/mscoco_robobo_label_map.pbtxt' by default
36 PATH_TO_LABELS = 'robobo/robobo_label_map.pbtxt'
37
38 category_index = label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
39 use_display_name=True)
40
41 # Load the model into memory.
42 detection_graph = tf.Graph()
43 with detection_graph.as_default():
44     od_graph_def = tf.GraphDef()
45     with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
46         serialized_graph = fid.read()
47         od_graph_def.ParseFromString(serialized_graph)
48         tf.import_graph_def(od_graph_def, name='')
49     sess = tf.Session(graph=detection_graph)
50
51 ## Define input and output tensors (i.e. data) for the object detection classifier
52 # Input tensor is the image
53 image_tensor = detection_graph.get_tensor_by_name('image_tensor:0')
54 # Output tensors are the detection boxes, scores, and classes
55 detection_boxes = detection_graph.get_tensor_by_name('detection_boxes:0)
```

```
55 detection_scores = detection_graph.get_tensor_by_name('detection_scores:0')
56 detection_classes = detection_graph.get_tensor_by_name('detection_classes:0')
57 num_detections = detection_graph.get_tensor_by_name('num_detections:0')
58
59 robobo.startColorImage()
60 robobo.moveTiltTo(105, 70)
61 while True:
62     start_time = time.time() # start time of the loop
63     frame = robobo.getColorImage()
64     # Acquire frame and expand frame dimensions to have shape: [1, None, None, 3]
65     # i.e. a single-column array, where each item in the column has the pixel RGB value
66     frame_rgb = cv.cvtColor(frame, cv.COLOR_BGR2RGB)
67     frame_expanded = np.expand_dims(frame_rgb, axis=0)
68     # Perform the actual detection by running the model with the image as input
69     (boxes, scores, classes, num) = sess.run(
70         [detection_boxes, detection_scores, detection_classes, num_detections],
71         feed_dict={image_tensor: frame_expanded})
72     # Visualization of the results of a detection
73     vis_util.visualize_boxes_and_labels_on_image_array(
74         frame,
75         np.squeeze(boxes),
76         np.squeeze(classes).astype(np.int32),
77         np.squeeze(scores),
78         category_index,
79         use_normalized_coordinates=True,
80         line_thickness=4,
81         min_score_thresh=0.70)
82     cv.imshow('Smartphone camera', frame)
83     print("FPS: ", 1.0 / (time.time() - start_time)) # FPS = 1 / time to process loop
84     # The program breaks out of the while loop when the user presses the 'q' key
85     if cv.waitKey(1) & 0xFF == ord('q'):
86         cv.destroyAllWindows()
87         break
88
```



UNIVERSIDADE DA CORUÑA



Escola Politécnica Superior

**TRABAJO FIN DE GRADO
CURSO 2019/20**

*CREACIÓN DE UN MODELO DE SIMULACIÓN DE
ROBOT MÓVIL PARA CONDUCCIÓN AUTÓNOMA*

Grado en Ingeniería en Tecnologías Industriales

Documento

PRESUPUESTO

C.I CAPÍTULO I. MATERIALES					
N/P	CONCEPTO	Uds.	Cantidad	Precio (€)	Importe (€)
1.1	Ordenador de sobremesa con Intel Core i5-2500 y 8 GB de RAM		1	500,00 €	500,00 €
1.2	Disco duro SSD de 480 GB		1	50,00 €	50,00 €
1.3	Tarjeta gráfica NVIDIA GeForce GTX 1050 Ti		1	145,00 €	145,00 €
	IMPORTE TOTAL CAPÍTULO I				695,00 €

C.II CAPÍTULO II. SOFTWARE					
N/P	CONCEPTO	Uds.	Cantidad	Precio (€)	Importe (€)
2.1	Licencia CoppeliaSim Edu (gratuita para uso educativo)		1	0,00 €	0,00 €
	IMPORTE TOTAL CAPÍTULO II				0,00 €

C.III CAPÍTULO III. MANO DE OBRA					
N/P	CONCEPTO	Uds.	Cantidad	Precio (€)	Importe (€)
3.1	Desarrollo del modelo	Horas	360 h	40,00 €/h	14 400,00 €
	IMPORTE TOTAL CAPÍTULO III				14 400,00 €

Resumen por capítulos	
Capítulo I. Materiales	695,00 €
Capítulo II. Software	0,00 €
Capítulo III. Mano de obra	14 400,00 €
IMPORTE DE EJECUCIÓN MATERIAL	15 095,00 €

IMPORTE DE EJECUCIÓN MATERIAL	15 095,00 €
13% de Gastos Generales	1 962,35 €
6% de Beneficio Industrial	905,70 €
IMPORTE DE EJECUCION	17 963,05 €
21% de IVA	3 772,24 €
IMPORTE DE CONTRATA	21 735,29 €

El importe de contrata asciende a la cantidad de **VEINTIÚN MIL SETECIENTOS TREINTA Y CINCO EUROS CON VEINTINUEVE CÉNTIMOS.**