



UNIVERSIDAD
DE MÁLAGA



ESCUELA DE INGENIERÍAS INDUSTRIALES

Ingeniería de Sistemas y Automática

TRABAJO FIN DE MÁSTER

**DESARROLLO DE UN SIMULADOR PARA EL DISEÑO DE SISTEMAS
DE PERCEPCIÓN Y CONTROL DE VEHÍCULOS AUTÓNOMOS EN EL
ENTORNO DE LA AMPLIACIÓN DEL CAMPUS DE TEATINOS**

**DEVELOPMENT OF A SIMULATOR FOR THE DESIGN OF
PERCEPTION AND CONTROL SYSTEMS OF AUTONOMOUS VEHICLES
IN THE AREA OF THE EXPANSION OF THE TEATINOS CAMPUS**

Máster en Ingeniería Mecatrónica

Autor: ELOY VERGARA GÓMEZ

Tutor: JUAN JESÚS FERNÁNDEZ LOZANO

Cotutor: JESÚS MORALES RODRÍGUEZ

MÁLAGA, JULIO de 2.020

Resumen:

Este trabajo se ocupa del desarrollo de un simulador para el diseño y control de vehículos autónomos en el entorno de la ampliación del Campus de Teatinos. La introducción de simuladores en una de las principales líneas de investigación del sector de la automoción como es el desarrollo de vehículos autónomos resulta muy interesante, ya que permite la realización de pruebas muy realistas.

Para la realización de pruebas de conducción autónoma en el simulador *CARLA Simulator*, se ha modelado la ampliación universitaria del Campus de Teatinos con el objetivo de experimentar el comportamiento del simulador en un entorno conocido. Para ello, se han utilizado vehículos autónomos con una configuración de sensores similares a los que disponen los coches comerciales actualmente. Se han utilizado varios LIDAR 3D y cámaras RGB para la visualización del entorno del vehículo, sensores inerciales para la detección de colisiones, sensores de invasión de líneas y un sensor GNSS (*Global Navigation Satellite System, Sistema global de navegación por satélite*, en español) para conocer la posición actual del vehículo. Además, se han probado otros sensores que podrían incorporarse en vehículos para la mejora de la conducción autónoma, como una cámara de profundidad y de segmentación semántica. Las pruebas han consistido en conducir por la zona universitaria bajo condiciones ambientales diversas (despejado, nublado, lluvioso, etc.) y a distintas horas del día (amanecer, mañana, atardecer y noche).

Los resultados obtenidos han mostrado el buen comportamiento del vehículo autónomo con los sensores utilizados, así como una estimación del coste computacional de la simulación, observando numerosas ventajas en la utilización de simuladores de conducción autónoma. Además, se ha almacenado y visualizado la información sensorial obtenida durante las simulaciones. Esto es necesario para que el simulador se pueda utilizar en el diseño del sistema de navegación de vehículos autónomos.

Palabras clave: simulador, vehículos autónomos, sensores, LIDAR 3D, cámaras RGB, sensores inerciales, sensor GNSS, coste computacional.

Abstract:

In this thesis has carried out the development of a simulator for the design and control of autonomous vehicles in the vicinity of the expansion of the Teatinos Campus. Introduction of simulators is one of the main lines of research in the automotive sector, such as the development of autonomous vehicles, is very interesting, since it allows the realization of very realistic tests.

To carry out driving tests in *CARLA Simulator*, the university expansion of the Teatinos Campus has been modeled in order to test the simulator's behavior in a known environment. For this, autonomous vehicles have been used with a sensor configuration similar to currently available for commercial cars. Several 3D LIDAR and RGB cameras have been used to visualize the vehicle environment, inertial sensors for collision detection, line invasion sensors and a GNSS (*Global Satellite Navigation System*) sensor to know the current position of the vehicle. In addition, other sensors have been tested that could be incorporated in vehicles for the improvement of autonomous driving such as a depth and semantic segmentation camera. The tests have consisted of driving around the university area under various weather conditions (clear, cloudy, rainy, etc.) and at different times of the day (sunrise, morning, sunset and night).

The obtained results have shown the good behavior of the autonomous vehicle with the sensors used, as well as an estimation of the computational cost of the simulation, observing numerous advantages in the use of autonomous driving simulators. In addition, sensory information obtained during simulations has been stored and visualized. This is necessary so that the simulator can be used in the design of the autonomous vehicle navigation systems.

Keywords: simulator, autonomous vehicles, sensors, 3D LIDAR, RGB cameras, inertial sensors, GNSS sensor, computational cost.

Índice general

Resumen:	VII
Abstract:	VIII
Índice general	IX
Índice de figuras.....	XIII
1. Introducción	1
1.1. Motivación	1
1.2. Objetivos	2
1.3. Estructura del proyecto.....	2
2. Antecedentes.....	5
2.1. Introducción	5
2.2. Vehículos autónomos	5
2.2.1. Historia de los vehículos autónomos	6
2.2.2. Vehículos autónomos en la actualidad.....	9
2.2.3. Niveles de autonomía.....	14
2.3. Simulación	17
2.4. Simulador CARLA	18
2.4.1. Estructura	19
2.4.2. Entorno.....	22
2.4.3. Cámaras y sensores disponibles.....	23
3. Modelado del entorno de pruebas	31
3.1. Introducción	31
3.2. Modelado en RoadRunner	33
3.2.1. Herramientas de RoadRunner	34

3.2.2.	Creación de la ampliación del Campus de Teatinos	40
3.3.	Modelado en CARLA.....	47
3.3.1.	Importación en CARLA.....	48
3.3.2.	Ajuste de los errores de importación	49
3.3.3.	Adición de componentes estáticos.....	52
4.	Implementación.....	61
4.1.	Introducción.....	61
4.2.	Configuración de la simulación en tiempo real	62
4.2.1.	Creación del mundo	64
4.2.2.	Control del teclado.....	67
4.2.3.	Configuración de la pantalla	69
4.2.4.	Configuración del vehículo.....	72
4.2.5.	Guiado autónomo de CARLA	74
4.3.	Configuración del sistema de almacenamiento de datos sensoriales....	75
4.3.1.	Almacenamiento de datos.....	75
4.3.2.	Procesamiento de datos en Matlab.....	78
5.	Pruebas y validación	83
5.1.	Introducción.....	83
5.2.	Simulación en tiempo real	84
5.3.	Estudio de los datos sensoriales.....	96
5.3.1.	Prueba 1	96
5.3.2.	Prueba 2	104
5.3.3.	Captura de imágenes y LIDAR.....	112
5.4.	Coste computacional	115
6.	Conclusiones y trabajos futuros	119
6.1.	Conclusiones	119
6.2.	Trabajos futuros.....	121
Anexo.....		123
A.1.	Instalación de CARLA en Ubuntu	123
A.2.	Manual_control.py.....	125

A.3. WheeledVehicleAIController.cpp.....	141
A.4. RoutePlanner.cpp.....	146
A.5. Código Matlab	150
Referencias	155

Índice de figuras

Figura 1. Furgoneta Mercedes del proyecto Eureka Prometheus [4].....	7
Figura 2. ParkShuttle [5]	8
Figura 3. Sensores utilizados por los sistemas de conducción autónoma [8]	9
Figura 4. Radar de corto alcance en la parte trasera del vehículo [9]	11
Figura 5. LIDAR situado en el techo del coche [10]	12
Figura 6. Cámaras frontales y laterales de un vehículo de la marca Tesla [11].....	13
Figura 7. Módulos de CARLA [14]	20
Figura 8. Unreal Editor	20
Figura 9. Ejecución del script de condiciones meteorológicas cambiantes	21
Figura 10. Ventanas <i>Pygame</i> con visión normal (arriba) y con cámara de profundidad (abajo)	22
Figura 11. Cámara RGB colocada en la parte superior del vehículo	24
Figura 12. Cámara de profundidad colocada en la parte superior del vehículo. Original (arriba) y convertida a escala de grises (abajo).....	25
Figura 13. Visualización de la cámara de segmentación semántica colocada en la parte superior del vehículo.....	26
Figura 14. Puntos obtenidos mediante el LIDAR [15]	27
Figura 15. Indicadores del sensor de colisión	28
Figura 16. Sensor de invasión de línea.....	29
Figura 17. Indicador de posición GNSS	29
Figura 18. Sensor IMU.....	30
Figura 19. Posición de la ampliación universitaria del Campus de Teatinos en Málaga	32
Figura 20. Ampliación universitaria del Campus de Teatinos y alrededores	32
Figura 21. Coordenadas y altitud en Google Earth	33
Figura 22. Herramienta para medir distancias en Google Earth	34
Figura 23. Herramienta para modelar carreteras en RoadRunner	35
Figura 24. Herramienta para el modelado de rotondas	35
Figura 25. Configuración de la altitud	36

Figura 26. Configuración de carril	37
Figura 27. Configuración de línea de carril.....	37
Figura 28. Modificación de la anchura del carril completo.....	38
Figura 29. Modificación parcial de la anchura del carril.....	38
Figura 30. Medidas de longitud.....	39
Figura 31. Herramienta para añadir objetos	40
Figura 32. Comparación de la longitud de la carretera delantera de la Escuela de Ingenierías. Medida de Google Earth arriba y medida en RoadRunner abajo.....	41
Figura 33. Comparación de la altitud de la carretera este de la Escuela de Ingenierías. Medida de Google Earth arriba y medida en RoadRunner abajo.....	42
Figura 34. Comparación del entorno real y el entorno virtual creado.....	43
Figura 35. Comparación de medidas de edificios en Google Earth (arriba) y RoadRunner (abajo)	44
Figura 36. Comparación 3D de la zona de las universidades en Google Earth (arriba) y RoadRunner (abajo)	45
Figura 37. Visualización de toda la ampliación utilizando la herramienta OpenDRIVE Preview Tool	46
Figura 38. Visualización de la zona de la Facultad de Ciencias de la Salud y la Facultad de Estudios Sociales y del Trabajo utilizando la herramienta OpenDRIVE Preview Tool	46
Figura 39. Salidas e incorporaciones de la rotonda norte que une la Escuela de Ingenierías y la Facultad de Estudios Sociales y del trabajo.....	47
Figura 40. Exportación de ficheros en RoadRunner.....	48
Figura 41. Mapa de la ampliación universitaria importado a CARLA	50
Figura 42. RoutePlanners y puntos intermedios en una rotonda del mapa.....	51
Figura 43. Ruta incompleta en la incorporación a una rotonda.....	52
Figura 44. Ruta corregida en la incorporación a una rotonda	52
Figura 45. Edificio de la Escuela de Ingenierías en CARLA (arriba) y Google Earth (abajo).....	53
Figura 46. Edificio de la Facultad de Comercio y Gestión/Facultad de Estudios Sociales y del Trabajo en CARLA (arriba) y Google Earth (abajo)	54
Figura 47. Facultad de Ciencias de la Salud en CARLA (arriba) y Google Earth (abajo).....	55
Figura 48. Edificios residenciales en CARLA (arriba) y Google Earth (abajo).....	56

Figura 49. Diferentes componentes estáticos añadidos al mapa	57
Figura 50. Vehículos estáticos aparcados y personas añadidas al mapa.....	57
Figura 51. Señal de ceda el paso	58
Figura 52. Señal de límite de velocidad y su configuración	59
Figura 53. Semáforo y configuración.....	60
Figura 54. Evolución del mapa de RoadRunner a CARLA	60
Figura 55. Iniciación de sesión de juego en el editor	63
Figura 56. Controles en el programa de simulación en tiempo real.....	68
Figura 57. Vehículo circulando en una rotonda de la ampliación.....	85
Figura 58. Incorporación a una rotonda	86
Figura 59. Comportamiento del vehículo ante un semáforo	87
Figura 60. Adaptación de la velocidad del vehículo sin obstáculos	88
Figura 61. Comportamiento ante obstáculos en la carretera.....	89
Figura 62. Comportamiento del vehículo ante cambios climatológicos	90
Figura 63. Comportamiento de la cámara y el LIDAR en diferentes condiciones climatológicas	91
Figura 64. Cámaras disponibles en el vehículo autónomo.....	93
Figura 65. Imágenes del radar durante la simulación	94
Figura 66. Simulación de una colisión.....	95
Figura 67. Recorrido de la primera simulación en conducción autónoma.....	96
Figura 68. Recorrido del vehículo obtenido mediante la localización de CARLA (arriba) y las coordenadas geográficas.....	97
Figura 69. Recorrido de la simulación automática en 3D.....	98
Figura 70. Velocidad (arriba), pedales de aceleración y freno (centro), aceleración lineal en X (abajo).....	99
Figura 71. Momento en el que el vehículo detecta una señal de límite 50 km/h	99
Figura 72. Intensidad pedales de aceleración y freno (grafica 1), aceleración lineal (gráfica 2), movimiento rotacional (gráfica 3) e intensidad de giro (gráfica 4)	100
Figura 73. Momento en el que el vehículo entra en la rotonda.....	101
Figura 74. Azimut frente a altitud con gradiente de profundidad	102
Figura 75. Detección de edificios y vehículos aparcados por el radar.....	102
Figura 76. Azimut frente a altitud con gradiente de velocidad hacia el radar	103
Figura 77. Detección de un vehículo por el radar	104
Figura 78. Recorrido en la prueba de conducción manual.....	105

Figura 79. Recorridos de la prueba de simulación manual a partir de la localización (arriba) y las coordenadas geográficas (abajo).....	106
Figura 80. Recorrido de la simulación manual en 3D	106
Figura 81. Velocidad (arriba), pedales de aceleración y freno (centro), aceleración lineal en X (abajo) en la prueba de conducción manual	107
Figura 82. Intensidad pedales de aceleración y freno (grafica 1), aceleración lineal (gráfica 2), movimiento rotacional (gráfica 3) e intensidad de giro (gráfica 4) en la prueba de simulación manual.....	108
Figura 83. Momento de la colisión en el segundo 62 de la simulación de conducción manual	108
Figura 84. Azimut frente a altitud con gradiente de profundidad en la prueba de conducción manual.....	109
Figura 85. Azimut frente a altitud con gradiente de velocidad en la prueba de conducción manual.....	110
Figura 86. Colisiones ocurridas durante la simulación manual.....	110
Figura 87. Impulso e intensidad detectada por el sensor de colisión	112
Figura 88. Captura de la cámara y mapa del LIDAR en la entrada al parking de la Escuela de Ingenierías	113
Figura 89. Captura de la cámara y mapa del LIDAR frente a la Facultad de Ciencias de la Salud	114
Figura 90. Captura de la cámara y mapa del LIDAR frente a la Facultad de Ciencias de la Salud en dirección oeste	115

Capítulo 1

Introducción

1.1. Motivación

En la actualidad, una de las principales líneas de investigación en el sector de la automoción es el desarrollo de vehículos autónomos. Aunque ya hay gran cantidad de vehículos capaces de percibir el entorno (gracias a la gran cantidad de sensores que poseen) y tomar decisiones comparables con la que toma un humano, aún es un tema abierto a la investigación, que requerirá más esfuerzos hasta llegar a la autonomía completa del vehículo, sin necesidad de intervención humana. Para describir el grado de autonomía de un vehículo se suele aludir a la clasificación que propone la SAE (Society of Automotive Engineers, Sociedad de Ingenieros de la Automoción, en español), que establece seis niveles de conducción autónoma que van desde coches que únicamente cuentan con sistemas de alerta (nivel 0) hasta coches en los que el conductor no tiene que intervenir en ningún momento en la conducción (nivel 5). A día de hoy, los sistemas comerciales más avanzando (como el *autopilot* de Tesla) se encuentran en el nivel 3, en los que el conductor tiene que estar preparado para intervenir en cualquier momento cuando el sistema lo solicite.

En fechas recientes, se están proponiendo técnicas prometedoras para alcanzar el nivel 5 de conducción autónoma. Entre ellas destacan las técnicas basadas en aprendizaje, como *deep learning* (DL). No obstante, estas técnicas cuentan con diversos inconvenientes en su aplicación a estos problemas. El principal radica en que las técnicas de DL obtienen buenos resultados cuando se dispone de un volumen elevado de datos. Disponer de esos

datos no es fácil en un sistema físico como un robot o un vehículo, por tiempo, por desgaste de elementos, o por la variabilidad de las condiciones. Una alternativa es la transferencia del entrenamiento desde un dominio simulado al dominio físico. Por tanto, el uso de simuladores para experimentación en conducción autónoma resulta de enorme interés por las grandes ventajas que conlleva y por su simplicidad respecto a experimentaciones reales.

1.2. Objetivos

El objetivo de este proyecto es la creación de un entorno de simulación de vehículos autónomos y el estudio de su comportamiento. Para ello, se modelará la zona de la ampliación del Campus de Teatinos, usando como base el simulador de vehículos autónomos “Carla Simulator”. Esto permitirá disponer de una plataforma en la que se puedan realizar simulaciones que permitan obtener datos para el desarrollo de sistemas de percepción y control para vehículos autónomos, en una fase posterior. Con el entorno creado, se llevarán a cabo diversas pruebas de conducción a través de la ampliación universitaria con el objetivo de estudiar el comportamiento del simulador en diferentes condiciones climatológicas (nublado, despejado, lluvia, etc.), así como en distintos momentos del día (amanecer, mañana, atardecer, noche). Para la conducción autónoma, se utilizará una configuración de sensores similar a la utilizada por los vehículos comercializados en la actualidad, por lo que se dispondrá de un sensor LIDAR 3D, cámaras RGB, sensores inerciales para la detección de colisiones, sensores de invasión de línea y un sensor GNSS (Sistema global de navegación por satélite) para conocer la posición actual del vehículo. Finalmente, se estudiará el comportamiento general del simulador, así como el coste computacional de las simulaciones para obtener una idea de la viabilidad de estos sistemas como medio de estudio en el sector de la conducción autónoma.

1.3. Estructura del proyecto

El trabajo se ha estructurado de la siguiente forma:

- En el capítulo dos se describen los antecedentes, donde se hará una pequeña revisión de la actualidad de los vehículos autónomos y se explicará en detalle el simulador *CARLA Simulator*.
- En el capítulo 3 se describe el proceso de modelado de la zona de la ampliación universitaria del Campus de Teatinos mediante el software *RoadRunner* (de VectorZero y MathWorks) y el simulador *CARLA Simulator*.
- En el cuarto capítulo se presenta la implementación de la simulación desde su configuración, hasta la configuración del mapa y los sensores dispuestos en el vehículo autónomo.
- En el quinto capítulo se presentan la experimentación y la validación de los resultados, donde se explicarán las diferentes pruebas realizadas en el simulador.
- En el capítulo 6 se muestran las conclusiones y los trabajos futuros.

Capítulo 2

Antecedentes

2.1. Introducción

Desde la creación del primer vehículo con motor de combustión interna con gasolina en 1885, los esfuerzos de la industria de la automoción han estado centrados en la mejora tecnológica y de la seguridad de estos. Para lograr este objetivo, las investigaciones han estado centradas en hacer a los coches cada vez más *inteligentes*, dotándolos de sistemas capaces de sincronizarse con el humano para mejorar la seguridad en la conducción (sensores de fatiga, detectores de línea, velocidad de cruce adaptativa, etc.). En la actualidad, el reto es lograr la total autonomía del vehículo dotándolo de sensores de última generación y con sistemas inteligentes que proporcionen al vehículo la capacidad de toma de decisiones para que pueda sustituir por completo al humano. Aunque ya existen vehículos con un alto nivel de autonomía, queda mucho trabajo por delante para poder lograr la autonomía completa en la que el humano no tenga que intervenir en ningún momento en la conducción y siendo esta totalmente segura.

2.2. Vehículos autónomos

Un vehículo autónomo puede definirse en primera instancia como un vehículo capaz de tomar información del entorno mediante sensores y tomar decisiones comparables con las que toma un humano. Sin embargo, la definición de vehículo autónomo es más amplia y depende del grado de autonomía del vehículo, que puede ir desde un simple sistema de

alerta hasta coches en los que el conductor no tiene que intervenir en ningún momento en la conducción. Para lograr la autonomía, estos vehículos pueden contar con numerosos sensores como radares, LIDAR, sonar, GPS, sensores de odometría y sensores inerciales. Además, deben contar con sistemas inteligentes capaces de almacenar gran cantidad de información y tomar decisiones acordes con la situación actual en la que se encuentra el vehículo.

2.2.1. Historia de los vehículos autónomos

A pesar de que el campo de los vehículos autónomos ha tomado mayor notoriedad en los últimos años, la búsqueda de nuevas soluciones en este sector tiene una antigüedad mucho mayor. La empresa General Motors (GM) fue de las primeras empresas en comenzar la investigación y desarrollo de nuevos vehículos que fueran capaz de conducirse sin la presencia de un conductor. De hecho, la primera idea de coche autónomo fue presentada en 1939 en una exposición universal organizada por GM [3]. Ese coche estaba controlado por un circuito implementado en la carretera.

En la década de los 80s, el proyecto EUREKA Prometheus buscaba el desarrollo de un vehículo autónomo. Una furgoneta de la marca Mercedes-Benz (figura 1) fue desarrollada por Ernst Dickmanns y la Universidad de Múnich [4]. La furgoneta estaba dotada de visión por computador, diferentes microprocesadores y enfoques probabilísticos para poder reaccionar al tráfico y a las diferentes situaciones posibles que pudieran darse. La dirección, el acelerador y los frenos eran controlados mediante comandos a tiempo real basados en la evaluación de las imágenes captadas por las cuatro cámaras situadas en el vehículo. Posteriormente, se desarrolló un Mercedes-Benz W140 S-Class basados en los sensores mencionados anteriormente que fue capaz de circular autónomamente de Múnich a Copenhague y vuelta.



Figura 1. Furgoneta Mercedes del proyecto Eureka Prometheus [4]

En la década de los 90, comenzó el desarrollo del “ParkShuttle” (figura 2), el cual era un vehículo eléctrico totalmente autónomo, diseñado para transportar viajeros entre varias estaciones en Rotterdam [5]. Este sistema de transporte se puso en marcha en 2006 con tres paradas durante su recorrido. El funcionamiento del sistema era sencillo, el vehículo seguía una ruta virtual en la que sus posiciones eran comprobadas por una referencia artificial de puntos, los cuales estaban sobre la superficie de la carretera. Además, dispone de sensores inteligentes en la parte frontal del vehículo que permite frenar en caso de obstáculos en el camino. A día de hoy, el vehículo continúa en funcionamiento y ha aumentado el número de paradas.



Figura 2. ParkShuttle [5]

Aunque se siguió trabajando en el desarrollo de vehículos autónomos, a partir de 2010 fue cuando la mayoría de las marcas de coches se centraron en desarrollar sistemas de conducción autónoma. En 2014, Audi consiguió que un RS7 circulara de forma autónoma en el circuito de Hockenheim, alcanzando los 240 km/h.

En 2015, un Audi SQ5 consiguió recorrer de forma autónoma más de 5600 km entre San Francisco y Nueva York [6]. Este coche estaba equipado con cuatro radares de corto alcance, tres cámaras, seis sensores laser, un sistema de localización GPS, sistemas de asistencia a la conducción y un sistema software inteligente para procesar toda la información recogida por los sensores y actuar en consecuencia.

Finalmente, uno de los sistemas pioneros y más prometedores es el Autopilot de Tesla, que es un sistema avanzado de asistencia a la conducción [7]. Este sistema, recoge información gracias a las numerosas cámaras situadas por todo el vehículo, además de un radar delantero y ultrasonidos. El piloto automático permite que el coche acelere, gire y frene automáticamente dentro de los carriles. Además, en las últimas versiones el vehículo tiene una capacidad de conducción prácticamente autónoma siendo capaz de circular a un destino sin la intervención del conductor, aunque siempre con la supervisión activa de este.

Por tanto, los sistemas de conducción autónoma están muy avanzados, aunque queda mucho por investigar y desarrollar para lograr la autonomía total sin la necesidad de supervisión del conductor y así lograr que los vehículos sean completamente autónomos.

2.2.2. Vehículos autónomos en la actualidad

Como se ha visto en el apartado anterior, los sistemas de conducción autónoma están bastante avanzados en la actualidad siendo prácticamente innecesaria la participación del conductor. Estos vehículos están formados por un complejo sistema de sensores y un software inteligente capaz de procesar toda la información recogida por los sensores y actuar de acuerdo con la situación actual. Aunque existen variaciones dependiendo de la marca y el sistema utilizado, la mayoría de los vehículos autónomos actuales cuentan con los sensores mostrados en la figura 3.

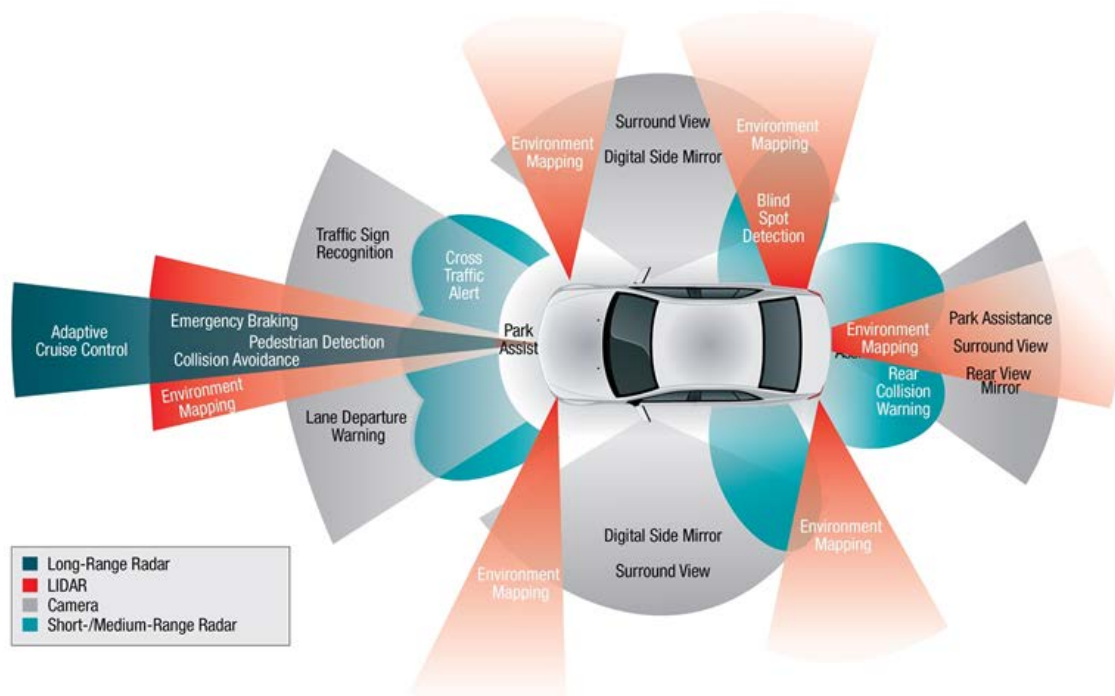


Figura 3. Sensores utilizados por los sistemas de conducción autónoma [8]

Cada uno de estos sensores tiene una funcionalidad distinta que enviará información al software inteligente del vehículo para poder tomar las decisiones. Estos sensores son:

- Radar: es un sistema que permite medir distancias, altitudes, direcciones y velocidades por medio de ondas electromagnéticas. El funcionamiento, similar a los ultrasonidos (con un alcance mayor que estos), se basa en la emisión de una onda de radio que se refleja y se recibe de nuevo en el emisor. La onda reflejada proporciona gran cantidad de información acerca de los obstáculos presentes alrededor del vehículo. Este sensor puede ser de largo alcance o de corto alcance para gestionar mayores o menores distancias. El de corto alcance puede gestionar distancias de entre 10 m y 30 m (con bandas de frecuencia de entre 24 y 29 GHz) y el radar de largo alcance detecta objetos de 100 m (con bandas de frecuencia de 76 y 77 GHz), aproximadamente. A mayor frecuencia, mayor es el alcance y la resolución del sensor, obteniendo datos más fiables. Este sistema permite gestionar diferentes sistemas de control del vehículo como el crucero adaptativo, el sistema de colisión frontal, detección de puntos ciegos, aparcamiento asistido y diferentes alertas de colisión. Este sensor no depende de la iluminación ambiental y rinde adecuadamente en condiciones climatológicas adversas, por lo que es muy utilizado para la detección de obstáculos alrededor del vehículo. Una de las principales desventajas de los radares es que solo captan información en 2D. Sin embargo, en la actualidad se trabaja en radares 3D capaces de captar más información que los radares de dos dimensiones, los cuales tienen dificultades para obtener información acerca de la altura de determinados obstáculos. Por lo general, este tipo de sensores están situados por todo el vehículo, por lo que se tiene información de los 360° alrededor del vehículo.

En la figura 4 se observa un ejemplo de radar de corto alcance con una frecuencia de 26 GHz en la parte trasera del vehículo capaz de detectar obstáculos hasta 30 m.

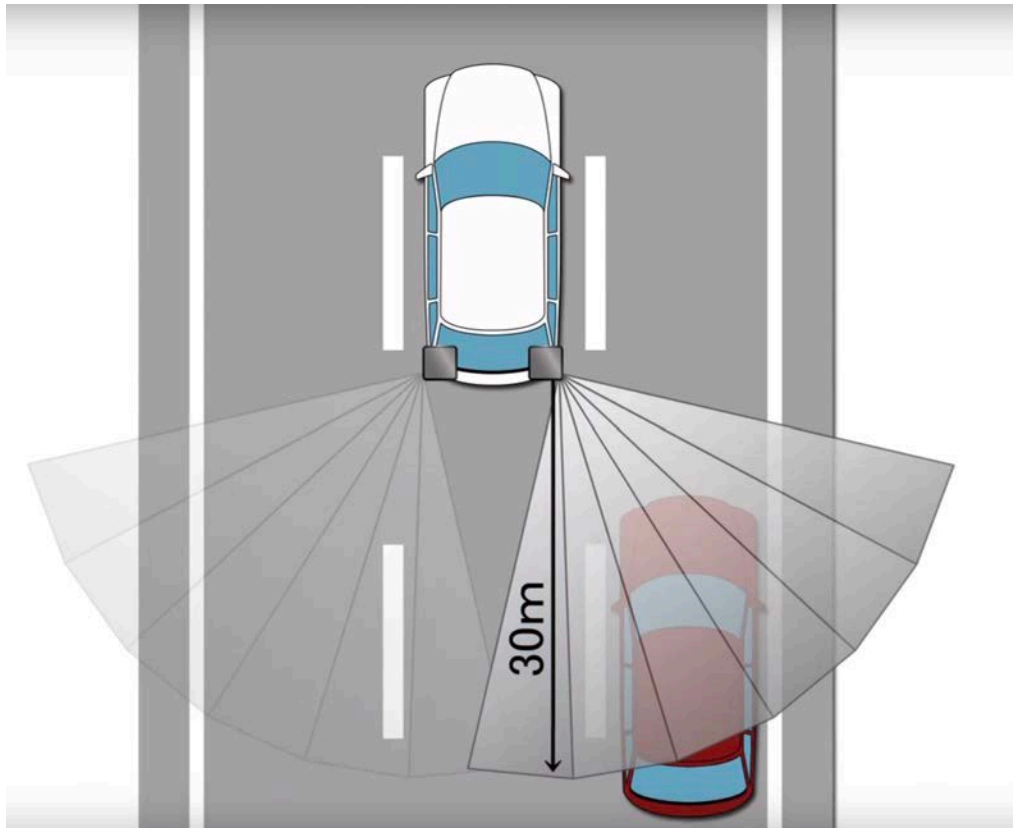


Figura 4. Radar de corto alcance en la parte trasera del vehículo [9]

- **LIDAR (*Light Detection and Ranging*)**: es una tecnología similar al radar, pero en lugar de ondas electromagnéticas el emisor es un láser, cuya señal reflejada da numerosa información. El haz de luz es muy estrecho y la velocidad de escaneo es rápida por lo que mediante nubes de puntos se crean representaciones 3D del entorno del coche. Algunos tipos de LIDAR permiten la evaluación de los 360° del vehículo, lo que supone una gran ventaja respecto otros sensores como el radar. Además, trabaja adecuadamente en condiciones climatológicas adversas y tiene un gran alcance. Los datos recogidos por el LIDAR no requieren un procesamiento tan complejo como ocurre con el caso de las cámaras, que, aunque cada vez son más utilizadas y fiables requieren altos tiempos de procesamiento. La mayoría de los sensores LIDAR utilizados en coches son los que permiten el escaneo de los 360° del coche (figura 5). Estos sensores suelen estar situados en el techo del coche para poder reconocer todo el entorno del vehículo. Este tipo de sensor, aunque es muy fiable, está

siendo sustituido por cámaras que junto con potentes softwares de inteligencia artificial tienen un comportamiento mucho más fiable.

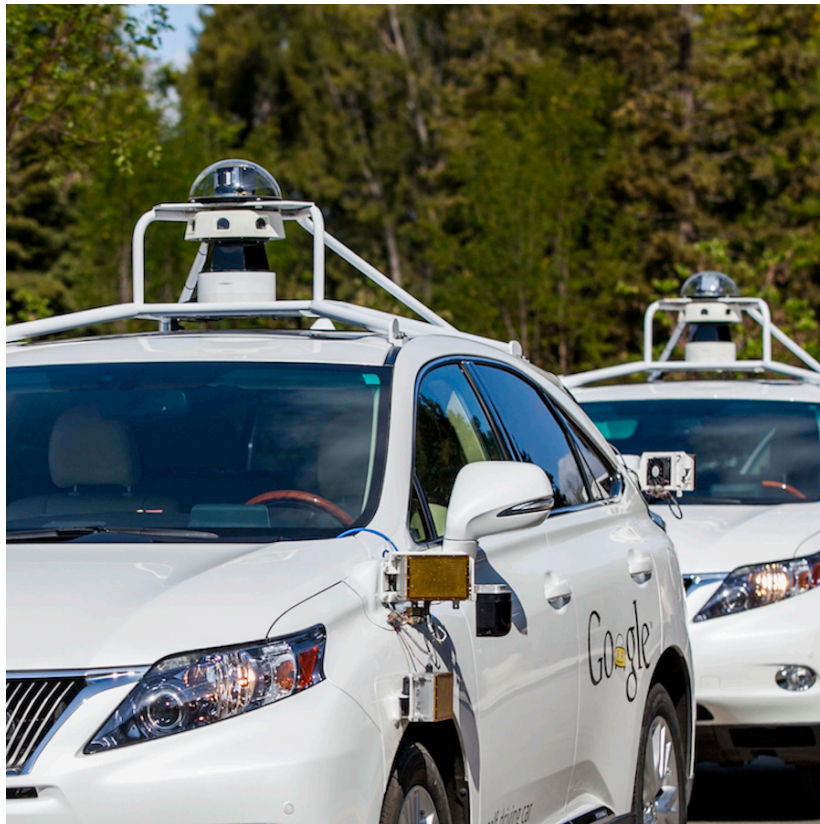


Figura 5. LIDAR situado en el techo del coche [10]

- Cámaras: este sistema básicamente consiste en diferentes cámaras situadas en las distintas partes del coche para tener visión de 360°. Las cámaras necesitan detrás un complejo sistema de procesamiento de inteligencia artificial para poder percibir las diferentes situaciones que se produzcan en el entorno del vehículo. Este sistema es complejo ya que necesita gran cantidad de datos para que los algoritmos de inteligencia artificial puedan trabajar correctamente, sin embargo, es más fiable y barato que otros sistemas como el LIDAR. Los algoritmos basados en inteligencia artificial pueden clasificar los diferentes objetos captados por las cámaras y actuar en consecuencia. Aunque es uno de los principales sensores utilizados para la conducción autónoma, las cámaras llevan años siendo utilizadas para otros objetivos como asistencia de marcha atrás, avisos de cruce de línea, etc. Las cámaras tienen algunas desventajas y

es que al igual que el ojo humano tiene dificultades al trabajar en condiciones de baja visibilidad. Para superar estos problemas existen cámaras infrarrojas que tienen un rendimiento superior en la oscuridad y con capacidades de detección adicionales. Además, existen muchos tipos de cámaras dependiendo del objetivo a detectar, como cámaras trifocales, cámaras de detección de luces de tráfico, de detección de señales, etc. En la figura 6 se muestran diferentes cámaras instaladas en el parabrisas y en la parte lateral de un vehículo de la marca Tesla.

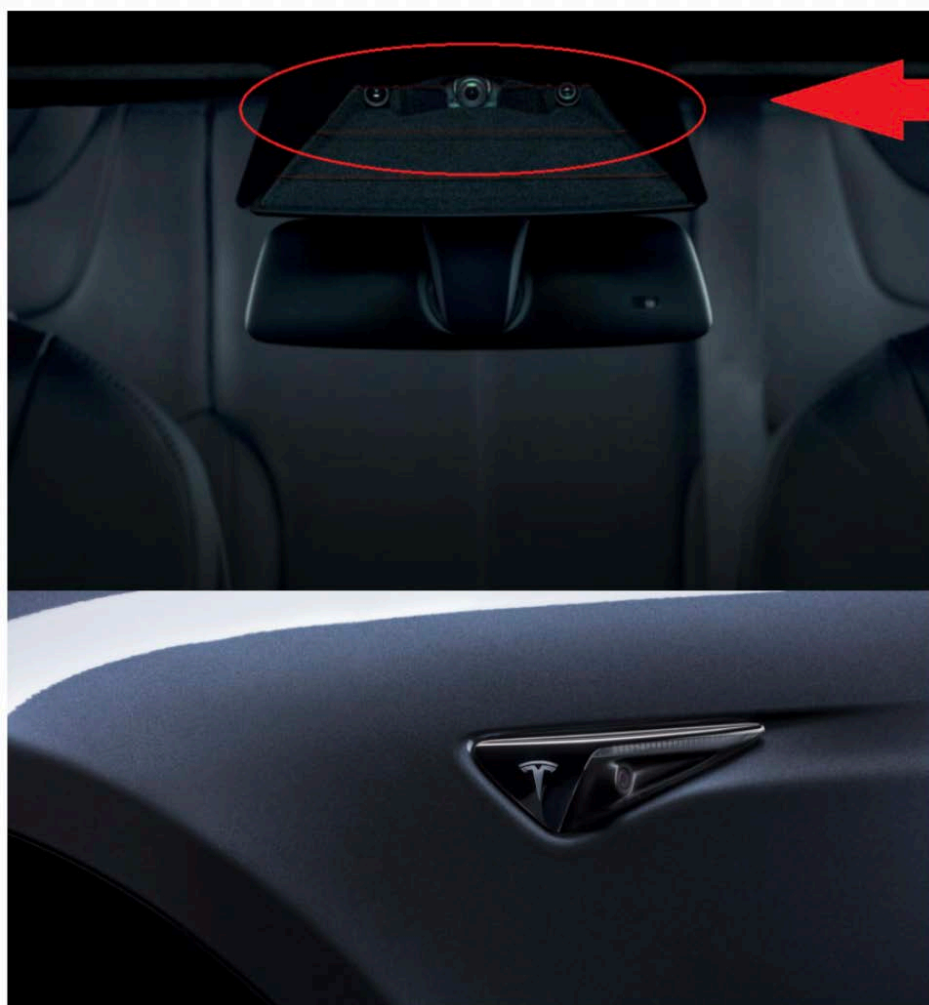


Figura 6. Cámaras frontales y laterales de un vehículo de la marca Tesla [11]

- Ultrasonidos: este sistema es similar al radar, pero utilizando ondas acústicas en lugar de ondas electromagnéticas. Este tipo de sensores, que trabaja de forma similar a como lo hacen los murciélagos y los submarinos, emiten

ondas sonoras que al impactar con los objetos producen ecos captados por los sensores. Midiendo el tiempo entre ondas se puede medir distancias a obstáculos cercanos al vehículo. Los sensores de ultrasonidos trabajan de forma óptima a bajas velocidades y con obstáculos cercanos, siendo muy útiles y ampliamente utilizados para los sensores de aparcamiento disponibles en los coches de la actualidad.

- GNSS (*Global Satellite Navigation System*) o Sistema global de navegación por satélite: el término GNSS se refiere a un grupo de satélites que proporcionan señales desde el espacio capaces de transmitir diferentes datos de posicionamiento o navegación. Gracias a la triangulación de señales se puede obtener la posición del vehículo. Los dos sistemas de navegación utilizados en la actualidad son el GPS (*Global Positioning System*), el GLONASS (*Global'naya Navigatsionnaya Sputnikovaya Sistema*) y el europeo Galileo. Gracias a estos sistemas se puede conocer la posición exacta del coche y es capaz de dirigirlo al destino indicado. Uno de los principales inconvenientes de estos sistemas son la cobertura, ya que una conducción totalmente autónoma requiere de una información muy precisa acerca de la ubicación exacta del coche para garantizar la máxima seguridad, especialmente en zonas de baja cobertura.

2.2.3. Niveles de autonomía

Para que un vehículo sea autónomo es necesario, además de un conjunto de sensores adecuado, un sistema de procesamiento que decida que acciones realizar a partir de la información obtenida y un sistema que controle la ejecución de los movimientos del vehículo. Este software está basado en inteligencia artificial con algoritmos de *machine learning* los cuales se pueden entrenar a través de imágenes con todas las posibles situaciones en las que se debe actuar de alguna forma para evitar los posibles peligros durante la conducción. A medida que este software se entrena, el índice de fallos será cada vez menor y por tanto la conducción más segura. Aquí radica uno de los principales problemas en el desarrollo de vehículos autónomos, y es la gran cantidad de datos necesarios para obtener un índice de fallos mínimo que garantice la conducción segura.

Además, para la toma de decisiones se necesita otro algoritmo que proporcione una actuación a partir de las situaciones procesadas a través de los datos recogidos por los sensores. Otra técnica que se está implantando en la actualidad son los modelos probabilísticos, mediante la cual se predice la posición del vehículo y de los demás objetos identificados mediante distribuciones de probabilidad para tomar las decisiones adecuadas en cada momento de la conducción.

Mediante la coordinación de estos elementos se pueden alcanzar diversos niveles de autonomía, por lo que varios organismos se han encargado de realizar diferentes clasificaciones para indicar en que punto de autonomía se encuentra un vehículo. Una de las primeras organizaciones en realizar una clasificación fue la NHTSA (*National Highway Traffic Safety Administration* o *Administración Nacional de Seguridad del Tráfico en las Carreteras*, en español), que es una agencia del gobierno de Estados Unidos encargada de gestionar las carreteras estadounidenses, centrándose en la seguridad de las carreteras y de los coches (similar a la Dirección General de Tráfico (*DGT*) en España). Esta clasificación está formada por cinco niveles basadas en la capacidad de un vehículo de conducirse por sí mismo [12]. Los niveles definidos por la NHTSA son:

- **Nivel 0 – Sin automatización:** el conductor tiene el control completo de las funciones principales del vehículo (frenos, volante, acelerador, etc). Únicamente existen diferentes alertas para avisar al conductor de diferentes situaciones como alerta de colisión frontal, detector de ángulo ciego, aviso de cruce de línea, etc.
- **Nivel 1 – Automatización en funciones específicas:** la automatización en este nivel implica una o mas funciones de control específicas. El conductor tiene el control general del vehículo, sin embargo, puede ceder la autoridad al vehículo de algunas funciones primarias como el control de crucero adaptativo, frenado de emergencia ante colisión, etc. Por tanto, no existe una combinación entre los sistemas del vehículo en los que el conductor pueda soltar por ejemplo los pedales y el volante al mismo tiempo. En este nivel, el conductor debe tener el control de la mayoría de las funciones del vehículo en todo momento.
- **Nivel 2 – Automatización en funciones combinadas:** en este nivel la automatización implica al menos dos funciones primarias del control de

vehículo. Aunque el conductor sigue teniendo el deber de monitorizar y garantizar la seguridad, puede ceder la autoridad al vehículo de diversas funciones en situaciones de conducción limitada. Un ejemplo de este sistema es la combinación del asistente de mantenimiento de carril y el control de crucero adaptativo.

- **Nivel 3 – Automatización de autoconducción limitada:** en este nivel de automatización el conductor puede ceder el control total de las funciones del vehículo en situaciones críticas de seguridad en ciertas condiciones de tráfico o ambientales. En este caso el conductor no tiene que estar constantemente monitorizando la carretera durante la conducción, a excepción de casos excepcionales como áreas de conducción, situaciones de tráfico especiales en los que el conductor debe disponer de nuevo del control del vehículo.
- **Nivel 4 – Automatización de autoconducción completa:** el vehículo está diseñado para tener un rendimiento completo en situaciones de seguridad crítica y monitoriza constantemente las condiciones de la carretera. En este caso no es necesaria la intervención del conductor en ningún momento, siendo posible la conducción sin una persona a los mandos del vehículo.

La Sociedad de Ingenieros de Automoción (SAE), una de las principales organizaciones en el sector de la automoción, se encargó también de realizar una clasificación [13], que dividió en seis los diferentes niveles de conducción autónoma. Estos niveles son:

- **Nivel 0 – Sin automatización:** similar al nivel 0 en la clasificación de la NHTSA. El humano realiza todas las tareas de la conducción dinámica disponiendo únicamente de diversas alertas que permitan avisar de determinadas situaciones de peligro.
- **Nivel 1 – Asistencia al conductor:** corresponde al nivel 1 de la NHTSA. El vehículo cuenta con algún sistema de asistencia al conductor para el movimiento longitudinal o lateral del vehículo. Es decir, el vehículo puede tomar el control del acelerador o del volante, pero no de ambos a la vez. La detección obstáculos y situaciones de peligro recaen en el conductor, aunque el sistema pueda realizar esta tarea de forma parcial y con elementos específicos. Ejemplos de esto son el control de crucero adaptativo o el sistema

de aparcamiento automático en el que el vehículo controla la dirección y el conductor la aceleración del vehículo.

- **Nivel 2 – Automatización parcial:** al igual que el nivel 2 de la clasificación de la NHTSA el vehículo puede tomar el control de uno o más sistemas de conducción dinámica como la dirección y la aceleración/desaceleración. El vehículo realiza las tareas de conducción, pero no dispone de un sistema de detección y respuesta ante objetos, por lo que esta tarea recae sobre el conductor.
- **Nivel 3 – Automatización condicionada:** en este caso el sistema sigue teniendo la capacidad de conducción (aceleración y dirección) además de poder monitorizar el ambiente y actuar ante determinadas situaciones. El sistema puede solicitar la actuación del conductor ante determinadas situaciones o en caso de fallo en las que el vehículo no pueda actuar de manera segura. El sistema debe informar al usuario con el tiempo suficiente para actuar de manera que se garantice la seguridad. El sistema es limitado en determinadas condiciones.
- **Nivel 4 – Automatización alta:** el vehículo dispone de todas las funciones de conducción dinámica y de monitorización del ambiente. Además, en caso de fallo, a diferencia del nivel anterior, el sistema de automatización dispone de un sistema secundario para sustituir al principal en caso de fallo. A pesar de ello, el sistema sigue siendo limitado en ciertas situaciones y no es necesaria la intervención del conductor.
- **Nivel 5 – Automatización completa:** el vehículo cuenta con todas las funciones de conducción dinámica y de monitorización. En este caso el sistema no está limitado a ninguna situación y puede actuar en cualquier momento y ante cualquier circunstancia inesperada. En este caso se sustituye completa al conductor y con una conducción mucho más fiable. Tanto en este nivel como en el anterior, no es necesario un volante o pedales ya que todo se controla mediante un computador y no se necesita la intervención humana.

2.3. Simulación

La simulación en computadores en el área de la automoción ha estado presente desde hace muchos años, destacando su uso en el sector de la seguridad del automóvil. En este

campo han realizado constantemente pruebas para estudiar el comportamiento del vehículo y sus componentes ante impactos bruscos, así como para determinar el ciclo de vida o la fiabilidad de estos antes de realizar las pruebas reales. Con la llegada de los simuladores de vehículos autónomos, se podría realizar una metodología similar, comenzando por la simulación en computador para trasladarlo posteriormente a pruebas reales. Este método, ahorraría numerosos costes en posibles fallos que pudieran aparecer durante los ensayos reales, los cuales podrían haberse subsanado si anteriormente se hubieran descubierto estos fallos en las pruebas simuladas. Por tanto, la simulación de vehículos autónomos puede resultar muy útil para realizar pruebas previas a los ensayos reales con vehículos.

Aunque se han desarrollado numerosos simuladores de conducción autónoma en los últimos años, uno de los más destacados es el simulador CARLA, de código abierto y desarrollado por investigadores de la Universidad Autónoma de Barcelona y el laboratorio Intel Computing Visual. En el siguiente apartado se hablará en profundidad de este simulador, el cual se utilizará para la realización de este trabajo.

2.4. Simulador CARLA

El entorno de simulación de código abierto CARLA (siglas del inglés de: *Car Learning to Act*, o coche que aprende a actuar) fue desarrollado por investigadores del Centro de Visión por Computador de la Universidad Autónoma de Barcelona en colaboración con Intel Computing Visual Lab. El simulador permite realizar pruebas de conducción autónoma en diferentes entornos y condiciones, dotando a los vehículos de diferentes sensores cuya información es procesada mediante inteligencia artificial. Gracias a esto, los vehículos pueden circular por los diferentes mapas disponibles en el simulador (o creados mediante otros programas) evitando las colisiones con los diferentes elementos del entorno y con otros vehículos, por los carriles y obedeciendo las diferentes señales de circulación dispuestas en el mapa. Este simulador destaca respecto a otros simuladores por varios aspectos: es de código abierto y uso gratuito, permite el control de las condiciones climatológicas y del comportamiento de vehículos, semáforos, ciclistas y peatones, entre otras cosas. Gracias a estos aspectos, se logra una simulación mucho más realista, con situaciones similares a las que se pueden encontrar en el mundo real.

Algunas de las características más destacables de CARLA son:

- Arquitectura multi-cliente: varios clientes pueden controlar diferentes actores en la escena.
- Flexibilidad de la API: la API permite a los usuarios el control de numerosos aspectos de la simulación como la variación de las condiciones climatológicas, modificación de los sensores, tráfico, fase del día, etc.
- Amplia variedad de sensores: CARLA dispone de gran cantidad de sensores que pueden ser colocados en el lugar deseado (cámaras RGB, cámaras de profundidad, LIDAR, sensor GNSS, etc.).
- Creación y modificación de mapas: CARLA permite la modificación de los propios mapas ya creados, además de la importación de mapas creados en otros softwares en formatos FBX y OpenDRIVE.
- Simulación en situaciones de tráfico específicas: el simulador permite estudiar el comportamiento de los vehículos en situaciones de tráfico concretas como puede ser un semáforo o el cruce inesperado de un peatón.
- Integración con ROS: a través de ROS-bridge CARLA puede integrarse con ROS.

2.4.1. Estructura

El simulador CARLA está compuesto por dos módulos distintos (figura 7), el módulo simulador *CARLA Simulator* y el módulo de scripts de usuario *CARLA Python API*. Por tanto, CARLA está diseñado como un sistema cliente-servidor, donde el servidor se encarga de la ejecución de la simulación y la renderización de la escena, mientras que el cliente API se encarga de la interacción entre el servidor y los agentes autónomos. El cliente se encarga de enviar comandos y meta-comandos al servidor y de recibir las lecturas de los diferentes sensores.

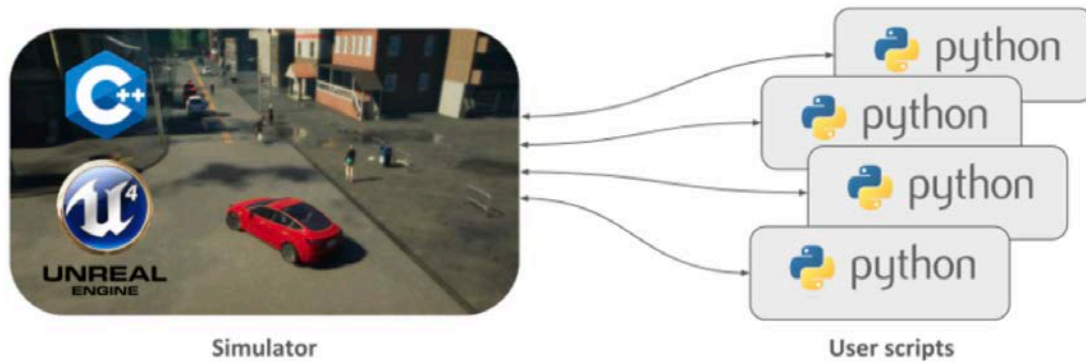


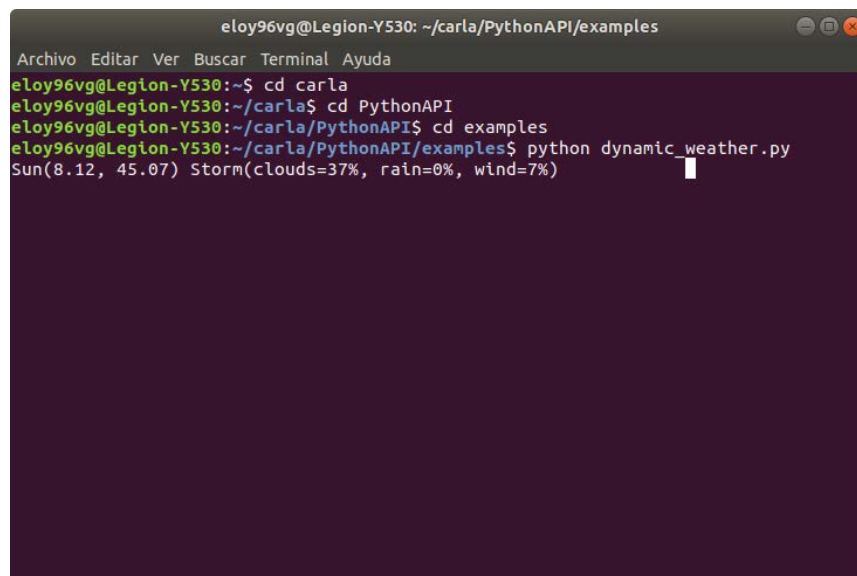
Figura 7. Módulos de CARLA [14]

El simulador realiza la mayoría del trabajo, controlando las lógicas, las físicas y la representación gráfica de los actores y sensores. El simulador está desarrollado en el lenguaje de programación C++ y se accede a él mediante el motor de juego Unreal Engine 4 creado por la compañía Epic Games. La interfaz de Unreal Engine 4 (*Unreal Editor*) permite el desarrollo, visualización y modificación de los diferentes mapas y elementos por los que está compuesto el simulador (figura 8). Además, permite la configuración de la simulación y diferentes aspectos de rendimiento.



Figura 8. Unreal Editor

Por otro lado, el módulo *CARLA Python API* permite la importación de ficheros escritos en lenguaje Python proporcionando una interfaz para controlar el simulador. Con este módulo se pueden controlar numerosos aspectos de la simulación como las condiciones meteorológicas (figura 9), la colocación y visualización de los sensores, la lectura de los datos proporcionados por éstos y la adición de actores a la simulación (vehículos y peatones), entre otros.



```
eloy96vg@Legion-Y530: ~/carla/PythonAPI/examples
Archivo Editar Ver Buscar Terminal Ayuda
eloy96vg@Legion-Y530:~$ cd carla
eloy96vg@Legion-Y530:~/carla$ cd PythonAPI
eloy96vg@Legion-Y530:~/carla/PythonAPI$ cd examples
eloy96vg@Legion-Y530:~/carla/PythonAPI/examples$ python dynamic_weather.py
Sun(8.12, 45.07) Storm(clouds=37%, rain=0%, wind=7%)
```

Figura 9. Ejecución del script de condiciones meteorológicas cambiantes

Además, el módulo permite la conducción de los vehículos mediante el conjunto de módulos de Python denominados *Pygame* (figura 10). Con estos módulos, además de la conducción del vehículo, se pueden observar diferentes aspectos de la conducción (velocidad, giro del volante, frenos), de la simulación (tiempo de simulación, FPS) y diferentes datos proporcionados por los sensores (localización, posición GNSS, altura). En esta ventana además se puede cambiar la visión normal por la visión de los diferentes sensores por los que está dotado el vehículo (cámara de profundidad, LIDAR, etc.). Estos aspectos se estudiarán en profundidad en las pruebas realizadas en el entorno de la ampliación universitaria del Campus de Teatinos.

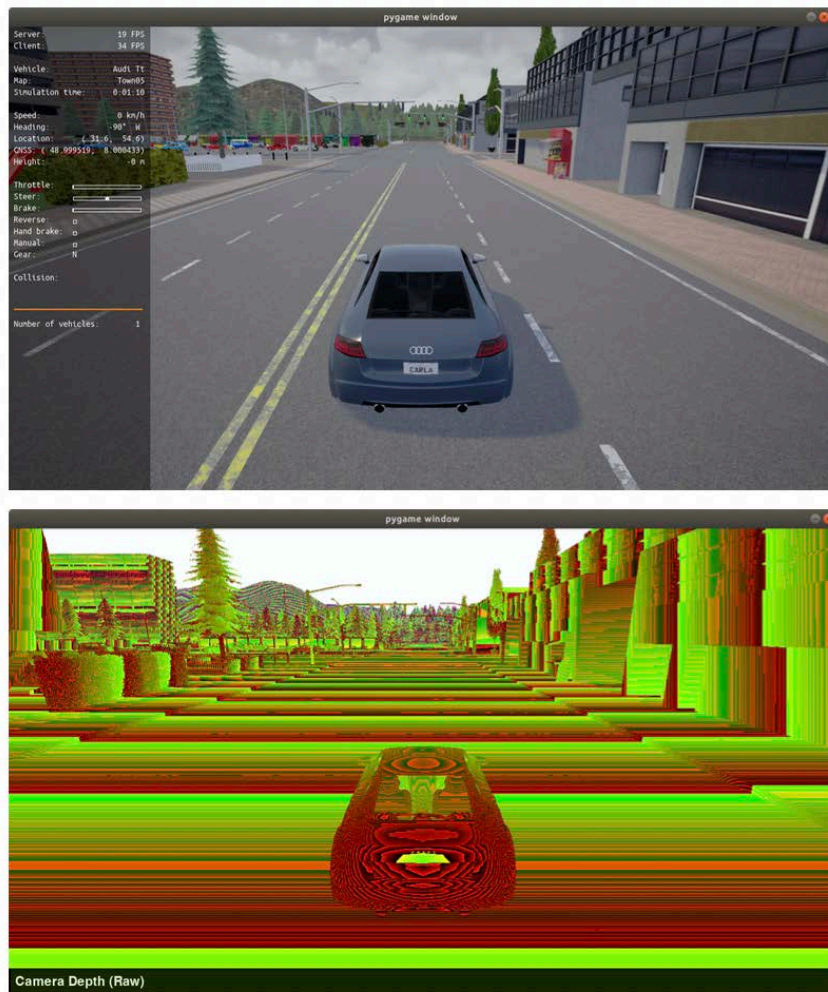


Figura 10. Ventanas *Pygame* con visión normal (arriba) y con cámara de profundidad (abajo)

2.4.2. Entorno

El entorno de simulación en CARLA está formado por modelos 3D de objetos o mallas estáticas como pueden ser edificios, señales de tráfico e infraestructuras, además de objetos dinámicos como coches y peatones. Estos objetos están diseñados con alto nivel de detalle, consiguiendo un realismo alto, y con modelos y texturas con un peso computacional bajo para lograr una simulación fluida en computadores no demasiado potentes. La librería de CARLA es muy amplia y dispone de alrededor de 40 edificios diferentes, 24 modelos de vehículos y 50 tipos de peatones animados. Todos los objetos son modificables tanto en escala, como en posición en el mapa. Gracias a la librería disponible, el simulador dispone de siete ciudades diferentes completamente modificables.

Durante la simulación, se pueden añadir a la escena tanto vehículos como peatones no controlables, los cuales se han diseñado para lograr un gran realismo en sus movimientos, desde la cinemática de sus movimientos hasta el propio comportamiento de estos. Gracias a ello, peatones y vehículos pueden detectarse mutuamente y entre ellos mismos circulando por las zonas indicadas para cada uno de ellos.

Para lograr un mundo más realista, los vehículos y peatones añadidos a la simulación son creados de forma aleatoria combinando diferentes colores en los distintos modelos de vehículos y diferentes tipos de ropa en el caso de los peatones. Parte del realismo del mundo se logra también gracias a la variedad en las condiciones climatológicas como nublado, lluvia intensa y soleado, entre otras. Además, el simulador soporta dos momentos distintos del día, mediodía y atardecer. En total, hay 18 combinaciones de iluminación y condiciones atmosféricas.

2.4.3. Cámaras y sensores disponibles

Las cámaras y sensores son un tipo de actor que permiten la medida y transmisión de datos. Gracias a los datos recogidos por los sensores y a su procesamiento, los vehículos pueden circular de forma autónoma. Para el control de estos sensores (posición, tiempo de capturas, etc.) se dispone de diferentes métodos y funciones englobados dentro del módulo *CARLA Python API*. En código 1 se muestra la configuración de una cámara de profundidad colocada en un vehículo.

```
# Se selecciona el tipo de sensor.
blueprint = world.get_blueprint_library().find('sensor.camera.depth')
# Se modifica la anchura (x) y altura (y) de la imagen en píxeles,
# además del campo de visión horizontal en grados (fov).
blueprint.set_attribute('image_size_x', '1920')
blueprint.set_attribute('image_size_y', '1080')
blueprint.set_attribute('fov', '100')
# Se configura el tiempo entre capturas del sensor (en segundos)
blueprint.set_attribute('sensor_tick', '1.0')
# Se configura la posición del sensor relativa al vehículo.
transform = carla.Transform(carla.Location(x=1, z=1.8))
# Se genera el sensor y se conecta al vehículo.
sensor = world.spawn_actor(blueprint, transform,
attach_to=my_vehicle)
# Nos suscribimos al sensor mediante una función de devolución de
# llamada, la cual es llamada cada vez que el sensor genera una nueva
# imagen
sensor.listen(lambda data: do_something(data))
```

Código 1. Configuración de una cámara de profundidad

El simulador proporciona numerosos sensores para obtener gran cantidad de información acerca del entorno del vehículo. La mayoría de estos sensores son los utilizados en los vehículos autónomos existentes en la actualidad (LIDAR, cámaras RGB, GNSS, etc.). Los sensores disponibles son:

- Cámara RGB: este sensor actúa como una cámara normal que captura imágenes reales de la escena. Este cámara es bastante útil ya que da la visión más realista de todas, aunque requiere un procesamiento mayor y algoritmos de inteligencia artificial más complejos. Este tipo de cámaras son las implantadas en los últimos modelos de vehículos autónomos. En la figura 11 se observa la visión de la cámara RGB colocada en la parte superior del vehículo.



Figura 11. Cámara RGB colocada en la parte superior del vehículo

- Cámara de profundidad: la cámara de profundidad proporciona una visión del entorno en la que la distancia de cada píxel a la cámara está codificada (se conoce como buffer de profundidad o z-buffer). Esta cámara resulta interesante para determinar la distancia a objetos y otros vehículos. En la figura 12 se muestran las imágenes obtenidas por la cámara. Concretamente se muestra la imagen original obtenida y una imagen convertida a escala de grises, la cual muestra de forma más clara los diferentes obstáculos del entorno.

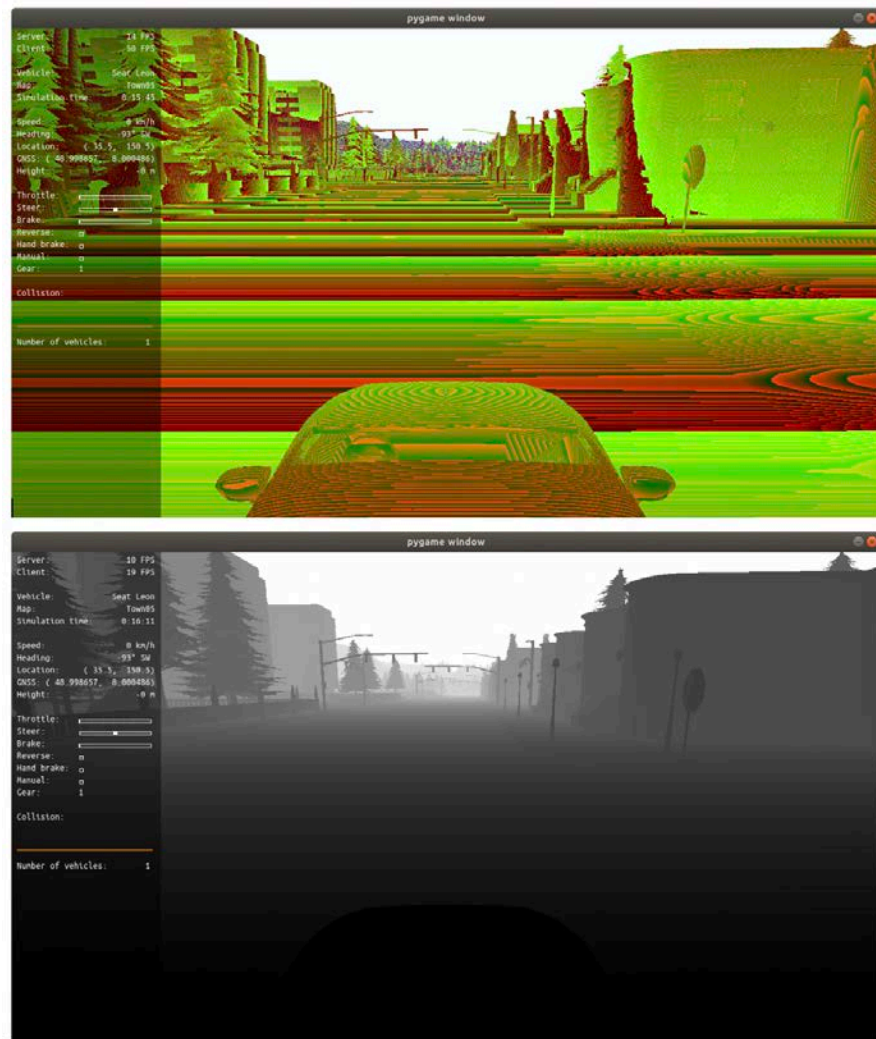


Figura 12. Cámara de profundidad colocada en la parte superior del vehículo. Original (arriba) y convertida a escala de grises (abajo)

- Cámara de segmentación semántica: esta cámara realiza una clasificación de cada objeto del entorno visualizado mostrándolo de un color diferente de acuerdo con la clase a la que pertenezcan. Esta cámara puede resultar muy útil para evitar colisionar con los diferentes objetos representados de un determinado color. Esta cámara necesita complejos algoritmos que reconozcan y clasifiquen cada uno de los objetos presentes en la escena. En la figura 13 se muestra la imagen de la cámara con los objetos clasificados por colores. Los coches están representados en azul, las aceras en color magenta, las líneas de la carretera en verde y las carreteras en violeta. Con esta clasificación se podría determinar los lugares por los que puede o no circular el vehículo y los obstáculos que debe evitar.

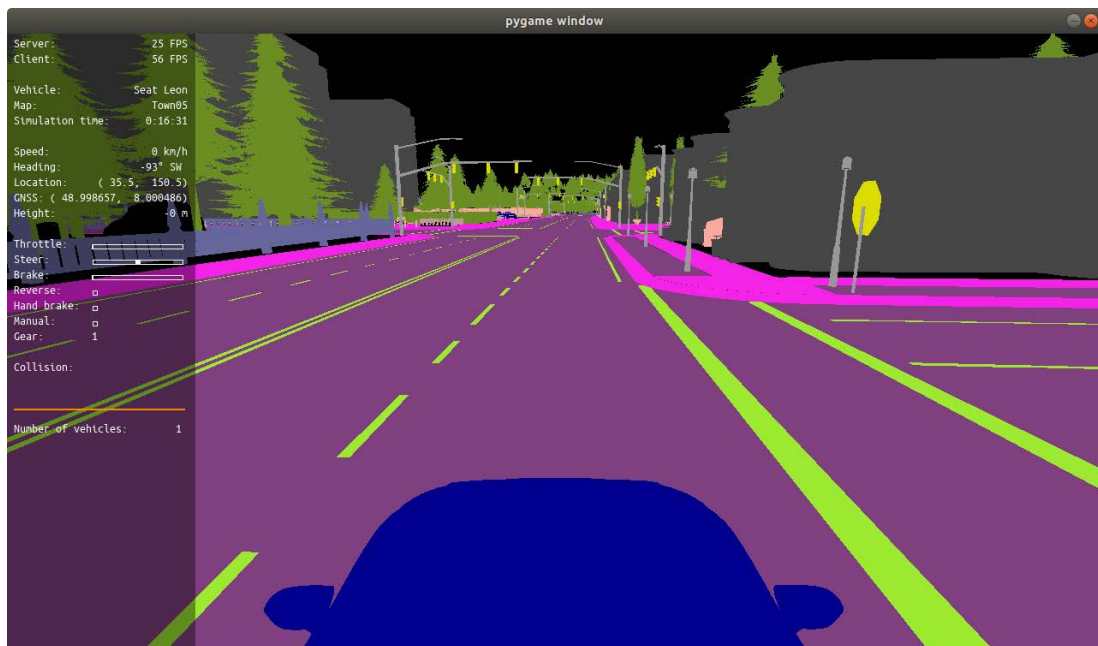


Figura 13. Visualización de la cámara de segmentación semántica colocada en la parte superior del vehículo.

- **LIDAR:** este sensor simula un LIDAR rotatorio. La nube de puntos se calcula agregando un láser por cada canal en el campo de visión vertical, simulando la rotación mediante el cálculo del ángulo horizontal en el que el LIDAR gira. En la figura 14 se muestran los puntos calculados tras la obtención de datos del LIDAR. Este sensor es muy utilizado en vehículos de conducción autónoma, ya que es bastante fiable a la hora de detectar la distancia a los diferentes obstáculos del entorno, además de tener información de los 360° del vehículo en el caso de que se disponga de un LIDAR rotatorio.

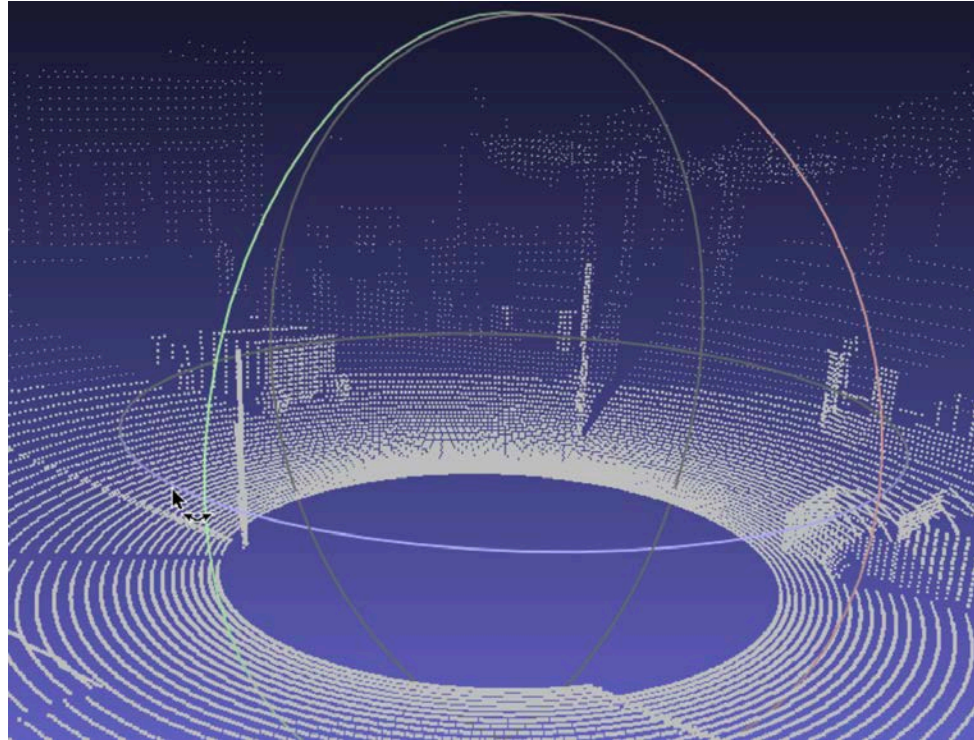


Figura 14. Puntos obtenidos mediante el LIDAR [15]

- Sensor de colisión: este sensor colocado en el vehículo detecta cuando este entra en contacto con algún otro objeto, registrando un evento. El sensor de colisión no puede ser configurado, a diferencia de lo que ocurría con las cámaras explicadas anteriormente. En la figura 15 se observa como el sensor detecta dos colisiones indicando la intensidad de estas (círculos rojos) y el objeto de la escena con el que se ha producido la colisión. Este sensor utiliza datos de otros sensores como el sensor inercial IMU del que dispone el vehículo. Estos sensores detectan cambios bruscos de la aceleración determinando de forma muy fiable si se ha producido una colisión (de la misma forma que lo hace el sensor de colisión del simulador).



Figura 15. Indicadores del sensor de colisión

- Sensor de invasión de línea: este sensor registra un evento cuando el vehículo cruza una línea de la carretera. Este sensor utiliza los datos de la carretera presentes en el simulador para saber cuándo el vehículo ha invadido otra línea. La información de la carretera está presente en los archivos OpenDRIVE proporcionados por el mapa. Los vehículos autónomos reales utilizan sensores de infrarrojos o cámaras para detectar los cambios involuntarios de carril, aunque el sistema de aviso se asemeja al utilizado en el simulador. En la figura 16 se observa el aviso del sensor al traspasar una línea continua.



Figura 16. Sensor de invasión de línea

- Sensor GNSS: este sensor indica la posición actual (en coordenadas geográficas) en la que se encuentra el vehículo. La posición GNSS se calcula internamente agregando una posición métrica a una ubicación de referencia definida en el mapa OpenDRIVE. Es decir, a partir de la localización en el mapa se determina una posición GNSS con una latitud y una longitud determinada, de manera similar a la posición GNSS calculada en los vehículos reales. Gracias a este sensor se sabe en todo momento en que punto del mapa está circulando el vehículo. En la figura 17 se observan las coordenadas calculadas por el sensor GNSS.



Figura 17. Indicador de posición GNSS

- Sensor IMU: este sensor, también denominado unidad de medición inercial, da información sobre la velocidad, orientación y fuerzas gravitacionales del vehículo. Esto se logra gracias a la combinación de un acelerómetro, un giroscopio y una brújula. Se muestra en la figura 18 cómo se obtiene la información de los tres ejes dados por el acelerómetro y el giroscopio, además de la orientación proporcionada por la brújula. En este tipo de sensor es importante conocer los tres ejes de medida del acelerómetro y giroscopio para poder entender los datos proporcionados. En este caso el eje X corresponde con el eje longitudinal del vehículo, el eje Y con el eje transversal y el eje Z con el eje vertical respecto del coche.



Figura 18. Sensor IMU

- Sensor de obstáculos: este sensor indica si hay obstáculos delante. El funcionamiento es similar al de los ultrasonidos en los coches de la actualidad, los cuales mediante la emisión y recepción de ondas ultrasónicas determinan la presencia de obstáculos cercanos al vehículo. En el simulador, este sensor funciona como un sensor de ultrasonidos de corto alcance ya que avisa únicamente cuando hay un obstáculo cerca.
- Sensor de obstáculos: este sensor emite señales electromagnéticas y calculando el tiempo de emisión/recepción de la señal es capaz de obtener información sobre la distancia a la que encuentran los objetos respecto al vehículo.

Capítulo 3

Modelado del entorno de pruebas

3.1. Introducción

El objetivo del presente trabajo es la creación de un entorno de simulación para realizar pruebas de conducción autónoma en el entorno virtual. Aunque CARLA proporciona diferentes mapas donde realizar las simulaciones, es interesante estudiar el comportamiento de los vehículos autónomos en un entorno real y conocido como es la ampliación universitaria del Campus de Teatinos. Para ello, habrá que modelar desde cero el entorno universitario, creando las mallas estáticas (carreteras, aceras, etc.) y las diferentes condiciones de circulación (sentido de la marcha, cruces, etc.). Este trabajo inicial se realizará mediante el programa RoadRunner de VectorZero. En segundo lugar, se importará el entorno creado a CARLA y se completará el mapa (adición de edificios, coches, etc.), además de corregir los posibles fallos ocasionados durante la importación.

La ampliación universitaria del Campus de Teatinos fue creada por la creciente demanda de estudiantes y la necesidad de nuevas facultades capaces de albergar esa gran cantidad de estudiantes. Para ello, se construyó una zona anexa al campus universitario de Teatinos para albergar dos facultades (Facultad de Ciencias de la Salud y Facultad de Estudios Sociales y del Trabajo), la Escuela de Ingenierías y varios centros de investigación como el Ada Byron. La ampliación universitaria se encuentra en la zona oeste de Málaga como se observa en la figura 19, situada al oeste del Campus de Teatinos original.

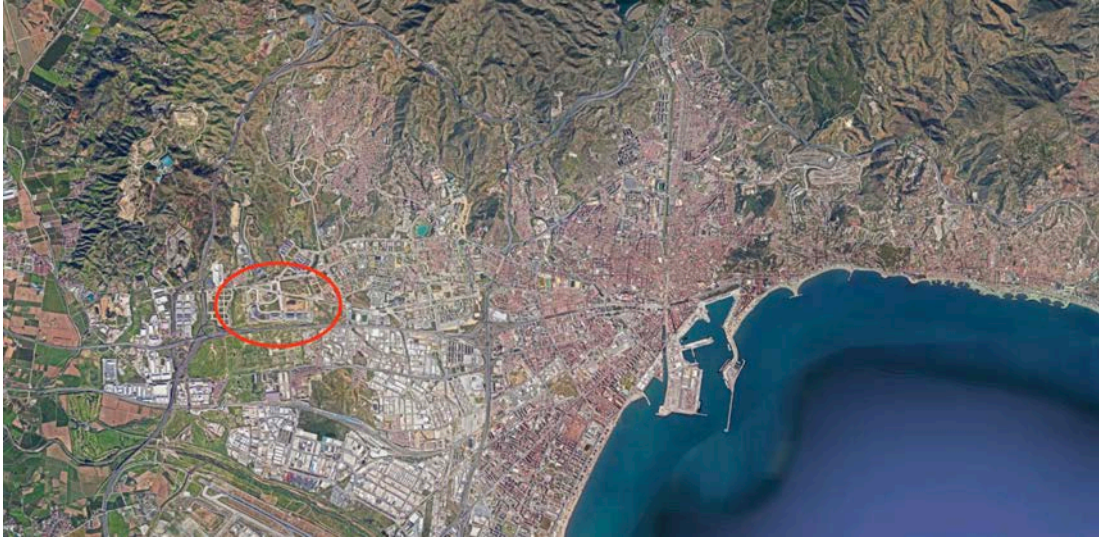


Figura 19. Posición de la ampliación universitaria del Campus de Teatinos en Málaga

Concretamente, el entorno modelado se muestra en la figura 20, que comprende la propia ampliación y los alrededores que proporcionan el acceso a esta.



Figura 20. Ampliación universitaria del Campus de Teatinos y alrededores

3.2. Modelado en RoadRunner

RoadRunner es un software de pago creado por la empresa VectorZero, cuya función principal es la creación de mapas virtuales que puedan ser exportados en distintos formatos y ser utilizados en el desarrollo de videojuegos o simuladores. Estos mapas pueden ser exportados para Unity, Unreal, en formato OpenDRIVE y FBX. Serán estos dos últimos formatos los utilizados para importar el entorno universitario en el simulador CARLA.

Para crear un entorno verosímil de dimensiones similares a las reales, era de especial necesidad conocer las longitudes y altitudes de las diferentes carreteras por las que está formada el entorno de la ampliación universitaria de Teatinos. Para lograr un resultado rápido y fiable se ha utilizado la herramienta Google Earth que proporciona numerosa información como se observa en la figura 21. Gracias a esta herramienta se puede conocer información importante como las coordenadas y la altitud.

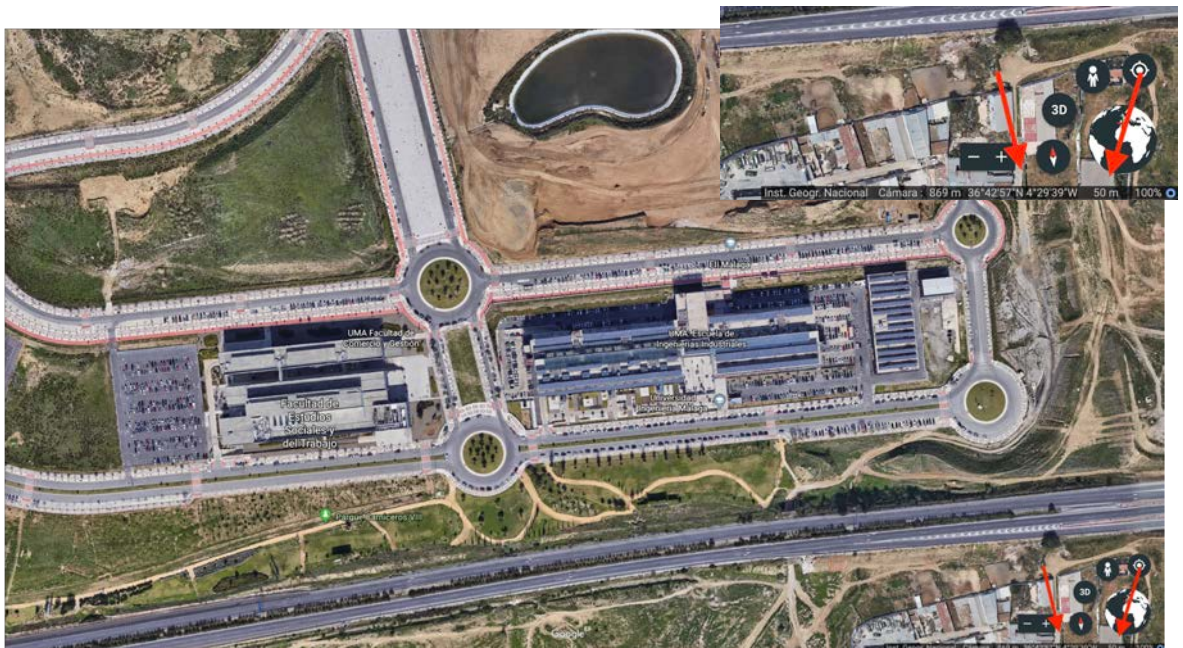


Figura 21. Coordenadas y altitud en Google Earth

Además, este programa dispone de una herramienta que permite mirar la distancia entre dos puntos (figura 22), lo que resulta muy útil para conocer la longitud de las carreteras y crearlas en RoadRunner con la mayor veracidad posible. Conociendo las

distancias y altitudes de las carreteras que conforman la ampliación universitaria se puede comenzar a modelar el mapa con la multitud de herramientas de las que dispone RoadRunner.



Figura 22. Herramienta para medir distancias en Google Earth

3.2.1. Herramientas de RoadRunner

RoadRunner dispone de multitud de herramientas para la creación de mapas muy realistas, por lo que en este apartado se mostrarán algunas de las más utilizadas para el modelado de la ampliación universitaria.

- Creación de carreteras: con esta herramienta se construyen las carreteras, seleccionado inicialmente el tipo de carretera (como se observa en la zona inferior derecha de la figura 23), así como la longitud y curvatura de esta. También permite cambiar los materiales como el color del asfalto o las aceras (recuadro a la derecha de la figura 23). Hay numerosos tipos de carreteras, como autopistas, carreteras convencionales, carreteras residenciales, etc.

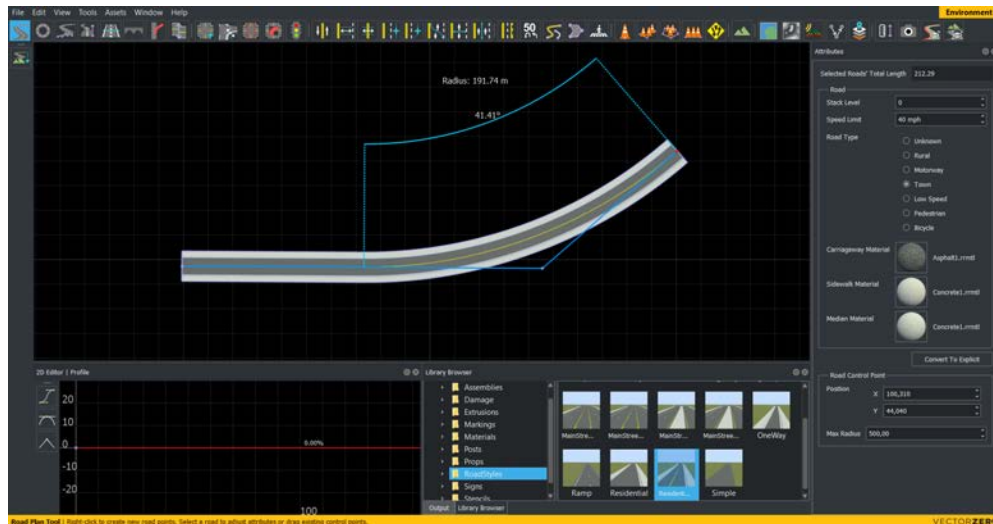


Figura 23. Herramienta para modelar carreteras en RoadRunner

- Creación de rotondas: esta herramienta funciona de forma similar a la anterior, seleccionando el tipo de carretera y el radio de la rotonda (figura 24). Las rotondas se unen automáticamente con el resto de carreteras al crearlas junto a estas últimas.

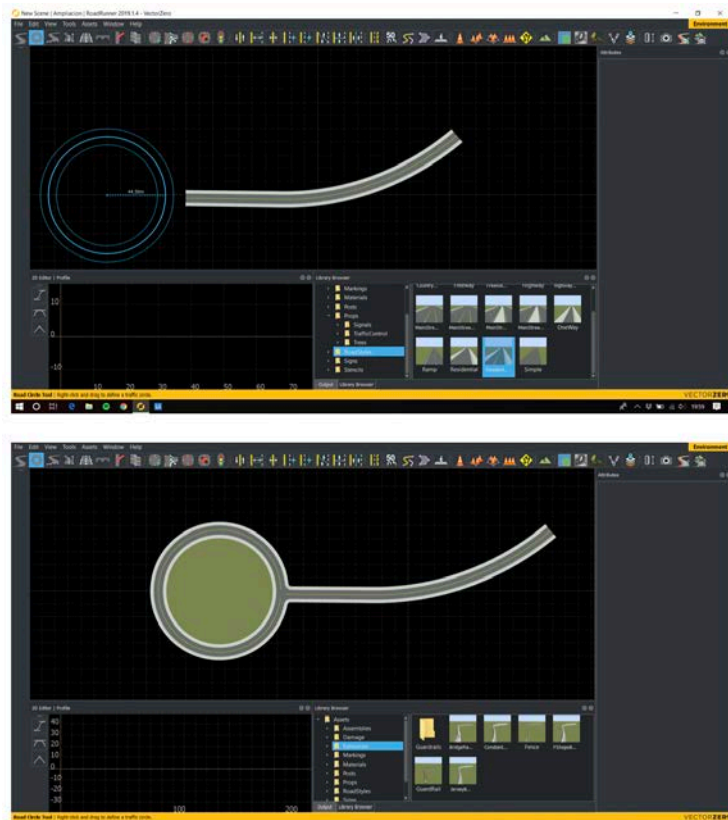


Figura 24. Herramienta para el modelado de rotondas

- Configuración de la altitud: gracias a esta herramienta se selecciona la altitud de cualquier punto de la carretera. Como se observa en la figura 25 se conoce la altitud en metros de un punto (recuadro derecho de la figura) y la pendiente (recuadro inferior a la izquierda).

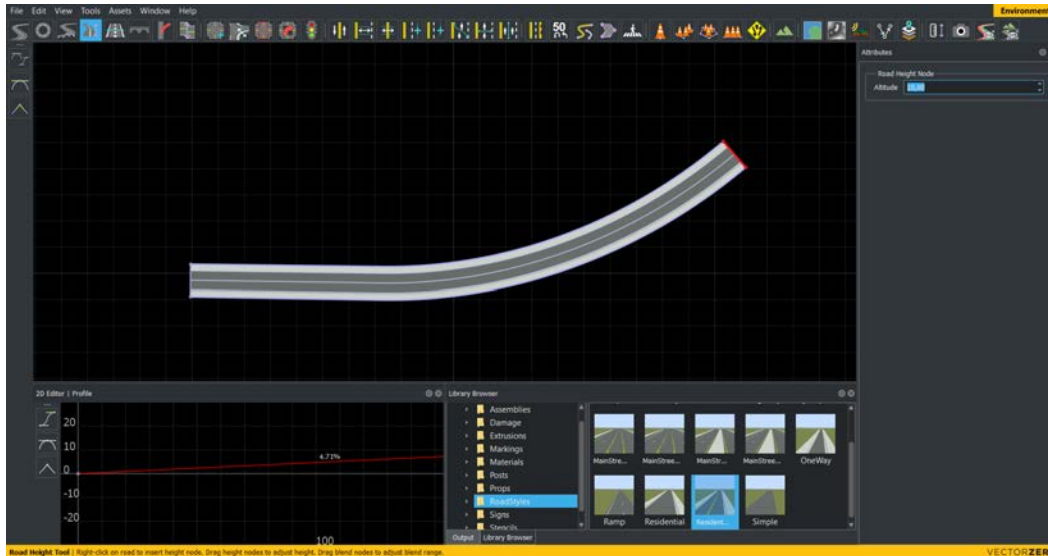


Figura 25. Configuración de la altitud

- Configuración de los carriles: esta herramienta permite seleccionar el sentido de circulación y el uso al que está destinado el carril (recuadro de la derecha de la figura 26). Existen multitud de opciones de tipo de carril como carril bici, carril restringido, parking, entrada, etc.

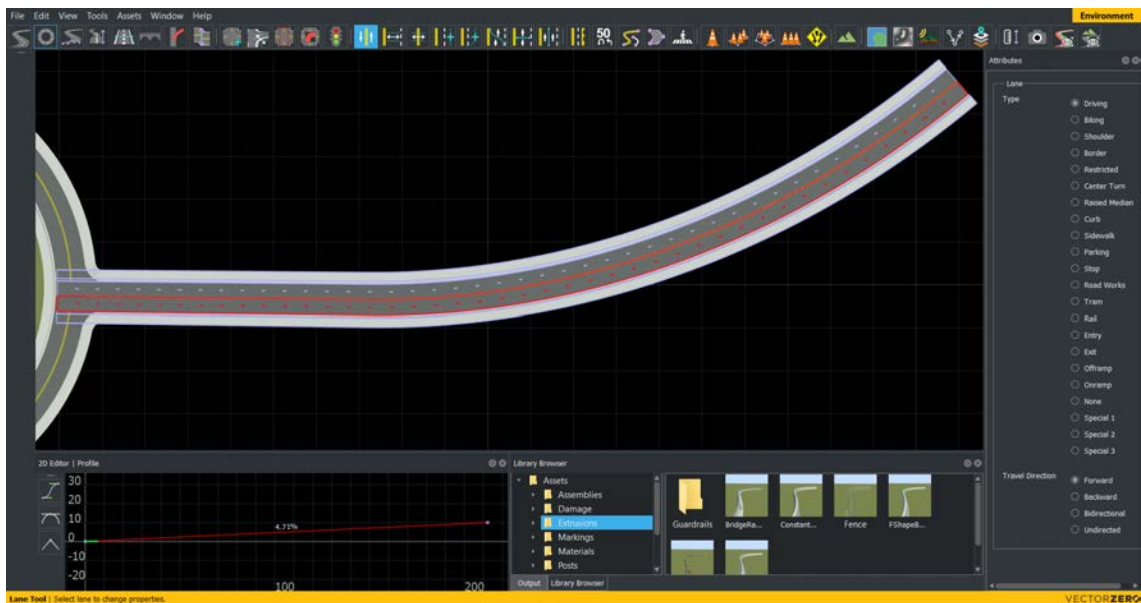


Figura 26. Configuración de carril

- Configuración de líneas: permite la selección del tipo de línea (línea continua, línea discontinua, línea continua amarilla, línea de carril bus, etc.). Además, permite modificar la longitud de la línea de carril seleccionada (figura 27).

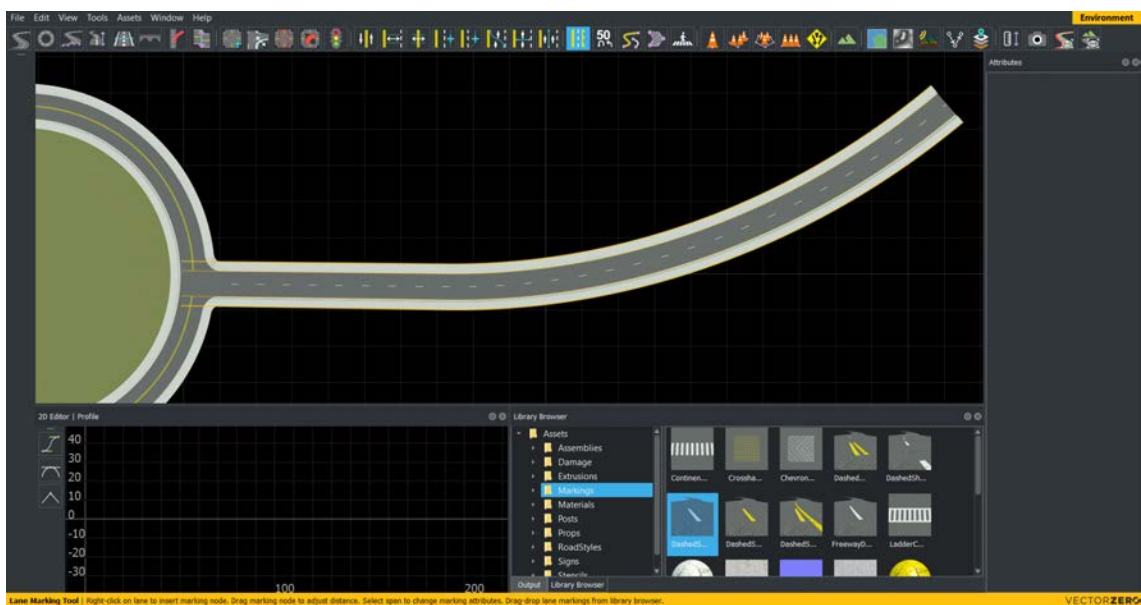


Figura 27. Configuración de línea de carril

- Modificación de la amplitud de los carriles: gracias a esta herramienta se puede ampliar o disminuir la anchura de los carriles de dos formas distintas, ampliando el carril completo (figura 28) o únicamente parte del carril (figura 29).

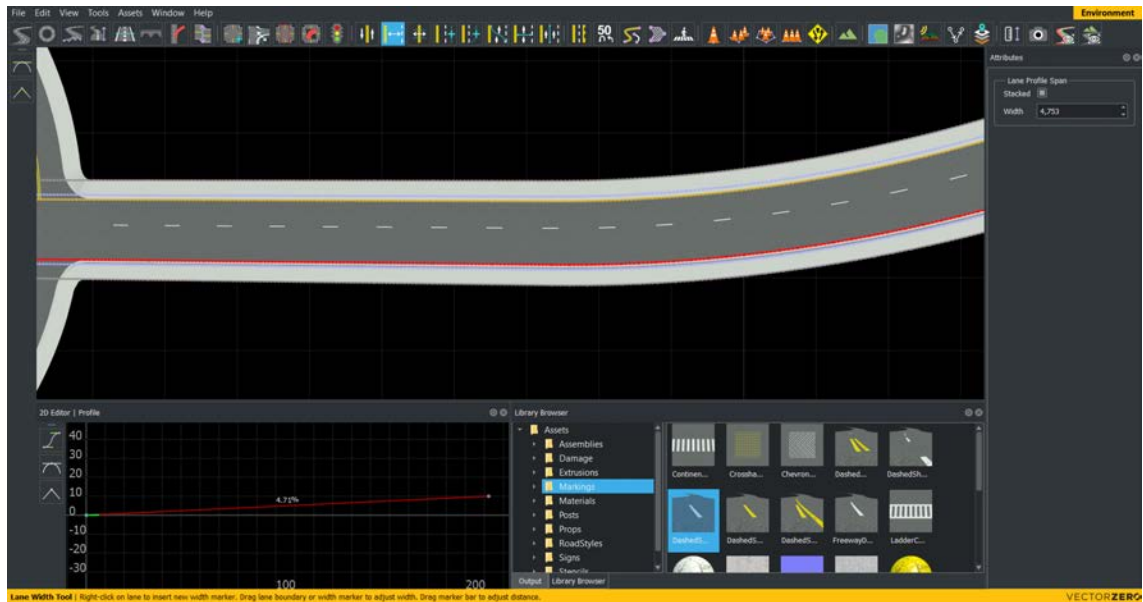


Figura 28. Modificación de la anchura del carril completo

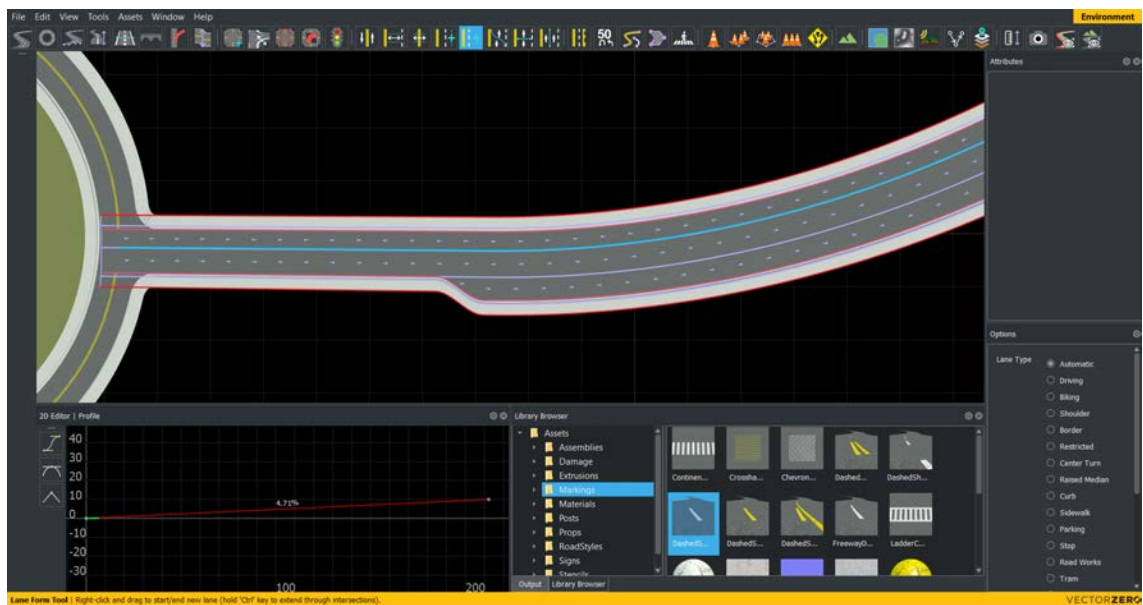


Figura 29. Modificación parcial de la anchura del carril

- Medidas de longitud: permite medir la distancia entre dos puntos del mapa. Gracias a esta herramienta y a Google Earth se ha conseguido crear carreteras con longitudes similares a las reales de la ampliación universitaria.

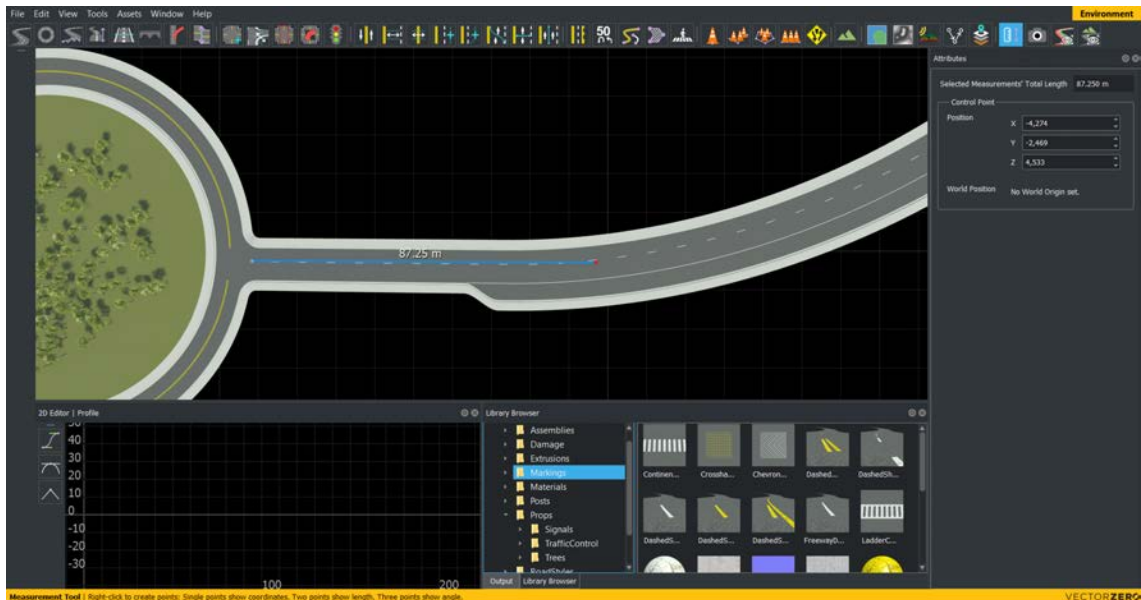


Figura 30. Medidas de longitud

- Adición de objetos: esta herramienta permite seleccionar el tipo y la cantidad de objetos que se quieren añadir al mapa (árboles en el caso mostrado en la figura 31). Se puede añadir un único objeto, varios objetos en línea o en una zona cerrada formada por la unión de varios puntos como se observa en la figura 31.

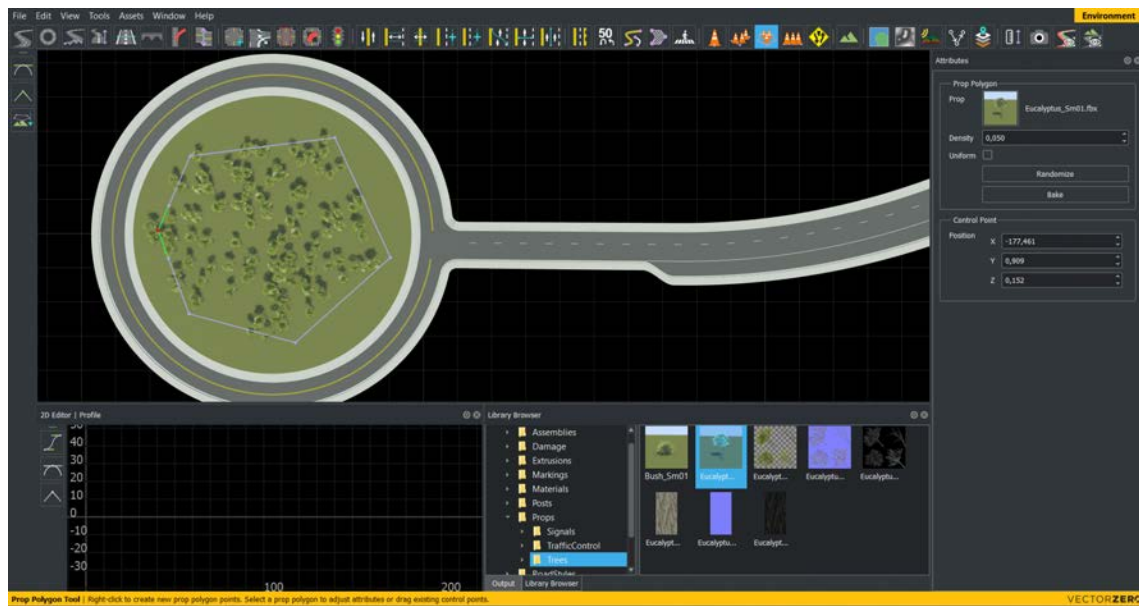


Figura 31. Herramienta para añadir objetos

3.2.2. Creación de la ampliación del Campus de Teatinos

Para el modelado de la ampliación universitaria se han utilizado las herramientas mencionadas en el apartado anterior. Para crear las diferentes carreteras del entorno, debido a la dificultad que entraña realizar las medidas en el entorno real, se ha utilizado Google Earth para medir la altitud y la longitud de la carretera a modelar y posteriormente se han creado con las medidas obtenidas. Por tanto, las medidas no son al cien por cien precisas y por tanto las carreteras construidas tampoco, aunque son aproximaciones que se asemejan bastante a la realidad, lo cual es suficiente para el objeto del proyecto. En las figuras 32 y 33, se muestran varias de las carreteras creadas en RoadRunner en comparación con las carreteras reales. Las carreteras tienen dimensiones similares a las calculadas con Google Earth, por lo que, aunque no sean exactamente iguales, el entorno virtual tendrá una escala prácticamente idéntica al entorno real. Las curvas y diferentes ángulos que toman las carreteras se han realizado a ojo, al no disponer de herramientas para hacer estas medidas con exactitud. Se observa en la figura 32 como las medidas de la recta situada en la puerta sur de la Escuela de Ingenierías son similares en Google Earth y en RoadRunner (356.3 m y 356.33 m, respectivamente).



Figura 32. Comparación de la longitud de la carretera delantera de la Escuela de Ingenierías. Medida de Google Earth arriba y medida en RoadRunner abajo.

Para determinar la altitud, se utiliza la herramienta de Google Earth situando el cursor en el punto concreto que se quiere medir. Para las carreteras, se mide la altitud en los dos extremos y se obtiene la diferencia. En RoadRunner se crea la carretera y se le indica la altitud en los dos extremos. La carretera sur de la Escuela de Ingenierías es el punto de referencia en cuanto altitud, siendo 0 m en este punto. A partir de aquí, se varia la altitud de acuerdo con las medidas en Google Earth. En la figura 33 se observan las medidas de altitud en la carretera este de la Escuela de Ingenierías. Esta varía de una altitud de 44 m a 49 m, por lo que la altitud aumenta 5 m. La rotonda a 44 m de altitud corresponde en RoadRunner con a la altitud inicial, por lo que en este programa la altitud será de 0 m, estando la rotonda superior a 4,60 m. En la imagen inferior se observa como este tramo de carretera tiene una pendiente del 5,08 %.



Figura 33. Comparación de la altitud de la carretera este de la Escuela de Ingenierías. Medida de Google Earth arriba y medida en RoadRunner abajo.

El resto de las carreteras se han creado siguiendo estos pasos hasta obtener un mapa similar al mostrado en la figura 20. En la figura 34 se observa una comparación del mapa real con el mapa construido en RoadRunner, los cuales son muy similares en cuanto a carreteras. La mayoría de la vegetación y los elementos no esenciales del mapa se han obviado, aunque algunos de ellos se añadirán posteriormente en CARLA con la biblioteca de objetos que tiene disponible para dotar al mapa de un mayor realismo.



Figura 34. Comparación del entorno real y el entorno virtual creado

Aunque en RoadRunner básicamente se han creado las diferentes carreteras y parkings del entorno de la ampliación, se han creado unas “plantillas” de edificios para utilizarlas como guía a la hora de crearlos. Para obtener las dimensiones de los edificios se ha utilizado la misma herramienta que para las medidas de las carreteras, teniendo en cuenta que la altura se ha aproximado al no disponer de herramientas que midan la altura de los edificios en Google Earth. En la figura 35 se comparan algunas de las mediciones realizadas a la Escuela de Ingenierías y la “plantilla” del edificio resultante en RoadRunner con las correspondientes medidas. Aunque las medidas no coinciden

exactamente, los edificios en el mapa virtual tienen una escala similar a los edificios reales.

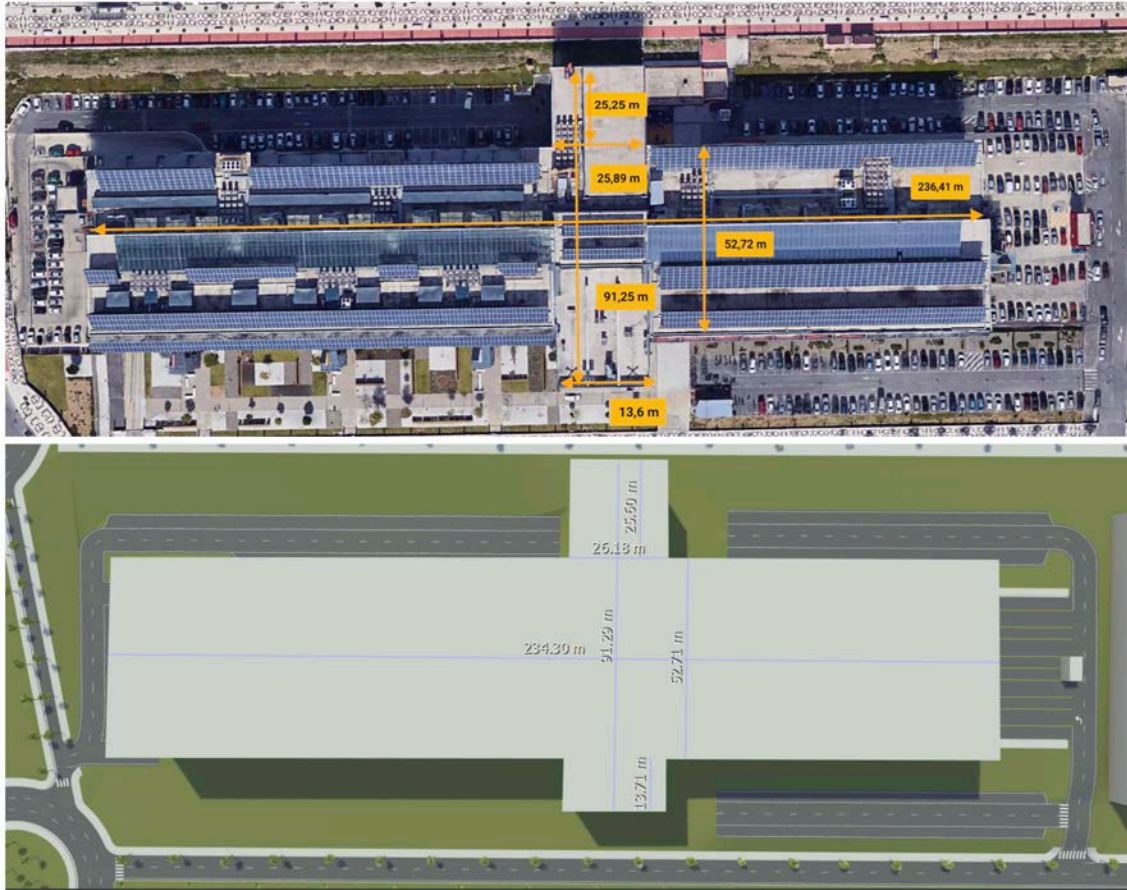


Figura 35. Comparación de medidas de edificios en Google Earth (arriba) y RoadRunner (abajo)

El objetivo de realizar multitud de mediciones y realizar aproximaciones de los edificios reales es dotar al mapa virtual del mayor realismo posible para obtener datos fiables en la simulación y tener la sensación de estar en un entorno real y conocido. Parte de este realismo se muestra en la figura 36 donde se compara parte de la ampliación en 3D, con imágenes obtenidas en Google Earth y en RoadRunner. Tanto carreteras como edificios tienen dimensiones muy similares, siendo una buena primera aproximación del modelo del mapa al que le faltan numerosos detalles que serán añadidos en etapas posteriores utilizando el propio simulador CARLA.



Figura 36. Comparación 3D de la zona de las universidades en Google Earth (arriba) y RoadRunner (abajo)

Antes de realizar la importación al simulador, es necesario comprobar en RoadRunner con la herramienta *OpenDRIVE Preview Tool* que las carreteras (en el sentido de circulación) y los diferentes cruces y rotondas están correctamente creados. Esto es necesario ya que el archivo en formato OpenDRIVE (formato de archivo para la descripción lógica de las carreteras) debe ser correcto para importarlo en CARLA sin errores.

La visualización completa de la ampliación utilizando la herramienta de OpenDRIVE en RoadRunner se muestra en la figura 37. Se observan distintos colores en función del sentido de circulación de los carriles, siendo rojo para un sentido y verde para el contrario. Además, existen otro tipo de colores, como azul, que indica que ese carril es un aparcamiento y por el está prohibida la circulación (carril de aparcamiento anexo a la Facultad de Ciencias de la Salud, figura 38).



Figura 37. Visualización de toda la ampliación utilizando la herramienta OpenDRIVE Preview Tool



Figura 38. Visualización de la zona de la Facultad de Ciencias de la Salud y la Facultad de Estudios Sociales y del Trabajo utilizando la herramienta OpenDRIVE Preview Tool

Además de cerciorarse que el sentido de circulación de los carriles es el correcto, es necesario comprobar que las salidas e incorporaciones de cruces y rotondas están creadas correctamente para evitar colisiones y una circulación incorrecta cuando se realicen pruebas de simulación. En la figura 39 se muestran las salidas e incorporaciones en la rotonda que une la Escuela de Ingenierías y la Facultad de Estudios Sociales y del Trabajo por la cara norte, en la que las salidas e incorporaciones se producen por el carril más a

la derecha, con la posibilidad de incorporarse al carril central o salirse de este en el caso de que la carretera tenga dos carriles de único sentido.



Figura 39. Salidas e incorporaciones de la rotonda norte que une la Escuela de Ingenierías y la Facultad de Estudios Sociales y del trabajo

3.3. Modelado en CARLA

Tras el modelado en el software RoadRunner de la ampliación universitaria de al Campus de Teatinos, el siguiente paso es importar el mapa resultante en CARLA y completarlo para poder realizar pruebas de conducción en el mismo. En CARLA, además de añadir edificios y otros elementos que doten al mapa de realismo, se configurarán los caminos por los que se circularán los coches y las diferentes normas de circulación (velocidades máximas, señales de ceda el paso, de stop, etc.). Al final de esta etapa, quedará un mapa completamente funcional donde se puedan probar diferentes aspectos de la conducción, para posteriormente sacar conclusiones acerca del comportamiento del simulador.

3.3.1. Importación en CARLA

Tras el modelado en RoadRunner del entorno de la ampliación, hay que importar el resultado en CARLA para realizar las pruebas de simulación. Este paso, complejo en las primeras versiones del simulador, se ha simplificado bastante con la creación de nuevos métodos de importación mucho más sencillos, ejecutando un único comando.

En primer lugar, se procede a exportar los ficheros necesarios de RoadRunner (FBX que contiene toda la información del mapa relacionado con las estructuras y texturas y XODR (OpenDRIVE) que, como se ha comentado con anterioridad en esta memoria, contiene la información relativa a la circulación). El proceso es sencillo, ya que RoadRunner dispone de una herramienta que exporta automáticamente los ficheros necesarios para la importación en CARLA (figura 40).

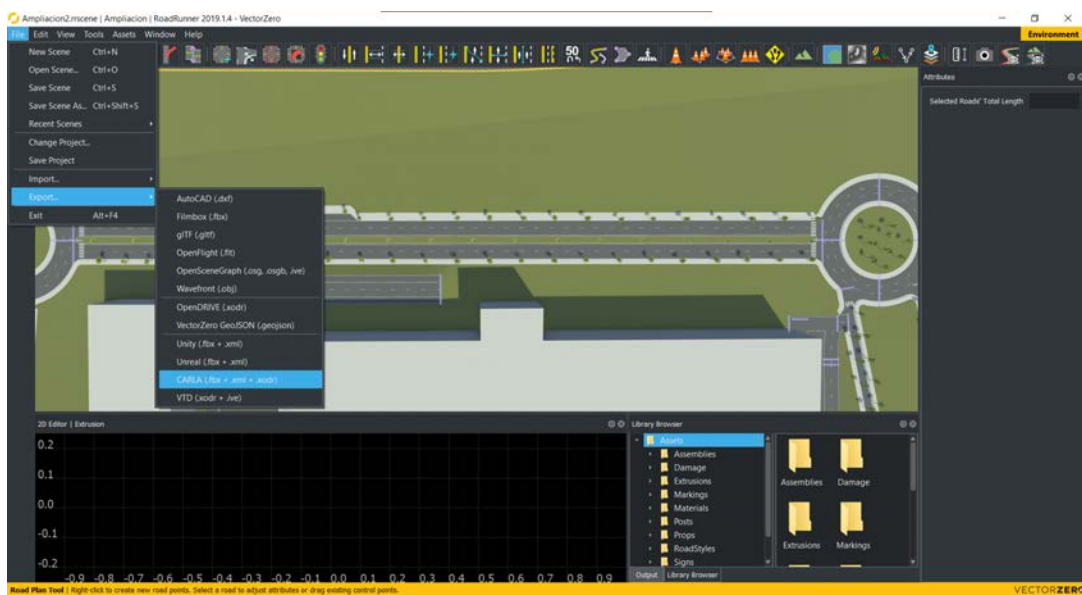


Figura 40. Exportación de ficheros en RoadRunner

Una vez obtenidos los ficheros necesarios para la importación, es necesario crear un archivo en formato JSON con diversa información como el nombre del mapa, la localización de los archivos y el tipo de materiales a utilizar en el mapa. Este archivo tiene una estructura fija formada por dos arrays (*maps* para importar mapas y *props* para importar objetos). Dentro de cada array cada objeto está formado por cuatro variables que

contienen el nombre del mapa, la localización de los ficheros FBX y XODR y el uso o no de los materiales de CARLA. En el código 2 se muestra el fichero *Package1.json* creado para la importación del mapa de la ampliación. El nombre del mapa es *Ampliacion2* y se ha seleccionado el uso de los materiales de RoadRunner en lugar de los de CARLA. Se observa cómo se ha añadido otro mapa simple a modo de prueba y comparación, aunque se trabajará únicamente con el mapa de la ampliación. En el segundo array (*props*) no se ha añadido nada ya que CARLA dispone de una amplia biblioteca con la que se pueden añadir numerosos activos al mapa.

```
{
  "maps": [
    {
      "name": "Ampliacion2",
      "source":
"/home/eloy96vg/carla/Import/Package1/Ampliacion2/Ampliacion2.fbx",
      "use_carla_materials": false,
      "xodr":
"/home/eloy96vg/carla/Import/Package1/Ampliacion2/Ampliacion2.xodr"
    },
    {
      "name": "Ciudad1",
      "source":
"/home/eloy96vg/carla/Import/Package1/Ciudad1/Ciudad1.fbx",
      "use_carla_materials": false,
      "xodr":
"/home/eloy96vg/carla/Import/Package1/Ciudad1/Ciudad1.xodr"
    }
  ],
  "props": []
}
```

Código 2. Fichero *Package.json* para la importación del mapa de la ampliación universitaria

Con los ficheros en la ruta indicada y el archivo JSON creado, se procede a importar el mapa mediante la ejecución del comando *make import* (dentro de la ruta de la carpeta de CARLA) en la consola de comandos.

Al abrir el simulador aparece un mapa similar al creado en RoadRunner pero con algunas características de CARLA que serán explicadas en los sucesivos apartados.

3.3.2. Ajuste de los errores de importación

Tras ejecutar el comando *make import* y abrir el simulador, el mapa aparece automáticamente como se observa en la figura 41. El mapa es idéntico al creado en RoadRunner, con unas texturas similares al disponer ambos softwares de mecanismos

para que estas no se pierdan. En concreto, CARLA utiliza *plugins* que permiten la importación correcta de mapas procedentes de RoadRunner. Las carreteras y las “plantillas” de los edificios se han importado correctamente, por lo que el siguiente paso es comenzar a añadir objetos y dotar de vida al mapa.

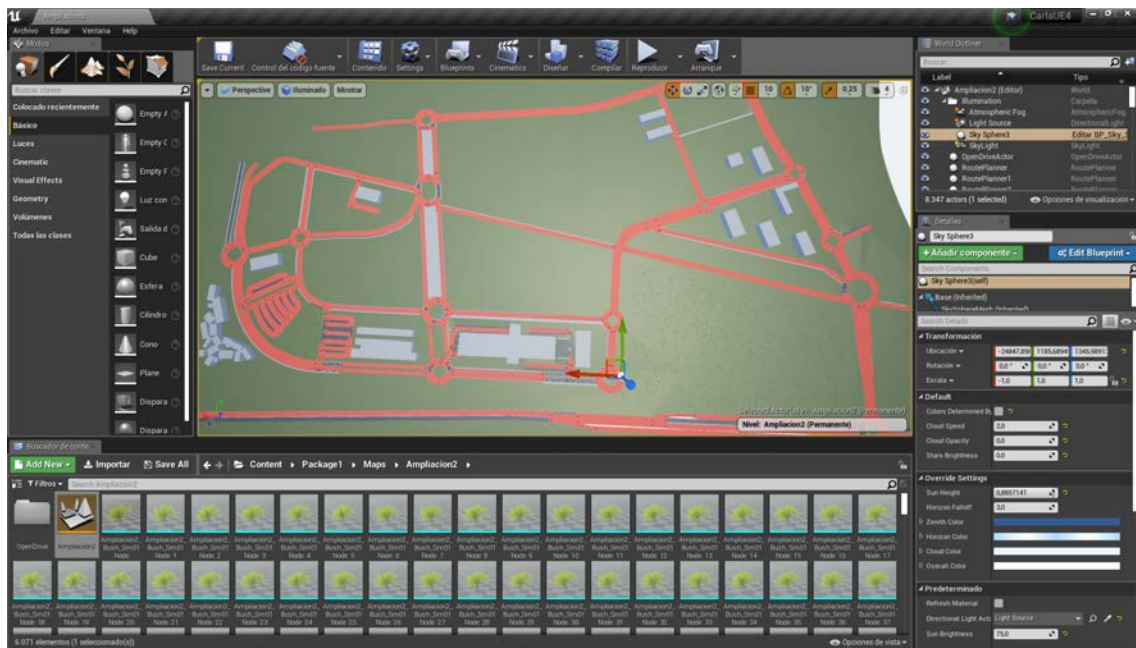


Figura 41. Mapa de la ampliación universitaria importado a CARLA

En la figura 41, se puede observar cómo aparecen una serie de puntos rojos a lo largo de todas las carreteras del mapa. Estos puntos, creados a partir de la información OpenDRIVE del fichero XODR, son una especie de guía por la que circularán los vehículos a través del mapa. Los puntos están relacionados con una posición GPS y cuando el vehículo vaya circulando se comparará la posición GPS actual con la de los diferentes puntos para asegurarse de que se sigue el camino correcto. Gracias a estos puntos, los vehículos circulan por el mapa de forma correcta realizando cruces, salidas e incorporaciones sin invadir un carril indebido, provocando accidentes. Estos puntos intermedios, unen varios *RoutePlanners* (figura 42) que son una especie de cajas que unen los diferentes puntos o rutas del mapa. Los *RoutePlanners* tienen especial interés en los cruces, incorporaciones y salidas de rotondas ya que unen ruta que sigue el vehículo por la carretera con la ruta de la rotonda. Esto es muy útil para evitar que el vehículo se incorpore por ejemplo desde el carril derecho de la carretera al carril interior de la rotonda,

acción poco correcta causante de muchos accidentes en la vida real. En este primer mapa recién importado, aun no se recogen otras indicaciones de circulación como cedas el paso y *stops*, o indicaciones de velocidad, las cuales serán añadidas posteriormente con las herramientas de CARLA.

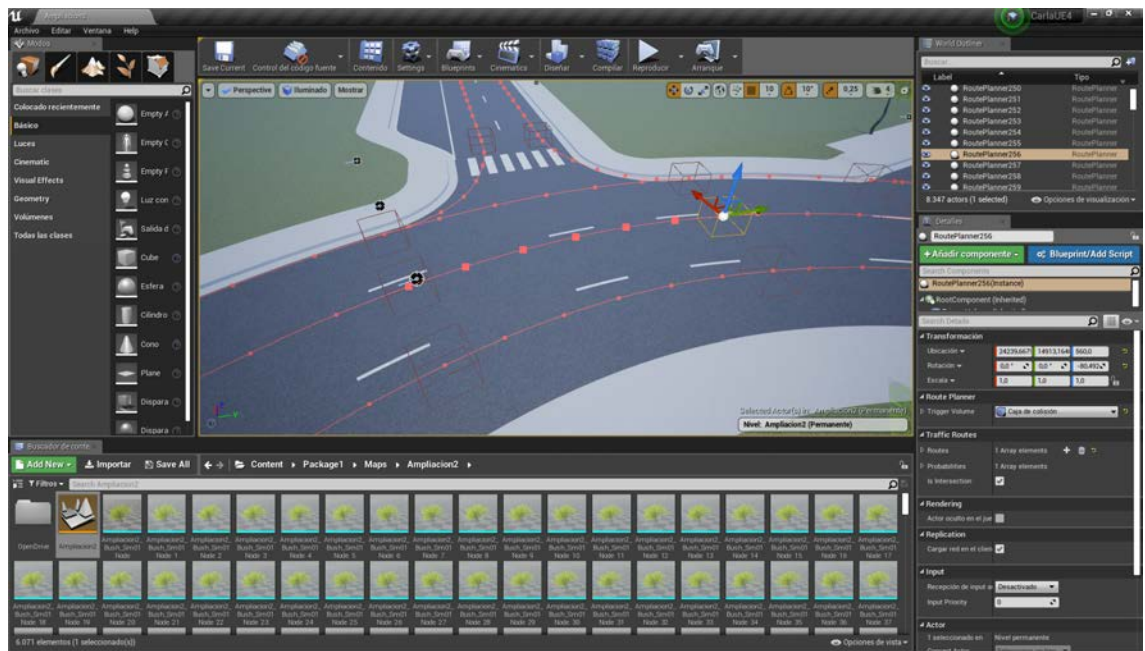


Figura 42. RoutePlanners y puntos intermedios en una rotonda del mapa

Aunque por lo general los *RoutePlanners* y los puntos intermedios se crean correctamente de forma automática, en algunos casos las rutas aparecen incompletas y faltan puntos (figura 43). En estos casos es necesario corregir los fallos para evitar que el vehículo siga un camino no deseado y pueda ocasionar colisiones e invasiones de carril indebidas. En la figura 44 se muestra un ejemplo de la corrección realizada en la ruta incompleta de la figura 43, en la que se ha añadido un *RoutePlanner* y los correspondientes puntos intermedios entre las “cajas” de la carretera principal y la rotonda. En el caso mostrado en las figuras 43 y 44, el vehículo solo podía incorporarse a la rotonda desde el carril derecho, pero tras la modificación también se puede realizar una incorporación al carril central de la rotonda desde el carril izquierdo de la carretera principal.



Figura 43. Ruta incompleta en la incorporación a una rotonda



Figura 44. Ruta corregida en la incorporación a una rotonda

3.3.3. Adición de componentes estáticos

El último paso para hacer al mapa más realista es añadir componentes estáticos, como edificios, coches aparcados, personas, etc. Para ello, como se ha comentado anteriormente, CARLA dispone de una amplia librería con gran variedad de componentes los cuales se irán añadiendo en función del entorno real. Cabe destacar, que los diferentes edificios y elementos añadidos son aproximaciones a los reales ya que se ha trabajado a partir de los elementos disponibles en lugar de crearlos de cero, lo cual es una ardua tarea

que tiene poca importancia con el objetivo del trabajo, que es la simulación de vehículos autónomos en un mapa aproximado a un entorno real. Por tanto, en cuanto a componentes, el entorno tendrá cierta similitud con el real sin ser exactamente igual.

En primer lugar, se han añadido los edificios, cuya única tarea ha sido en este punto seleccionar el tipo de edificio y adecuarlos al tamaño de las plantillas creadas en RoadRunner (figura 45). En las sucesivas figuras se observan varias comparaciones de los edificios de las tres facultades de la ampliación en comparación con los edificios reales (utilizando imágenes de Google Earth).



Figura 45. Edificio de la Escuela de Ingenierías en CARLA (arriba) y Google Earth (abajo)

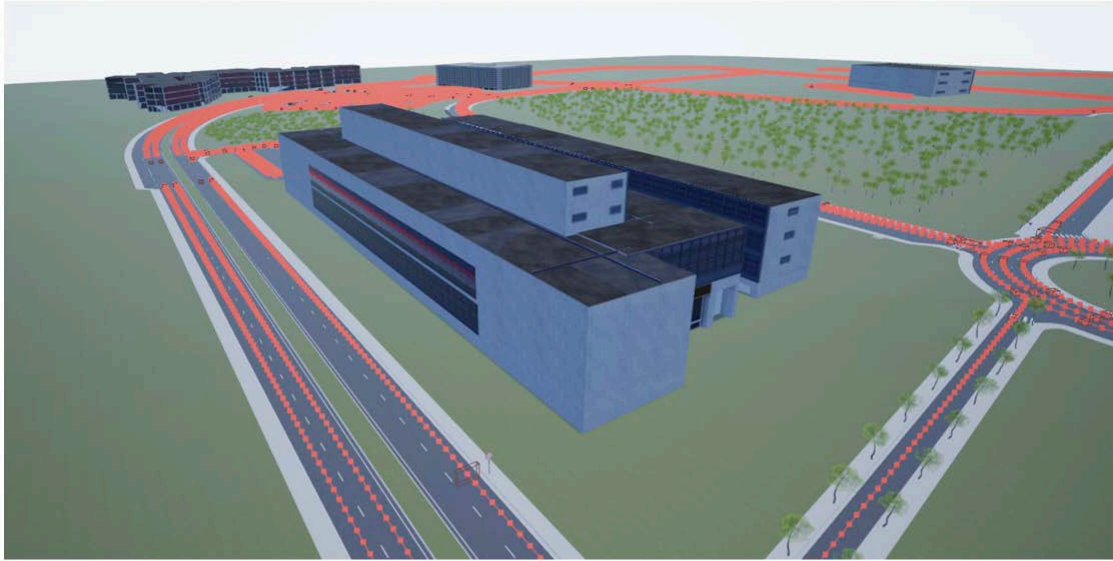


Figura 46. Edificio de la Facultad de Comercio y Gestión/Facultad de Estudios Sociales y del Trabajo en CARLA (arriba) y Google Earth (abajo)



Figura 47. Facultad de Ciencias de la Salud en CARLA (arriba) y Google Earth (abajo)

Además, se han modelado también edificios residenciales y sus respectivas vallas de separación con la carretera como se observa en la figura 48.



Figura 48. Edificios residenciales en CARLA (arriba) y Google Earth (abajo)

Aparte de los edificios, también se han añadido otros componentes visuales que, aunque no tendrán efectos sobre la simulación, harán al mapa mucho más realista. Estos elementos son árboles, paradas de autobús, vallas, gasolineras, etc. En la figura 49 se pueden ver algunos de estos elementos añadidos al mapa de acuerdo con los elementos reales.



Figura 49. Diferentes componentes estáticos añadidos al mapa

Otro de los elementos importantes añadidos al mapa han sido personas y vehículos aparcados (figura 50). Los vehículos permanecen estáticos, ya que los vehículos móviles se añadirán durante la simulación. Las personas añadidas, son estáticas en principio, aunque en etapas posteriores se podrá seleccionar que se muevan por las zonas designadas para ello (aceras y pasos de peatones).



Figura 50. Vehículos estáticos aparcados y personas añadidas al mapa

Por último, uno de los componentes estáticos añadidos con mayor importancia han sido las señales de regulación de circulación (stop, ceda el paso, límites de velocidad). Estas señales no son simplemente componentes de decoración en el mapa, sino que realmente regulan la circulación durante la circulación. Una de las señales más importantes para regular la circulación es la señal de ceda el paso (figura 51). Este elemento está compuesto de varios componentes en CARLA. En primer lugar, la señal física para saber de qué señal se trata y dos “cajas” que detectarán la presencia de vehículos. El funcionamiento es básico, si al llegar el vehículo a la caja de entrada a la rotonda detecta la presencia de otro vehículo circulando por la caja situada en el interior de la rotonda, este se detiene hasta que pase el otro vehículo. De esta forma, se concede prioridad de paso al vehículo que circula por la intersección circular y se evitan colisiones. El funcionamiento de las señales de stop es similar, aunque en este caso el vehículo se detiene de forma obligatoria durante algunos segundos.



Figura 51. Señal de ceda el paso

Otra de las señales que regulan el tráfico son las de velocidad. Estas señales, también estarán formadas por un *RoutePlanner* que hará que el vehículo que pase sobre él se adapte a la velocidad indicada. En la figura 52 se representa un ejemplo de señal de límite de velocidad, en la que se puede ver también cómo se selecciona la velocidad máxima a la que el vehículo podrá circular por ese tramo. El vehículo mantendrá una velocidad aproximada a la indicada en la señal hasta que se encuentre otra señal o una situación de peligro inminente en la que disminuirá su velocidad automáticamente. Ya que en la ampliación del Campus de Teatinos la velocidad está limitada a 50 en la mayoría de las carreteras, se han modificado los límites de velocidad en algunas zonas para estudiar el comportamiento del simulador en diferentes condiciones y velocidades.

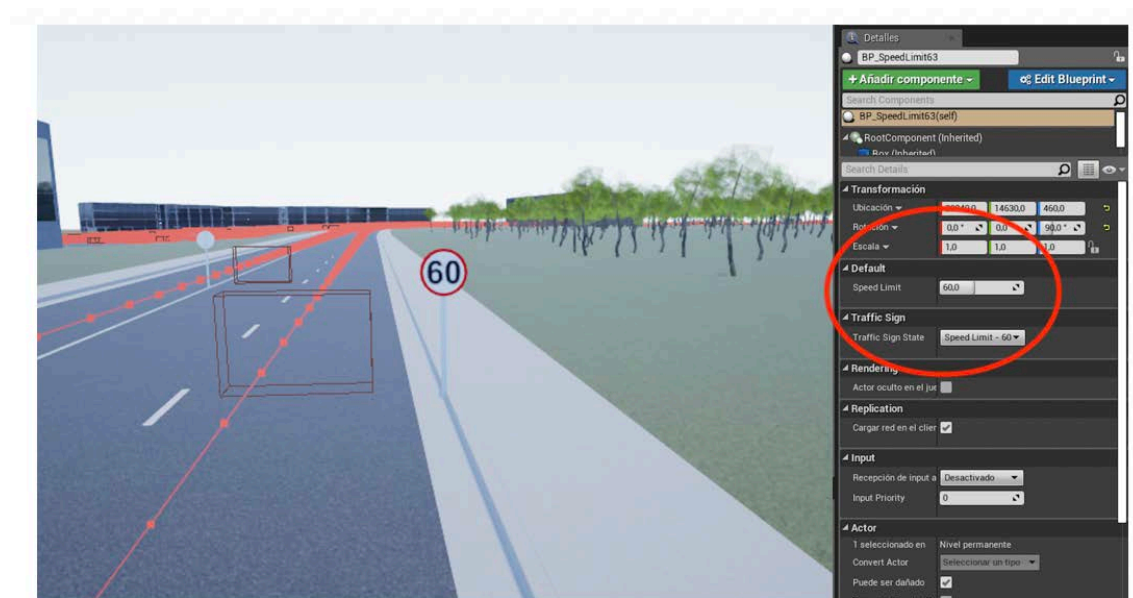


Figura 52. Señal de límite de velocidad y su configuración

Además, se pueden incluir otros reguladores de tráfico como los semáforos. En este caso, en la zona de la ampliación únicamente existen dos semáforos en la zona de paso del metro en la entrada a la ampliación. El funcionamiento del semáforo es similar al del resto de señales, formado por un *RoutePlanner* que dependiendo el color del semáforo hace que los vehículos se paren o continúen con su marcha. Como se ve en la figura 53, se puede configurar el tiempo que estará encendido cada color, en este caso 60 s en verde, 2 s en ámbar y 15 s en rojo.



Figura 53. Semáforo y configuración

Tras añadir todos los componentes mencionados por todo el mapa, este quedaría completo y listo para realizar pruebas de simulación de conducción autónoma en el entorno creado. En la figura 54 se muestra la evolución del mapa de RoadRunner al mapa completado de CARLA.



Figura 54. Evolución del mapa de RoadRunner a CARLA

Capítulo 4

Implementación

4.1. Introducción

En el anterior capítulo se detalló una de las partes más importantes del proyecto, que es el modelado de la ampliación del campus universitario de Teatinos mediante la utilización de los softwares RoadRunner y CARLA. Al final de esta etapa, se obtuvo un mapa completamente funcional en el que realizar pruebas de simulación de conducción autónoma. Por tanto, el siguiente paso es realizar la implementación y preparar el resto del entorno para la realización de pruebas, desde la configuración del vehículo autónomo hasta la configuración de las condiciones de simulación y conducción. Una de las principales características que es necesaria configurar es los sensores de los que dispondrá el vehículo y el lugar en el que estarán situados. Además, se definirán distintas configuraciones para realizar diferentes simulaciones con diversas condiciones climatológicas y se ajustará el mapa para realizar dichas pruebas. En definitiva, en esta etapa se dejará todo listo para poder realizar las pruebas de conducción en el mapa creado y poder sacar conclusiones a partir de estas. Concretamente, se realizarán dos tipos de pruebas. En una primera prueba se realizará una simulación en tiempo real (en el sentido de controlar directamente el vehículo mediante la interfaz de usuario) en el entorno de la ampliación bajo diferentes condiciones climatológicas y en la que se visualizará, también en tiempo real, las imágenes y datos captados por los diferentes sensores de los que estará compuesto el vehículo. En segundo lugar, se realizará una simulación centrada en la captura y almacenamiento de datos para procesarlos posteriormente y obtener

información del comportamiento de estos. Se realizará una prueba utilizando el modo de guiado autónomo de CARLA (del que se hablará en el apartado 4.2.5) y una segunda prueba en control manual en el que se expondrá a los sensores a situaciones más extremas como colisiones y movimientos bruscos.

En ambos casos, el código tendrá una estructura similar con pequeñas modificaciones. En el diagrama 1 se muestran las diferentes clases y funciones por las que esta formado el programa *manual_control.py*.

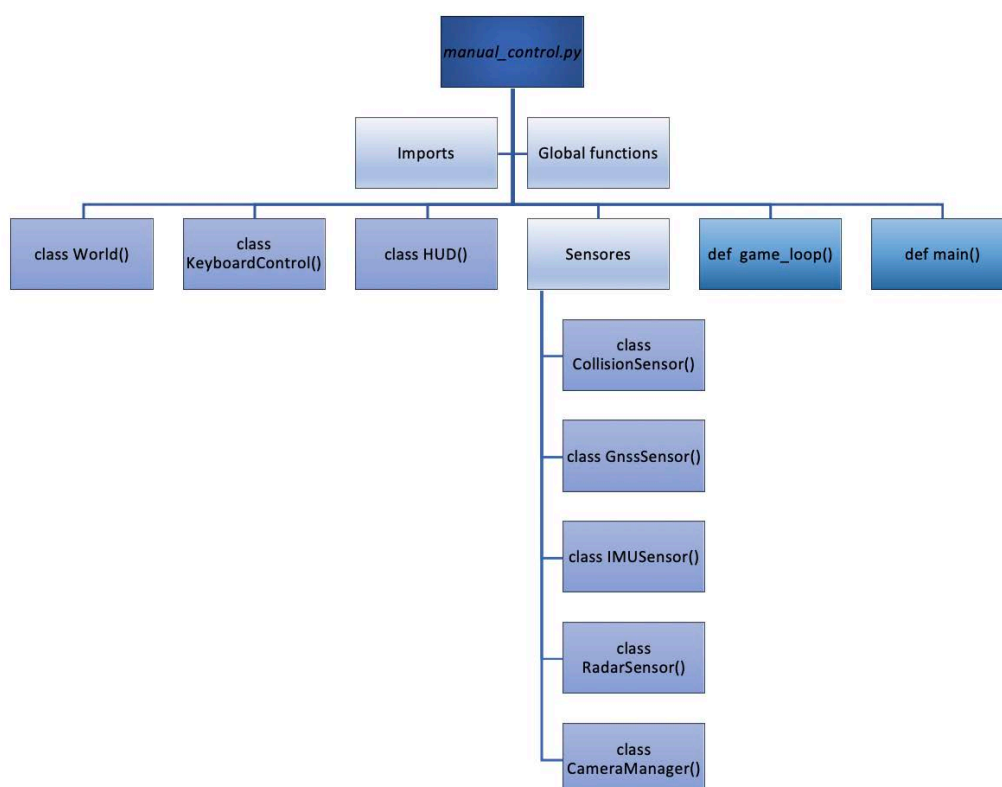


Diagrama 1. Diagrama de bloques del programa *manual_control.py*

4.2. Configuración de la simulación en tiempo real

En el presente apartado, se detallará la implementación para la realización de una simulación en tiempo real en la que se podrá seguir el vehículo y todos los acontecimientos ocurridos a su alrededor. Además, con este modo se podrá manejar el vehículo manualmente y configurar diferentes aspectos al momento como las condiciones climáticas y la cantidad de vehículos controlados por la CPU. Uno de los aspectos más

importantes que hay que configurar es el vehículo, añadiéndole los diferentes sensores con los que contará y su disposición en el vehículo. Se ha desarrollado un programa en Python que permite la simulación y visualización en pantalla en tiempo real. El programa está disponible en los ficheros de CARLA (*manual_control.py*) y en los sucesivos apartados se detallará el contenido de este. La simulación se lleva a cabo gracias a un bucle (código 3) que utilizará las diferentes funciones detalladas en los siguientes apartados. El bucle de simulación consistirá en una función en la que en primer lugar se realizará una conexión con el Unreal Editor, en el cual se iniciará una sesión de juego (figura 55) necesaria para que el programa se pueda conectar a CARLA. Una vez conectado al editor, se recupera el mundo, el cual se define en la función detallada en el apartado 4.2.1 y se obtienen todos los elementos de este (mapas, sensores, vehículos, etc.). En el resto del bucle se tienen en cuenta diferentes aspectos, como la visualización de la situación en pantalla y el control de teclado, así como un control del tiempo de simulación y la finalización de este bucle.

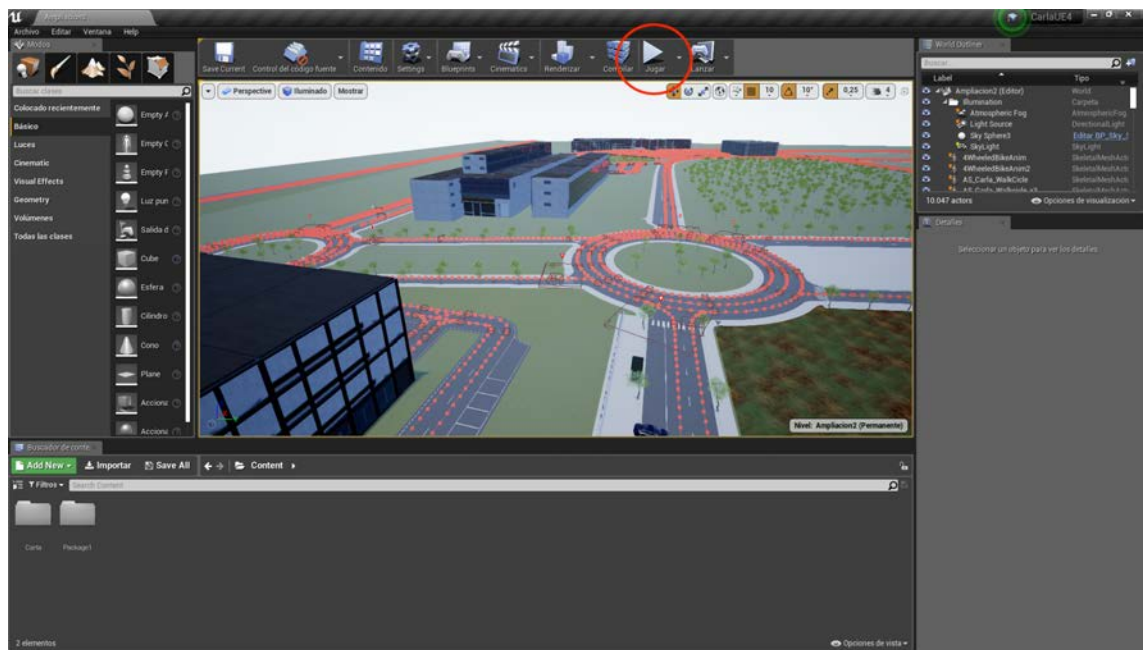


Figura 55. Iniciación de sesión de juego en el editor

```
def game_loop(args):
    pygame.init()
    pygame.font.init()
    world = None

    try:
        client = carla.Client(args.host, args.port)
```

```

client.set_timeout(2.0)

display = pygame.display.set_mode(
    (args.width, args.height),
    pygame.HWSURFACE | pygame.DOUBLEBUF)

hud = HUD(args.width, args.height)
world = World(client.get_world(), hud, args)
controller = KeyboardControl(world, args.autopilot)

clock = pygame.time.Clock()
while True:
    clock.tick_busy_loop(60)
    if controller.parse_events(client, world, clock):
        return
    world.tick(clock)
    world.render(display)
    pygame.display.flip()

finally:

    if (world and world.recording_enabled):
        client.stop_recorder()

    if world is not None:
        world.destroy()

pygame.quit()

```

Código 3. Bucle de simulación

4.2.1. Creación del mundo

En primer lugar, es necesario importar los diferentes módulos y paquetes necesarios para el desarrollo del programa de simulación a tiempo real. Uno de los primeros pasos, es la creación del “mundo” o entorno en el que se desarrollará la simulación. El código completo se muestra en el anexo 1, aunque en este y los sucesivos apartados se detallarán las partes más importantes. Esta función, se encarga básicamente de inicializar todos los aspectos del mundo para que puedan ser utilizados y configurados por otras funciones posteriormente. En código 4 se observa cómo se inicializan y se configuran los diferentes sensores que serán utilizados a la hora de posicionarlos en el vehículo. Por ejemplo, para configurar el sensor de colisión, en primer lugar, se inicializa:

self.collision_sensor = None.

A continuación, se configura utilizando la clase creada para el sensor de colisión:

self.collision_sensor = CollisionSensor(self.player, self.hud).

La configuración del resto de sensores se realiza de manera similar, inicializando en primer lugar y configurándolos posteriormente. Además, en esta clase se definirá la función `next_weather()` que permite cambiar entre las diferentes condiciones meteorológicas disponibles.

```
class World(object):
    def __init__(self, carla_world, hud, args):
        self.world = carla_world
        self.actor_role_name = args.rolename
        try:
            self.map = self.world.get_map()
        except RuntimeError as error:
            print('RuntimeError: {}'.format(error))
            print(' The server could not send the OpenDRIVE (.xodr)
file:')
            print(' Make sure it exists, has the same name of your
town, and is correct.')
            sys.exit(1)
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None
        self.gnss_sensor = None
        self.imu_sensor = None
        self.radar_sensor = None
        self.camera_manager = None
        self._weather_presets = find_weather_presets()
        self._weather_index = 0
        self._actor_filter = args.filter
        self._gamma = args.gamma
        self.restart()
        self.world.on_tick(hud.on_world_tick)
        self.recording_enabled = False
        self.recording_start = 0

    def restart(self):
        self.player_max_speed = 1.589
        self.player_max_speed_fast = 3.713
        # Keep same camera config if the camera manager exists.
        cam_index = self.camera_manager.index if self.camera_manager
is not None else 0
        cam_pos_index = self.camera_manager.transform_index if
self.camera_manager is not None else 0
        # Get a random blueprint.
        blueprint =
random.choice(self.world.get_blueprint_library().filter(self._actor_
filter))
        blueprint.set_attribute('role_name', self.actor_role_name)
        if blueprint.has_attribute('color'):
            color =
random.choice(blueprint.get_attribute('color').recommended_values)
            blueprint.set_attribute('color', color)
        if blueprint.has_attribute('driver_id'):
            driver_id =
random.choice(blueprint.get_attribute('driver_id').recommended_value
s)
            blueprint.set_attribute('driver_id', driver_id)
```

```

        if blueprint.has_attribute('is_invincible'):
            blueprint.set_attribute('is_invincible', 'true')
        # set the max speed
        if blueprint.has_attribute('speed'):
            self.player_max_speed =
float(blueprint.get_attribute('speed').recommended_values[1])
            self.player_max_speed_fast =
float(blueprint.get_attribute('speed').recommended_values[2])
        else:
            print("No recommended values for 'speed' attribute")
        # Spawn the player.
        if self.player is not None:
            spawn_point = self.player.get_transform()
            spawn_point.location.z += 2.0
            spawn_point.rotation.roll = 0.0
            spawn_point.rotation.pitch = 0.0
            self.destroy()
            self.player = self.world.try_spawn_actor(blueprint,
spawn_point)
        while self.player is None:
            if not self.map.get_spawn_points():
                print('There are no spawn points available in your
map/town.')
                print('Please add some Vehicle Spawn Point to your
UE4 scene.')
                sys.exit(1)
            spawn_points = self.map.get_spawn_points()
            spawn_point = random.choice(spawn_points) if
spawn_points else carla.Transform()
            self.player = self.world.try_spawn_actor(blueprint,
spawn_point)
        # Set up the sensors.
        self.collision_sensor = CollisionSensor(self.player,
self.hud)
        self.lane_invasion_sensor = LaneInvasionSensor(self.player,
self.hud)
        self.gnss_sensor = GnssSensor(self.player)
        self.imu_sensor = IMUSensor(self.player)
        self.camera_manager = CameraManager(self.player, self.hud,
self._gamma)
        self.camera_manager.transform_index = cam_pos_index
        self.camera_manager.set_sensor(cam_index, notify=False)
        actor_type = get_actor_display_name(self.player)
        self.hud.notification(actor_type)

    def next_weather(self, reverse=False):
        self._weather_index += -1 if reverse else 1
        self._weather_index %= len(self._weather_presets)
        preset = self._weather_presets[self._weather_index]
        self.hud.notification('Weather: %s' % preset[1])
        self.player.get_world().set_weather(preset[0])

    def toggle_radar(self):
        if self.radar_sensor is None:
            self.radar_sensor = RadarSensor(self.player)
        elif self.radar_sensor.sensor is not None:
            self.radar_sensor.sensor.destroy()
            self.radar_sensor = None

    def tick(self, clock):

```

```

        self.hud.tick(self, clock)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy_sensors(self):
        self.camera_manager.sensor.destroy()
        self.camera_manager.sensor = None
        self.camera_manager.index = None

    def destroy(self):
        if self.radar_sensor is not None:
            self.toggle_radar()
        actors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor,
            self.imu_sensor.sensor,
            self.player]
        for actor in actors:
            if actor is not None:
                actor.destroy()

```

Código 4. Código para la creación del mundo

4.2.2. Control del teclado

Esta función se encarga de definir diferentes métodos con los que se va a poder controlar tanto el vehículo como diferentes aspectos de la simulación en tiempo real. Como se muestra en código 5, se asignan a diferentes teclas del teclado el control manual del vehículo durante la simulación (W para avanzar, A y D para girar, P para activar el control automático, números 1 al 9 para cambiar entre los diferentes sensores, C para cambiar el tiempo, etc). Estos controles aparecen indicados en el terminal al ejecutar el programa (figura 56).

```
eloy96vg@Legion-Y530: ~/carla/PythonAPI/examples
Archivo Editar Ver Buscar Terminal Ayuda
eloy96vg@Legion-Y530:~$ cd carla
eloy96vg@Legion-Y530:~/carla$ cd PythonAPI
eloy96vg@Legion-Y530:~/carla/PythonAPI$ cd examples
eloy96vg@Legion-Y530:~/carla/PythonAPI/examples$ python manual_control.py
pygame 1.9.6
Hello from the pygame community. https://www.pygame.org/contribute.html
INFO: listening to server 127.0.0.1:2000

Welcome to CARLA manual control.

Use ARROWS or WASD keys for control.

    W            : throttle
    S            : brake
    AD           : steer
    Q            : toggle reverse
    Space        : hand-brake
    P            : toggle autopilot
    M            : toggle manual transmission
    ,/.          : gear up/down

    TAB          : change sensor position
    `            : next sensor
    [1-9]        : change to sensor [1-9]
    G            : toggle radar visualization
    C            : change weather (Shift+C reverse)
    Backspace    : change vehicle

    R            : toggle recording images to disk

    CTRL + R     : toggle recording of simulation (replacing any previous)
    CTRL + P     : start replaying last recorded simulation
    CTRL + +     : increments the start time of the replay by 1 second (+SHIFT = 10 seconds)
    CTRL + -     : decrements the start time of the replay by 1 second (+SHIFT = 10 seconds)

    F1          : toggle HUD
    H/?         : toggle help
    ESC         : quit
```

Figura 56. Controles en el programa de simulación en tiempo real

```
class KeyboardControl(object):
    def __init__(self, world, start_in_autopilot):
        self._autopilot_enabled = start_in_autopilot
        if isinstance(world.player, carla.Vehicle):
            self._control = carla.VehicleControl()
            world.player.set_autopilot(self._autopilot_enabled)
        elif isinstance(world.player, carla.Walker):
            self._control = carla.WalkerControl()
            self._autopilot_enabled = False
            self._rotation = world.player.get_transform().rotation
        else:
            raise NotImplementedError("Actor type not supported")
        self._steer_cache = 0.0
        world.hud.notification("Press 'H' or '?' for help.",
                               seconds=4.0)

    def parse_events(self, client, world, clock):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            elif event.type == pygame.KEYUP:
                if self._is_quit_shortcut(event.key):
```

```

        return True
    elif event.key == K_BACKSPACE:
        world.restart()
    elif event.key == K_F1:
        world.hud.toggle_info()
    elif event.key == K_h or (event.key == K_SLASH and
pygame.key.get_mods() & KMOD_SHIFT):
        world.hud.help.toggle()
    elif event.key == K_TAB:
        world.camera_manager.toggle_camera()
    elif event.key == K_c and pygame.key.get_mods() &
KMOD_SHIFT:
        world.next_weather(reverse=True)
    elif event.key == K_c:
        world.next_weather()
    elif event.key == K_g:
        world.toggle_radar()
    elif event.key == K_BACKQUOTE:
        world.camera_manager.next_sensor()
    elif event.key > K_0 and event.key <= K_9:
        world.camera_manager.set_sensor(event.key - 1 -
K_0)
    elif event.key == K_r and not (pygame.key.get_mods()
& KMOD_CTRL):
        world.camera_manager.toggle_recording()
    elif event.key == K_r and (pygame.key.get_mods() &
KMOD_CTRL):
        if (world.recording_enabled):
            client.stop_recorder()
            world.recording_enabled = False
            world.hud.notification("Recorder is OFF")
        else:
            client.start_recorder("manual_recording.rec")
            world.recording_enabled = True
            world.hud.notification("Recorder is ON")

```

Código 5. Código para la configuración del control de teclado

4.2.3. Configuración de la pantalla

Esta parte del código se encarga de configurar la pantalla de simulación en la que se mostrará tanto el vehículo y el entorno, como diferente información útil para conocer el comportamiento del vehículo. Esta información se recoge de los diferentes sensores incorporados en el vehículo (cada sensor tiene sus diferentes métodos y funciones) y es mostrada en pantalla en el apartado dedicado para ello, como se observa en la figura 18. La información proporcionada es variada, como el modelo del vehículo utilizado, la velocidad, la marcha engranada, las colisiones, datos sensor IMU, etc. Por último, esta función se encargará de la visualización del vehículo con el que se está realizando la

simulación en pantalla y el entorno por medio de una conexión con el motor Unreal Engine en funcionamiento, como se mostraba en la figura 55.

```

class HUD(object):
    def __init__(self, width, height):
        self.dim = (width, height)
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        font_name = 'courier' if os.name == 'nt' else 'mono'
        fonts = [x for x in pygame.font.get_fonts() if font_name in
x]
        default_font = 'ubuntumono'
        mono = default_font if default_font in fonts else fonts[0]
        mono = pygame.font.match_font(mono)
        self._font_mono = pygame.font.Font(mono, 12 if os.name ==
'nt' else 14)
        self._notifications = FadingText(font, (width, 40), (0,
height - 40))
        self.help = HelpText(pygame.font.Font(mono, 24), width,
height)
        self.server_fps = 0
        self.frame = 0
        self.simulation_time = 0
        self._show_info = True
        self._info_text = []
        self._server_clock = pygame.time.Clock()

    def on_world_tick(self, timestamp):
        self._server_clock.tick()
        self.server_fps = self._server_clock.get_fps()
        self.frame = timestamp.frame
        self.simulation_time = timestamp.elapsed_seconds

    def tick(self, world, clock):
        self._notifications.tick(world, clock)
        if not self._show_info:
            return
        t = world.player.get_transform()
        v = world.player.get_velocity()
        c = world.player.get_control()
        compass = world.imu_sensor.compass
        heading = 'N' if compass > 270.5 or compass < 89.5 else ''
        heading += 'S' if 90.5 < compass < 269.5 else ''
        heading += 'E' if 0.5 < compass < 179.5 else ''
        heading += 'W' if 180.5 < compass < 359.5 else ''
        colhist = world.collision_sensor.get_collision_history()
        collision = [colhist[x + self.frame - 200] for x in range(0,
200)]
        max_col = max(1.0, max(collision))
        collision = [x / max_col for x in collision]
        vehicles = world.world.get_actors().filter('vehicle.*')
        self._info_text = [
            'Server:  % 16.0f FPS' % self.server_fps,
            'Client:  % 16.0f FPS' % clock.get_fps(),
            '',
            'Vehicle: % 20s' % get_actor_display_name(world.player,
truncate=20),
            'Map:      % 20s' % world.map.name,
            'Simulation time: % 12s' %

```

```

datetime.timedelta(seconds=int(self.simulation_time)),
    '..',
    'Speed:    % 15.0f km/h' % (3.6 * math.sqrt(v.x**2 +
v.y**2 + v.z**2)),
    u'Compass:% 17.0f\N{DEGREE SIGN} % 2s' % (compass,
heading),
    'Accelero: (%5.1f,%5.1f,%5.1f)' %
(world.imu_sensor.accelerometer),
    'Gyrosco: (%5.1f,%5.1f,%5.1f)' %
(world.imu_sensor.gyroscope),
    'Location:% 20s' % ('(% 5.1f, % 5.1f)' % (t.location.x,
t.location.y)),
    'GNSS:% 24s' % ('(% 2.6f, % 3.6f)' %
(world.gnss_sensor.lat, world.gnss_sensor.lon)),
    'Height:  % 18.0f m' % t.location.z,
    '..']
    if isinstance(c, carla.VehicleControl):
        self._info_text += [
            ('Throttle:', c.throttle, 0.0, 1.0),
            ('Steer:', c.steer, -1.0, 1.0),
            ('Brake:', c.brake, 0.0, 1.0),
            ('Reverse:', c.reverse),
            ('Hand brake:', c.hand_brake),
            ('Manual:', c.manual_gear_shift),
            'Gear:      %s' % {-1: 'R', 0: 'N'}.get(c.gear,
c.gear)]
    elif isinstance(c, carla.WalkerControl):
        self._info_text += [
            ('Speed:', c.speed, 0.0, 5.556),
            ('Jump:', c.jump)]
    self._info_text += [
        '..',
        'Collision:',
        collision,
        '..',
        'Number of vehicles: % 8d' % len(vehicles)]
    if len(vehicles) > 1:
        self._info_text += ['Nearby vehicles:']
        distance = lambda l: math.sqrt((l.x - t.location.x)**2 +
(l.y - t.location.y)**2 + (l.z - t.location.z)**2)
        vehicles = [(distance(x.get_location()), x) for x in
vehicles if x.id != world.player.id]
        for d, vehicle in sorted(vehicles):
            if d > 200.0:
                break
            vehicle_type = get_actor_display_name(vehicle,
truncate=22)
            self._info_text.append('% 4dm %s' % (d,
vehicle_type))

    def toggle_info(self):
        self._show_info = not self._show_info

    def notification(self, text, seconds=2.0):
        self._notifications.set_text(text, seconds=seconds)

    def error(self, text):
        self._notifications.set_text('Error: %s' % text, (255, 0,
0))

```

```

def render(self, display):
    if self._show_info:
        info_surface = pygame.Surface((220, self.dim[1]))
        info_surface.set_alpha(100)
        display.blit(info_surface, (0, 0))
        v_offset = 4
        bar_h_offset = 100
        bar_width = 106
        for item in self._info_text:
            if v_offset + 18 > self.dim[1]:
                break
            if isinstance(item, list):
                if len(item) > 1:
                    points = [(x + 8, v_offset + 8 + (1.0 - y) *
30) for x, y in enumerate(item)]
                    pygame.draw.lines(display, (255, 136, 0),
False, points, 2)
                item = None
                v_offset += 18
            elif isinstance(item, tuple):
                if isinstance(item[1], bool):
                    rect = pygame.Rect((bar_h_offset, v_offset +
8), (6, 6))
                    pygame.draw.rect(display, (255, 255, 255),
rect, 0 if item[1] else 1)
                else:
                    rect_border = pygame.Rect((bar_h_offset,
v_offset + 8), (bar_width, 6))
                    pygame.draw.rect(display, (255, 255, 255),
rect_border, 1)
                    f = (item[1] - item[2]) / (item[3] -
item[2])
                    if item[2] < 0.0:
                        rect = pygame.Rect((bar_h_offset + f *
(bar_width - 6), v_offset + 8), (6, 6))
                    else:
                        rect = pygame.Rect((bar_h_offset,
v_offset + 8), (f * bar_width, 6))
                    pygame.draw.rect(display, (255, 255, 255),
rect)
                    item = item[0]
                if item: # At this point has to be a str.
                    surface = self._font_mono.render(item, True,
(255, 255, 255))
                    display.blit(surface, (8, v_offset))
                    v_offset += 18
            self._notifications.render(display)
            self.help.render(display)

```

Código 6. Configuración de pantalla

4.2.4. Configuración del vehículo

El último paso para poder llevar a cabo la simulación es configurar los diferentes sensores que se colocarán en el vehículo y que permitirá la conducción autónoma mediante el procesamiento de las señales obtenidas por estos. Se utilizarán sensores de

colisión, de invasión de línea, de posición GNSS, sensor IMU, radar, LIDAR y diferentes tipos de cámaras para obtener diferentes imágenes del entorno. El procedimiento para la creación y configuración de los sensores es similar para todos, con algunas excepciones. En el Anexo 1 se muestra el código completo del programa *manual_control.py* donde se puede ver la parte correspondiente a la configuración del vehículo.

En primer lugar, se debe iniciar el sensor y crear un objeto (*blueprint*) indicando el tipo de sensor de entre los que se encuentran disponibles en la biblioteca de objetos:

```
bp=world.get_blueprint_library().find('sensor.other.collision').
```

Posteriormente, se coloca el sensor en el vehículo:

```
self.sensor = world.spawn_actor(bp, carla.Transform(), attach_to=self._parent).
```

A continuación, se crean alertas en el caso de que ocurra algún evento, como en los casos del sensor de colisión y el sensor de invasión de línea:

```
self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self, event)).
```

Por último, se definen los métodos necesarios para cada sensor, como el indicador de la intensidad de colisión, el indicador de posición GNSS, dando información de la latitud y la longitud en la que se encuentra el vehículo y otros métodos matemáticos para la conversión de datos y que puedan visualizarse de manera correcta.

Algunos sensores necesitan un tratamiento especial como es el caso de las cámaras. Estas deben estar situadas estratégicamente en alguna parte específica del vehículo por lo que hay que trasladarlas respecto al punto de origen del vehículo ($X = Y = Z = 0$). En este caso mediante:

```
self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0,  
z=1.8)), attach_to=self._parent),
```

el sensor se traslada 1 metro hacia delante y 1.8 metros hacia arriba. Por último, las imágenes de estos sensores deben ser tratadas para obtener diferentes tipos de imágenes como las de segmentación semántica y en escala de grises. En cuanto al LIDAR, necesita además una serie de procedimientos matemáticos como se observa en el código para crear la nube de puntos y procesar los datos para obtener información detallada del entorno.

En el capítulo 5.1 se muestran los resultados de esta primera simulación.

4.2.5. Guiado autónomo de CARLA

Aunque el sistema de conducción autónomo de CARLA es algo complejo, el automóvil tiene un comportamiento determinado en función de las situaciones en las que se encuentre. Para actuar, el sistema utiliza la información recogida por los sensores para determinar como proceder las diferentes situaciones por las que se va encontrando el vehículo. Por ejemplo, el vehículo se mueve hacia delante a la velocidad establecida por los límites recogidos en cada zona, siendo la velocidad predeterminada 30 km/h. Por tanto, el vehículo circulará a esta velocidad hasta que se encuentre una señal que indique que puede modificar su velocidad. El vehículo se detiene cuando detecta la presencia de otros automóviles o peatones en sus alrededores, así como si detecta alguna señal de stop, ceda el paso o semáforos en rojo. Esto se logra mediante los métodos *RayCast()* y *IsThereAnObstacleAhead()*, los cuales aparecen en el archivo *WheeledVehicleAIController.cpp* (Anexo A.3). Para que el vehículo actúe a partir de la información recogida por los diferentes sensores utilizados, el método *AWheeledVehicleAIController::ConfigureAutopilot()* se encarga de controlar el acelerador, el freno y el giro de las ruedas del vehículo. Además, el vehículo utiliza los datos del mapa (los datos OpenDRIVE) para mantenerse en el lado derecho de la carretera y no invadir carriles de sentido contrario o aceras. En el caso de que el vehículo ingrese en una intersección se activa un evento que llama al *RoutePlanner* correspondiente, el cual proporciona una determinada ruta que corresponde con el conjunto de puntos *spline* que determinará la dirección en la que se dirigirá el automóvil. Estos puntos se envían al controlador del vehículo mediante *SetFixedRoute()* en el fichero *RoutePlanner.cpp*. En caso de que existan varias rutas disponibles, se seleccionarán aleatoriamente mediante *RandomEngine*. Estas rutas, en lugar de determinarse aleatoriamente pueden ser establecidas introduciendo adecuadamente en el código la ruta deseada en cada una de las intersecciones existentes en el mapa. De forma resumida, este es el funcionamiento del piloto automático de CARLA en las situaciones en las que se puede encontrar el vehículo a lo largo de la simulación.

4.3. Configuración del sistema de almacenamiento de datos sensoriales

En este apartado se detallan los distintos métodos implementados para la captura y almacenamiento de datos, útiles para corroborar el correcto comportamiento del vehículo a partir de los datos sensoriales y de los datos de los actuadores del propio vehículo como los pedales de aceleración y el volante.

4.3.1. Almacenamiento de datos

De los numerosos sistemas sensoriales de los que dispone el vehículo se pueden extraer numerosos datos que resultan interesantes para llevarlos a estudio y comprender adecuadamente el funcionamiento de estos y del vehículo. Para ello, al código anterior se añadirá una serie de métodos para poder almacenar en ficheros de texto la información recogida por los sensores y posteriormente procesarla en Matlab, obteniendo una serie de gráficas que ayudarán a explicar el comportamiento del vehículo.

En primer lugar, el programa `manual.control.py` tiene una opción preprogramada para capturar las imágenes recogidas por las cámaras, así como la nube de puntos creada por el LIDAR. En este caso, únicamente hay que añadir una función para detectar si se ha pulsado el botón R, en cuyo caso comienza la captura de imágenes. Posteriormente, se utiliza el método `save_to_disk()` para guardar los datos en la carpeta deseada.

```
def toggle_recording(self):
    self.recording = not self.recording
    self.hud.notification('Recording %s' % ('On' if
self.recording else 'Off'))

@staticmethod
def _parse_image(weak_self, image):
    self = weak_self()
    if not self:
        return
    if self.sensors[self.index][0].startswith('sensor.lidar'):
        points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
        points = np.reshape(points, (int(points.shape[0] / 3), 3))
        lidar_data = np.array(points[:, :2])
        lidar_data *= min(self.hud.dim) / 100.0
        lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
        lidar_data = np.fabs(lidar_data) # pylint: disable=E1111
        lidar_data = lidar_data.astype(np.int32)
        lidar_data = np.reshape(lidar_data, (-1, 2))
```

```

        lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
        lidar_img = np.zeros((lidar_img_size), dtype = int)
        lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
        self.surface = pygame.surfarray.make_surface(lidar_img)
    else:
        image.convert(self.sensors[self.index][1])
        array = np.frombuffer(image.raw_data,
dtype=np.dtype("uint8"))
        array = np.reshape(array, (image.height, image.width, 4))
        array = array[:, :, :3]
        array = array[:, :, ::-1]
        self.surface =
pygame.surfarray.make_surface(array.swapaxes(0, 1))
    if self.recording:
        image.save_to_disk('Datos_Camara/%08d' % image.frame)

```

Código 7. Recogida de datos de las cámaras y el LIDAR

En segundo lugar, hay que recoger los datos del resto de sensores, como el sensor GNSS, el sensor inercial, sensor de colisión, etc. Además, se recogerá la información de aceleración, freno y giro que permite conocer la intensidad de la aceleración y el freno (valores entre 0 y 1) y el giro del vehículo (entre -1 y 1). Estos datos se pueden comparar con los obtenidos de los sensores para comprender el movimiento del vehículo durante la simulación. Para almacenar estos datos, como se muestra en el código 8, se creará un fichero de texto en el que se almacenarán los datos por columnas. En total se almacenan 17 datos en este fichero con la distribución mostrada en la tabla 1.

Columna	Dato recogido
1	Tiempo de simulación (s)
2	Velocidad (<i>km/h</i>)
3	Localización en X (<i>m</i>)
4	Localización en Y (<i>m</i>)
5	Localización en Z/Altitud (<i>m</i>)
6	Latitud (°)
7	Longitud (°)
8	Aceleración lineal eje X (<i>m/s²</i>)

9	Aceleración lineal eje Y (m/s^2)
10	Aceleración lineal eje Z (m/s^2)
11	Movimiento rotacional eje X (rad/s)
12	Movimiento rotacional eje Y (rad/s)
13	Movimiento rotacional eje Z (rad/s)
14	Brújula ($^\circ$)
15	Intensidad pedal aceleración [0,1]
16	Intensidad giro volante [-1,1]
17	Intensidad pedal freno [0,1]

Tabla 1. Datos recogidos de diferentes sensores

```

datos = open("datos.txt", "a")
datos.write('\n' +
str(datetime.timedelta(seconds=float(self.simulation_time)).total_se
conds()) + " " + str(
    3.6 * math.sqrt(v.x ** 2 + v.y ** 2 + v.z ** 2)) + " " +
str(t.location.x) + " " + str(
    t.location.y) + " " + str(t.location.z) + " " +
str(world.gnss_sensor.lat) + " " + str(
    world.gnss_sensor.lon) + " " +
str(world.imu_sensor.accelerometer[0]) + " " + str(
    world.imu_sensor.accelerometer[1]) + " " +
str(world.imu_sensor.accelerometer[2]) + " " + str(
    world.imu_sensor.gyroscope[0]) + " " +
str(world.imu_sensor.gyroscope[1]) + " " + str(
    world.imu_sensor.gyroscope[2]) + " " +
str(world.imu_sensor.compass) + " " + str(
    float(c.throttle)) + " " + str(float(c.steer)) + " " +
str(float(c.brake)))
datos.close()

```

Código 8. Almacenamiento de datos de los sensores del vehículo

Además, para recoger los datos del sensor de invasión de línea, de colisión y del radar se añaden las líneas de código mostradas en Código 9, 10 y 11. Del sensor de línea únicamente se recogerán datos cada vez que se cruce una línea y el tipo de línea que se ha cruzado. Los sensores de colisión y el radar recogerán los datos mostrados en las tablas 2 y 3. El sensor de colisión recoge los impulsos en cada uno de los ejes, que mide la fuerza con la que ha impactado el vehículo con un obstáculo. La intensidad de colisión se mide de la siguiente forma:

$$Intensidad = \sqrt{ImpulsoX^2 + ImpulsoY^2 + ImpulsoZ^2} \quad (1)$$

Por otro lado, el radar recoge el ángulo orientación respecto la superficie (azimut), la altitud, el ángulo de altitud, la distancia al objeto (profundidad), la velocidad hacia el sensor y una velocidad transformada que tendrá un rango entre -1 y 1.

Columna	Dato recogido
1	Impulso en X ($kg \cdot m/s$)
2	Impulso en Y ($kg \cdot m/s$)
3	Impulso en Z ($kg \cdot m/s$)
4	Intensidad de colisión ($kg \cdot m/s$)

Tabla 2. Datos recogidos por el sensor de colisión

Columna	Dato recogido
1	Azimut (rad)
2	Altitud (rad)
3	Profundidad (m)
4	Velocidad (m/s)
5	Velocidad normal (m/s)

Tabla 3. Datos recogidos por el radar

4.3.2. Procesamiento de datos en Matlab

El programa creado en Matlab (mostrado en el Anexo), básicamente se encarga de clasificar los datos del fichero de texto recogidos asignando una variable a cada uno de los valores de las columnas de datos (código 9).

```
file = load('datosaut1.txt');
tiempo = file(:,1);
velocidad = file(:,2);
posx = file(:,3);
posy = file(:,4);
posz = file(:,5);
latitud = file(:,6);
```

```

longitud = file(:,7);
acelerometroX = file(:,8);
acelerometroY = file(:,9);
acelerometroZ = file(:,10);
giroscopioX = file(:,11);
giroscopioY = file(:,12);
giroscopioZ = file(:,13);
brujula = file(:,14);
aceleracion = file(:,15);
giro = file(:,16);
freno = file(:,17);

```

Código 9. Ejemplo de clasificación de datos recogidos en el fichero de texto

Posteriormente se representarán diferentes gráficas que permitirán conocer el funcionamiento del vehículo respecto a los datos obtenidos por los sensores. Un ejemplo de gráfica se muestra en el código 10, donde se representa la velocidad, la intensidad de la aceleración y la aceleración lineal en X recogida por el acelerómetro del sensor inercial.

```

% Velocidad, pedal aceleración y aceleración lineal en X
figure;
subplot(3,1,1);
plot(tiempo,velocidad);
title('Velocidad','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Velocidad (Km/h)','Interpreter','Latex','FontSize',16);
grid on;
subplot(3,1,2);
plot(tiempo,aceleracion,tiempo,freno)
title('Aceleracion','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Aceleraci\''on y freno
[0,1]','Interpreter','Latex','FontSize',16);
legend({'Aceleraci\''on',
'Freno'},'Interpreter','Latex','FontSize',14)
grid on;
subplot(3,1,3);
plot(tiempo,acelerometroX);
title('AcelerometroX','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Aceleraci\''on lineal X
(m/s^{2})','Interpreter','Latex','FontSize',16);
grid on;

```

Código 10. Ejemplo de creación de gráfica de datos recogidos

Los datos del radar necesitan una representación especial, donde las gráficas estarán formadas por dos ejes en los que se representarán los datos de azimuth y altitud. Además, en función de la profundidad y la velocidad hacia el radar se mostrarán los puntos de

diferente color. Con esto se obtiene una visión sobre los diferentes ángulos en los que el radar detecta obstáculos y la velocidad y distancia a la que se encuentran dichos obstáculos.

```
% Profundidad
figure;
scatter(x,y,[],z,'fill');
title('Profundidad (m)','Interpreter','Latex','FontSize',16);
xlabel('Azimuth (rad)','Interpreter','Latex','FontSize',16);
ylabel('Altitud (rad)','Interpreter','Latex','FontSize',16);
xlim([-0.2 0.2]);
ylim([-0.2 0.2]);
colorbar;

% Velocidad
figure;
scatter(x,y,[],v,'fill');
title('Velocidad (m/s)','Interpreter','Latex','FontSize',16);
xlabel('Azimuth (rad)','Interpreter','Latex','FontSize',16);
ylabel('Altitud (rad)','Interpreter','Latex','FontSize',16);
xlim([-0.2 0.2]);
ylim([-0.2 0.2]);
colorbar;
```

Código 11. Representación de los datos del radar

En cuanto a los datos obtenidos por el sensor de colisión, necesitan un tratamiento especial ya que solo se recogen datos cuando ocurre una colisión. Por ello, los datos no están en función del tiempo y habrá que adaptarlos. Para ello, siguiendo un video de la simulación realizado se aproxima el tiempo creando un vector con los mismos puntos que los cuatro datos recogidos por la simulación. Con los vectores de dimensiones similares, se pueden representar en Matlab obteniendo información de las diferentes colisiones ocurridas durante la simulación.

```
%% Colisiones
clc
clear all

filecollision = load('collisiondata.txt');
tiempo = 23:0.027:25
impulsox = filecollision(1:75,1);
impulsoy = filecollision(1:75,2);
impulsoz = filecollision(1:75,3);
intensidad = filecollision(1:75,4);

figure;
subplot(2,1,1)
plot(tiempo,impulsox,tiempo,impulsoy,tiempo,impulsoz);
title('Impulso','Interpreter','Latex','FontSize',20);
xlabel('Tiempo','Interpreter','Latex','FontSize',16);
ylabel('Impulso (kgm/s)','Interpreter','Latex','FontSize',16);
legend({'X', 'Y', 'Z'},'Interpreter','Latex','FontSize',14)
```

```
grid on;  
subplot(2,1,2)  
plot(tiempo,intensidad);  
title('Intensidad','Interpreter','Latex','FontSize',20);  
xlabel('Tiempo','Interpreter','Latex','FontSize',16);  
ylabel('Intensidad (kgm/s)','Interpreter','Latex','FontSize',16);  
legend({'X', 'Y', 'Z'},'Interpreter','Latex','FontSize',14)  
grid on;
```

Código 12. Ejemplo de representación de los datos recogidos por el sensor de colisión

Capítulo 5

Pruebas y validación

5.1. Introducción

En este capítulo se comentarán los resultados de las distintas simulaciones realizadas para estudiar el comportamiento del sistema. En concreto, se realizan dos tipos de pruebas: una simulación en tiempo real en la que se puede alternar entre el manejo manual y automático del vehículo, del que se tiene una respuesta instantánea pudiendo visualizar todos los datos obtenidos por los sensores al momento y una simulación en la que se capturarán y almacenarán datos para su posterior procesamiento. Para hacer unas simulaciones más realistas, se variarán las condiciones ambientales con el objetivo de comprobar que los sensores trabajan de forma correcta, independientemente del momento del día y de las condiciones climatológicas. Durante las simulaciones se someterá al vehículo a diferentes situaciones de tráfico realistas para verificar que se comporta de forma adecuada, respetando tanto las directrices de tráfico, como límites de velocidad, señales de stop, cedas el paso, etc. Además, se estudiarán diferentes casos en los que se produzcan colisiones o aparezcan obstáculos en la carretera. En cuanto a la captura de datos, se almacenarán datos de la mayoría de los sensores de los que dispone el vehículo y se procesarán en Matlab para visualizar dichos datos en diferentes gráficas explicativas y así entender el comportamiento sensorial del vehículo autónomo.

5.2. Simulación en tiempo real

En este caso, como se explicó en el capítulo anterior, se iniciará la sesión de juego (figura 55) para ejecutar el simulador en el motor de Unreal Engine. Con el simulador en funcionamiento, se utilizarán los diferentes programas creados en el módulo *PythonAPI* para las simulaciones de conducción. En este caso se utilizarán concretamente los programas *manual_control.py* (para la interacción con el vehículo) y el programa *spawn_npc.py* (para añadir vehículos y personas al mapa). En esta simulación los vehículos añadidos al mapa circularán automática y aleatoriamente a lo largo del mapa. Se pueden añadir tantos vehículos como se deseen, aunque inicialmente las pruebas se realizarán con 100 vehículos para evitar sobrecargas en el programa. Por otro lado, mediante *manual_control.py* aparecerá una pantalla en la que el usuario puede visualizar el vehículo y controlarlo o accionar el control manual, como se explicó en capítulos anteriores. En esta pantalla se puede visualizar gran cantidad de información a tiempo real que proporcionan diferentes sensores, así como diferentes tipos de cámaras. Esta prueba, se realizará completamente con el control automático activado y por tanto el vehículo circulará aleatoriamente, al no haberle asignado ningún destino concreto. Ante la imposibilidad de mostrar en la memoria archivos audiovisuales, las pruebas realizadas en la ampliación universitaria se ilustrarán mediante un conjunto de imágenes.

Como se puede observar en la figura 57, el vehículo circula por la rotonda situada entre la Escuela de Ingenierías Industriales y la Facultad de Estudios Sociales y del Trabajo, donde en todo momento se sitúa dentro de su carril y su velocidad se adecua a la situación del tráfico. En este caso se encuentra rodeado de vehículos, por lo que circula a velocidad moderada, en concreto, a 21 km/h como se observa en el panel lateral, y no realiza el cambio de carril hasta que pueda hacerlo con seguridad. Además, el panel lateral da numerosa información como la posición en el mapa, la marcha engranada, etc. También, muestra información del entorno, detectando los vehículos que tiene cercanos, así como la distancia respecto a éstos. Los vehículos más cercanos son un Lincoln a 3m, que es el que se encuentra a la derecha, el coche de policía a 7m y la bici que se observa en la imagen a 11m. Esta información resulta muy útil ya que se conoce la distancia a estos vehículos y el tipo de vehículo del que se trata.



Figura 57. Vehículo circulando en una rotonda de la ampliación

Otra de las situaciones a estudiar en el simulador es el comportamiento del resto de vehículos que circulan por el mapa. En la figura 58 se muestra una vista en primera persona cuando el vehículo circula por una rotonda y otro coche pretende incorporarse a ésta. Se puede ver como el vehículo que pretende incorporarse, detecta que un vehículo que circula por la rotonda se aproxima y por tanto se detiene para ceder el paso al vehículo que tiene preferencia. Por tanto, tras diversas pruebas de esta índole se ha determinado que tanto el vehículo con el que se circula y los que se encuentran en el resto del mapa circulan adecuadamente respetando las normas de circulación establecidas por medio de señales e indicaciones.



Figura 58. Incorporación a una rotonda

Otra de las situaciones interesantes es el comportamiento del vehículo autónomo ante un semáforo. En la figura 59 se presenta esta situación en la que el vehículo, al detectar el semáforo en rojo se detiene hasta que vuelve a ponerse en verde y pueda continuar su marcha. Este semáforo, situado en la llegada de la ampliación para permitir el paso del metro, es el único en todo el mapa por lo que resulta de especial interés el comportamiento de los vehículos ante éste. El vehículo circula a 26 km/h cuando se acerca al semáforo, que se encuentra en verde. Al acercarse, este cambia a color rojo y el vehículo se detiene completamente como se puede ver en la imagen con una velocidad de 0 km/h.

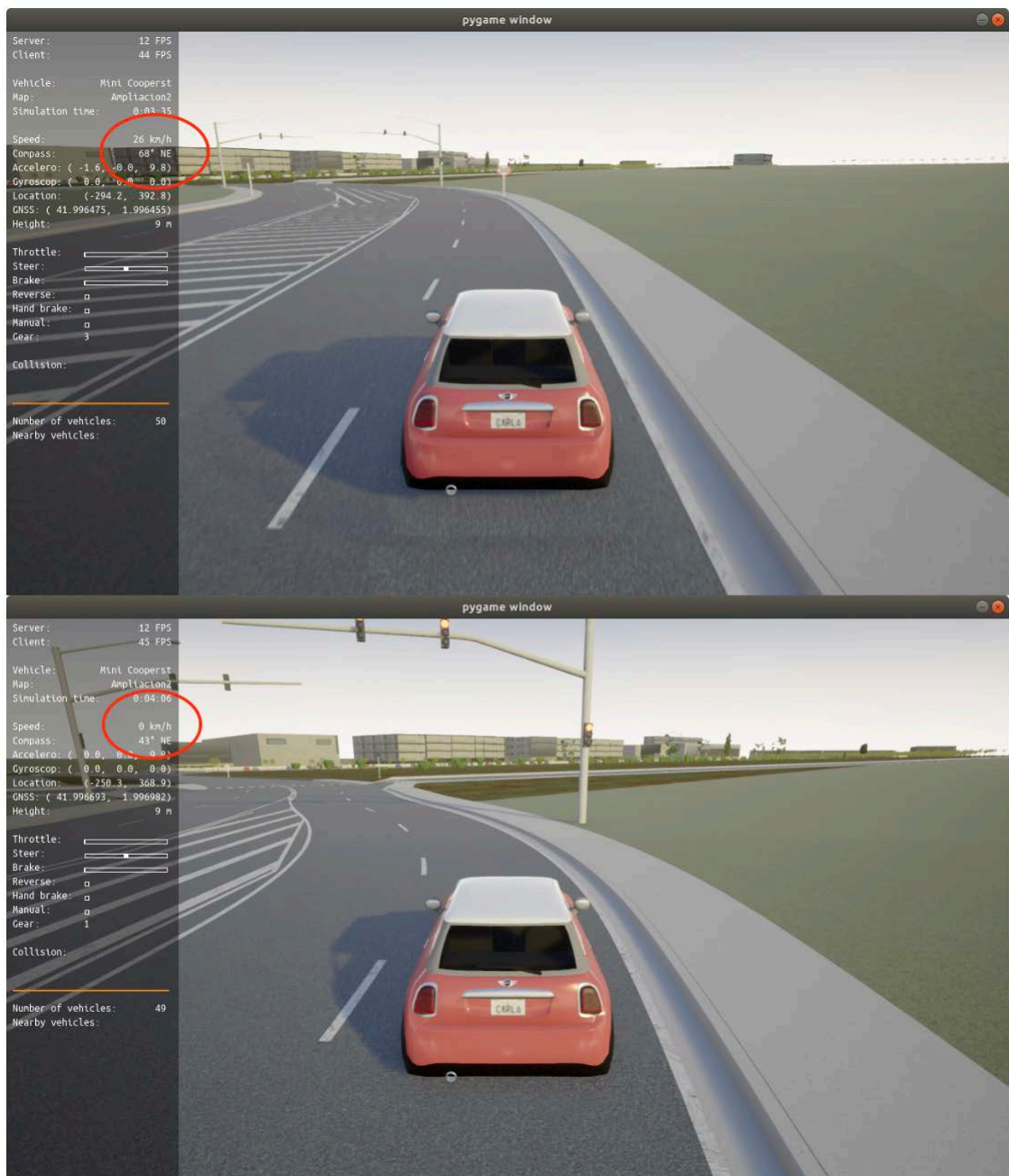


Figura 59. Comportamiento del vehículo ante un semáforo

Es interesante estudiar como el vehículo adapta su velocidad dependiendo de la situación del tráfico en la que se encuentra. En caso de que circulen vehículos delante el vehículo autónomo reduce a una velocidad segura para circular detrás del otro vehículo sin que se produzca una colisión. Sin embargo, en situaciones de tráfico fluido en la que no existen obstáculos el vehículo adapta su velocidad siguiendo las señales de limitación de velocidad situadas en la carretera. En la figura 60 se muestra esta situación en la que

el vehículo circula a 42 km/h, que es una velocidad próxima a la velocidad máxima permitida para zonas urbanas (50 km/h). Como se puede ver en la figura 60 a) el vehículo detecta una señal que indica que la velocidad mínima permitida es 90 km/h y como no hay tráfico delante el vehículo acelera hasta los 68 km/h (figura 60 b)). Cabe destacar que esta zona corresponde a la carretera situada en la parte trasera de la Escuela de Ingenierías Industriales, cuyo límite de velocidad en la realidad es 50 km/h. Se ha añadido un límite superior en esta zona con fines de estudio de la capacidad de adaptación de la velocidad del vehículo, ya que el límite de velocidad en prácticamente la totalidad del entorno es de 50 km/h o inferior, al tratarse de zona urbana. En las figuras 60 c) y d) se muestra la situación en la que el vehículo va llegando a la rotonda y la velocidad máxima pasa a ser 30, por lo que se reduce la velocidad a 23 km/h para entrar con seguridad a la intersección circular.

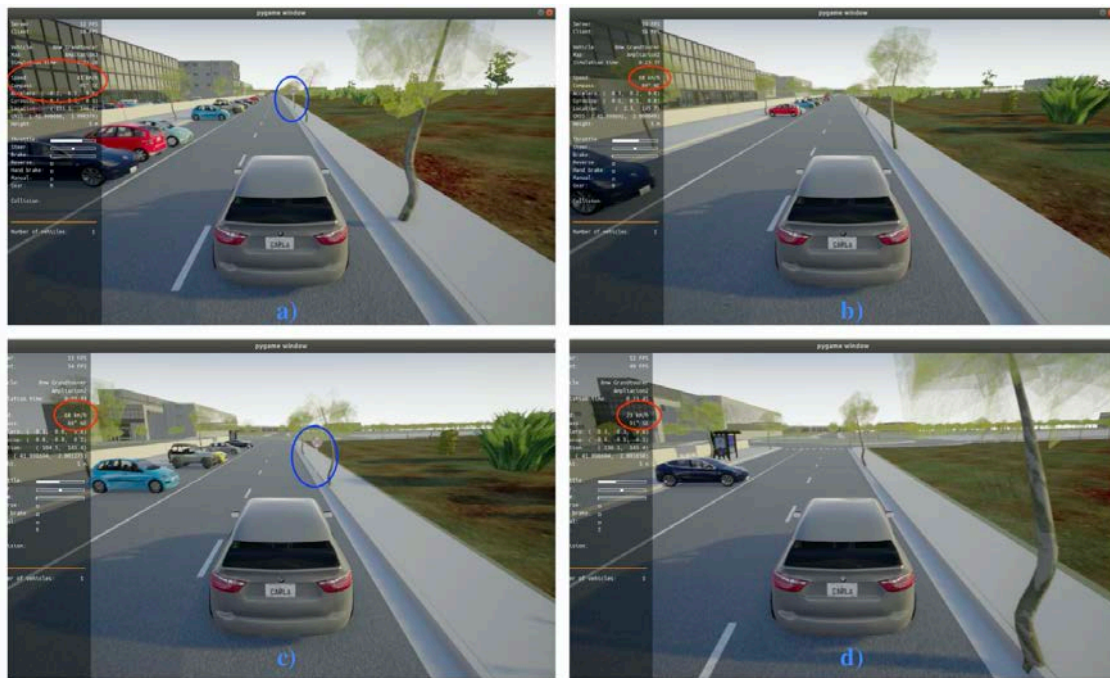


Figura 60. Adaptación de la velocidad del vehículo sin obstáculos

Es importante observar el comportamiento del vehículo ante la presencia de un obstáculo en la carretera como puede ser una persona o un vehículo en doble fila. Estas dos situaciones concretas se han simulado, ver figura 61, para estudiar si el vehículo sigue por su carril ocasionando una colisión, cambia de carril o se detiene. En la primera prueba realizada, donde hay un vehículo averiado en el carril derecho de la recta frente a la

Escuela de Ingenierías Industriales se observa como el vehículo al detectar la presencia de un obstáculo, disminuye su velocidad y cambia al carril izquierdo para proseguir su marcha, ver figuras 61 a) y c). El vehículo que circulaba a velocidades cercanas a 30 km/h disminuye su velocidad hasta 8 km/h para poder realizar la maniobra sin peligro. Por otro lado, en la salida del parking este de la Escuela de Ingenierías se ha simulado el paso de dos personas por un paso de peatones. En este caso, el vehículo se detiene completamente antes de llegar al paso de peatones para evitar un atropello (figuras 61 b) y d)). Por tanto, se puede concluir que ante obstáculos los sensores son capaces de detectarlos con antelación y actuando correctamente para evitar posibles accidentes.

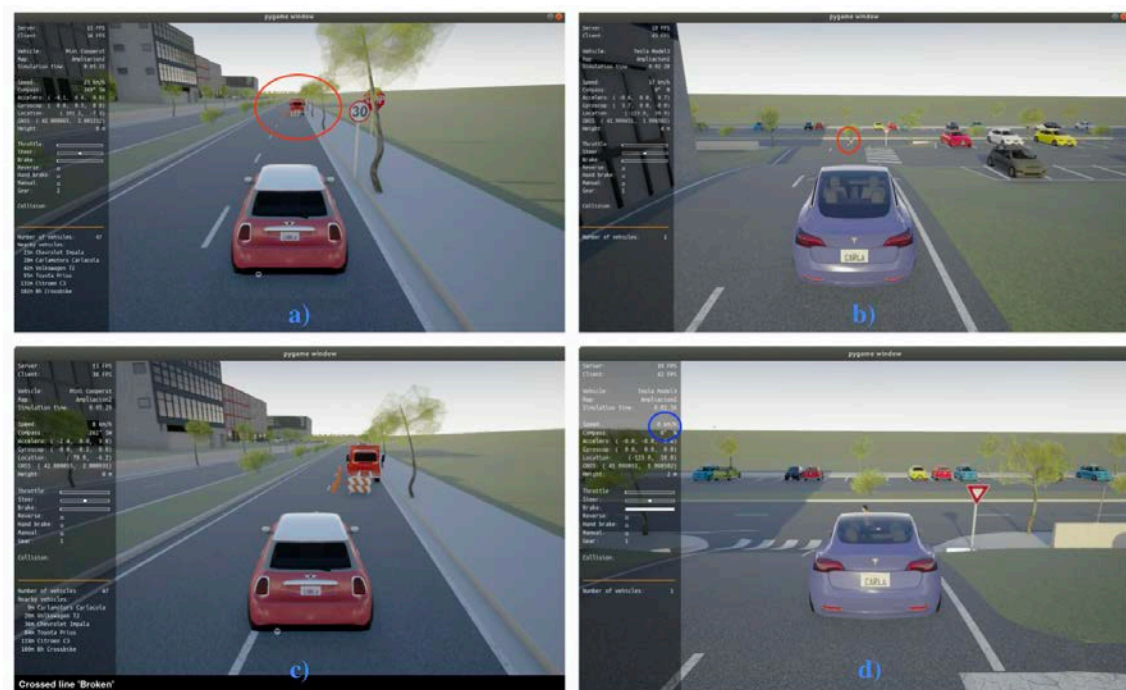


Figura 61. Comportamiento ante obstáculos en la carretera

Otra de las situaciones a estudiar, era el comportamiento del vehículo ante cambios climatológicos y frente a diferentes condiciones de luz. En la figura 62, se presentan diferentes capturas de las diferentes condiciones atmosféricas estudiadas desde la conducción con lluvia intensa a una situación con el cielo completamente despejado y con mucha luz. Los diferentes ambientes que se pueden configurar en la simulación son soleado de día con cielo despejado, puesta de sol con cielo despejado, noche nublada, puesta de sol nublada, lluvia fuerte por la noche, lluvia fuerte con puesta de sol, lluvia

media con puesta de sol, lluvia media por la noche, llovizna por la noche, llovizna con puesta de sol, nublado húmedo por la noche, nublado húmedo con puesta de sol, atardecer con ambiente húmedo, noche con ambiente húmedo, noche despejada y atardecer despejado.

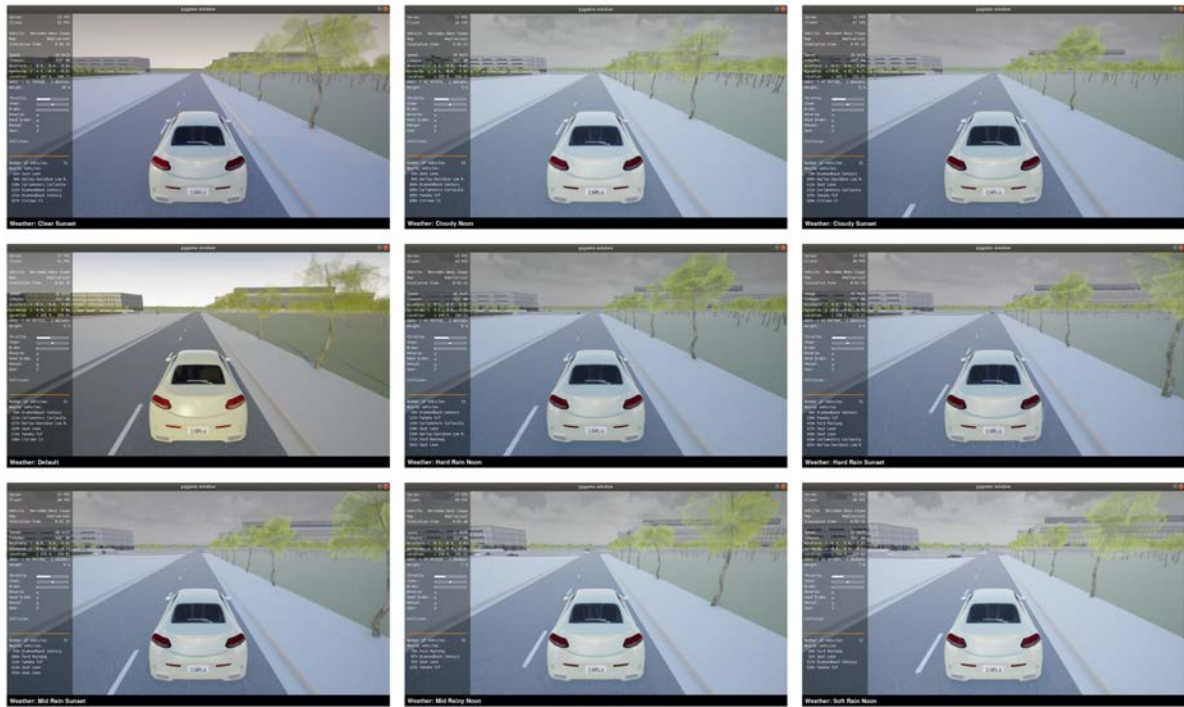


Figura 62. Comportamiento del vehículo ante cambios climatológicos

Tras las diferentes pruebas, el comportamiento del vehículo es similar en las diferentes condiciones meteorológicas estudiadas. En cuanto al comportamiento de los sensores, la mayoría no se ven afectados por las condiciones externas como se observa en la figura 63. En este caso, se ha comparado dos situaciones extremas, un día soleado y una noche con lluvia intensa. Con la cámara RGB se observa como la visión disminuye como consecuencia de la falta de luminosidad en el ambiente y la lluvia. Sin embargo, las imágenes capturadas por la cámara de profundidad son similares en ambos casos, por lo que no se ve afectada por el clima. Algo similar ocurre con el LIDAR, donde en la misma situación que se observa en las cámaras, se detectan los mismos elementos y con un nivel de detalle similar. Por tanto, en este punto CARLA debe mejorar ya que en la realidad los diferentes sensores se ven afectados en mayor medida por los cambios en la luminosidad

exterior, algo que no ocurre en CARLA, ya que salvo con la cámara RGB, los datos de los sensores son similares en las distintas condiciones estudiadas. En este hecho también puede influir que, aunque CARLA permita variar los momentos del día y las condiciones climatológicas, los cambios entre estas no son tan grandes, observando cómo, por ejemplo, en el caso de conducción nocturna no hay un cambio brusco de luminosidad y por tanto, los sensores no detectan esa pequeña variación.

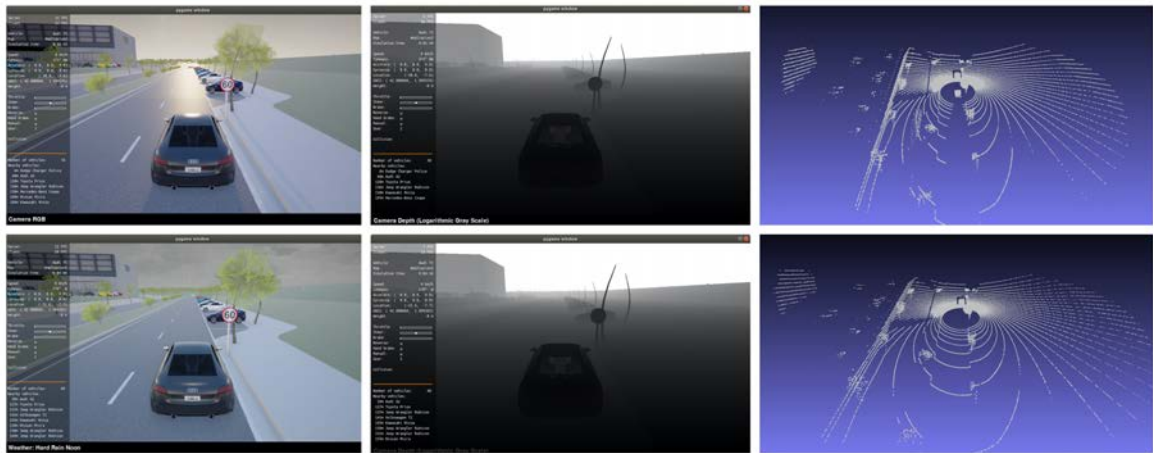


Figura 63. Comportamiento de la cámara y el LIDAR en diferentes condiciones climatológicas

Una herramienta muy útil de la que dispone el simulador es la visualización de diferentes sensores de los que dispone el vehículo autónomo. Esto, permite hacerse una idea de la información recogida y de cómo actúa el vehículo dependiendo del entorno en el que se encuentra. En la figura 64 se observan las diferentes cámaras disponibles. La cámara RGB (figura 64 a)) recoge información con el modelo cromático universal que permite visualizar los colores tal como lo veríamos en la realidad, diferenciando los colores de los vehículos del entorno y de las diferentes estructuras. La siguiente cámara es la de profundidad en formato RAW, ver figura 64 b), (contiene todos los datos de la imagen tal y como son captadas por la cámara), escala de grises (figura 64 c)) y escala de grises logarítmica (figura 64 d)), y representa la profundidad de los elementos en el campo de visión del mapa. Esta cámara es interesante para calcular la distancia a la que se encuentran otros vehículos y estructuras, como se puede ver en la cámara de profundidad (RAW) y cámara de profundidad (escala de grises logarítmica). Con estas dos cámaras se distinguen claramente el resto de los elementos que rodean al vehículo, sin embargo, la

cámara en escala de grises sin el procesamiento logarítmico no es capaz de detectar otros elementos obteniendo una imagen negra en la que lo único que permite diferenciar es el cielo de otras estructuras. El siguiente sensor, se denomina cámara de segmentación semántica (*CityScapes Palette*), ver figura 64 e). Este sensor no es una cámara real, sino que es una idealización del conjunto que forman la cámara y un algoritmo de segmentación. El algoritmo asigna a cada elemento del mapa un color determinado para detectar de qué se trata. Se observa como los coches se clasifican en azul, las señales de tráfico en amarillo y las construcciones (Faculta de Ciencias de la Salud en este caso) se clasifican como gris. Además, al fondo se observa una parada de autobús en rosa claro. Este sensor, realmente no proporciona información realista para la navegación del vehículo, sin embargo, puede ser muy útil para otras tareas, como por ejemplo, proporcionar el *ground truth* con el que entrenar redes neuronales para la clasificación o segmentación de imágenes. Hay otra cámara de segmentación en formato RAW que no es capaz de detectar ningún elemento ya que necesita procesar la imagen asignando colores a los diferentes elementos del mapa como se ha comentado anteriormente. La última cámara es la RGB distorsionada en la que no existen grandes diferencias con la RGB normal, apreciando imágenes prácticamente idénticas. También, se puede obtener una imagen visual del funcionamiento del LIDAR (figura 64 g)), en la que se observa la nube de puntos generada y que se utilizará posteriormente para obtener un mapa del entorno en 3D. En este caso no se observa gran cosa ya que para visualizar la nube de puntos hay que capturar los datos y representarlos posteriormente. En la figura 63 y en las diferentes figuras del apartado 5.3.3 se observan diferentes capturas de las nubes de puntos creadas por el LIDAR en diferentes situaciones utilizando el programa MeshLab. Este sensor, aunque pueda parecer que proporciona escasa información, es muy importante y valioso, ya que ofrece medidas de distancias en el espacio a elementos del espacio con mucha rapidez, con la cual se puede detectar infraestructuras, obstáculos, otros vehículos, de forma mucho más rápida a como se podría hacer a partir de imágenes, las cuales necesitan de un procesamiento previo muy costoso para extraer información tridimensional.

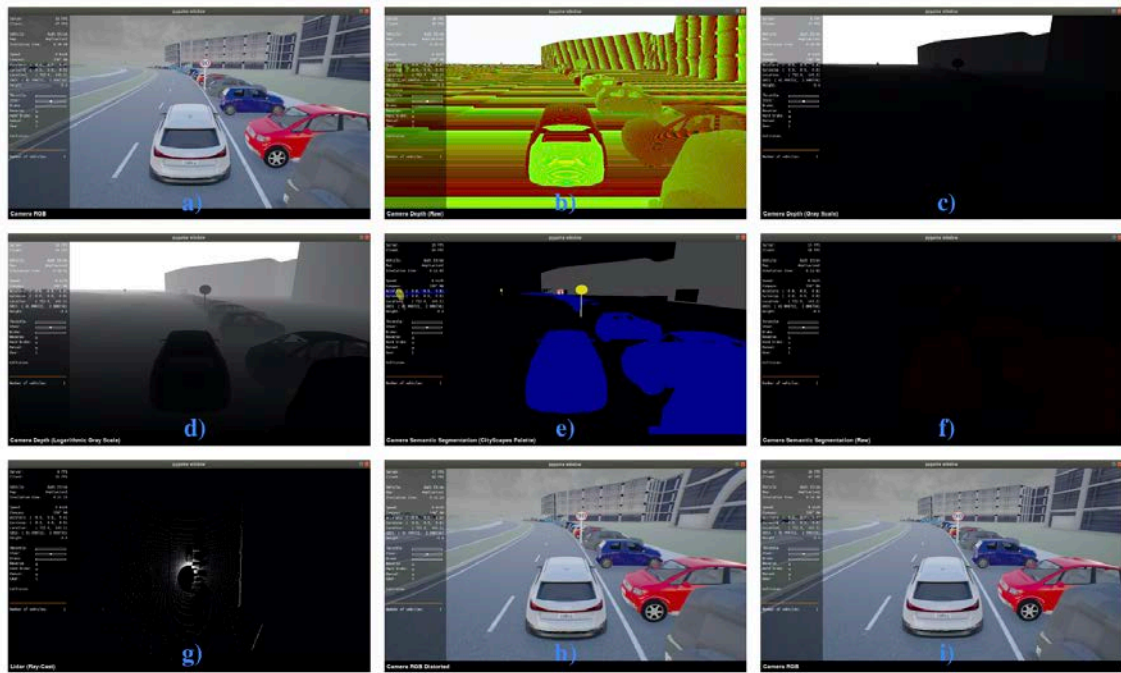


Figura 64. Cámaras disponibles en el vehículo autónomo

Otro de los sensores de los que se puede obtener información visual durante la simulación es el radar. Como se explicó en los capítulos iniciales, el radar emite señales electromagnéticas y calculando el tiempo de emisión/recepción de la señal es capaz de obtener información sobre la distancia a la que se encuentran los objetos respecto al vehículo. En la simulación, se muestran la visualización del radar donde se observa una serie de puntos de distintos colores. Cuando no existe velocidad, los puntos son blancos, ver figura 65 a), mientras que a medida que se aumenta la velocidad del vehículo los puntos se convierten en rojo (figura 65 b)), lo que indica velocidad negativa. Mientras mayor sea la velocidad del vehículo los puntos van cambiando de color hacia color rojo intenso (figura 65 c)), lo que indica que está aumentando la velocidad negativa y que por tanto se alcanzarán en menos tiempo dichos objetos. En situaciones en las que por ejemplo un vehículo que circula delante va a una velocidad inferior a la del propio vehículo sigue existiendo velocidad negativa y los puntos siguen siendo rojos. Sin embargo, si el vehículo de delante circula a mayor velocidad existe velocidad positiva y los puntos son azules como se muestran en la figura 65 d). Por tanto, los puntos blancos y azules indican que no existe peligro de alcance, mientras que los rojos indican que si puede existir ese peligro en caso de que se dirigiese hacia el objeto detectado a la velocidad a la que se está circulando en ese momento.

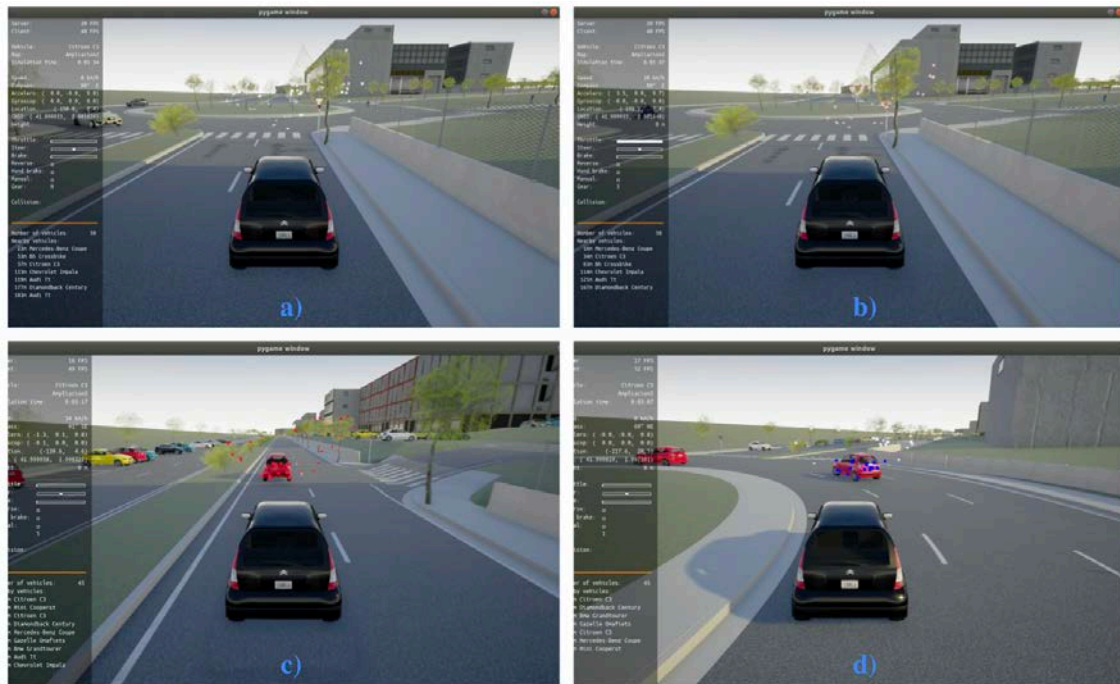


Figura 65. Imágenes del radar durante la simulación

También, utilizando el control manual se ha estudiado que ocurre cuando existe una colisión. En este caso, el control automático no se ha podido utilizar ya que en raras ocasiones se producen colisiones con este control. En la figura 66 se muestra la situación en la que se ha simulado una colisión con otro vehículo del mapa. En primer lugar, se puede observar como el detector de líneas ha detectado que se ha cruzado una línea mediante el mensaje “Crossed line Broken” (figura 66 a)). Por otro lado, en la figura 66 b) se puede ver como en el momento de la colisión salta un mensaje que indica que se ha producido una colisión y el vehículo con el que ha ocurrido el incidente. Además, en el panel lateral hay una gráfica en la que se recogen las colisiones observando un pico con la intensidad y el momento en el que ha ocurrido el choque.

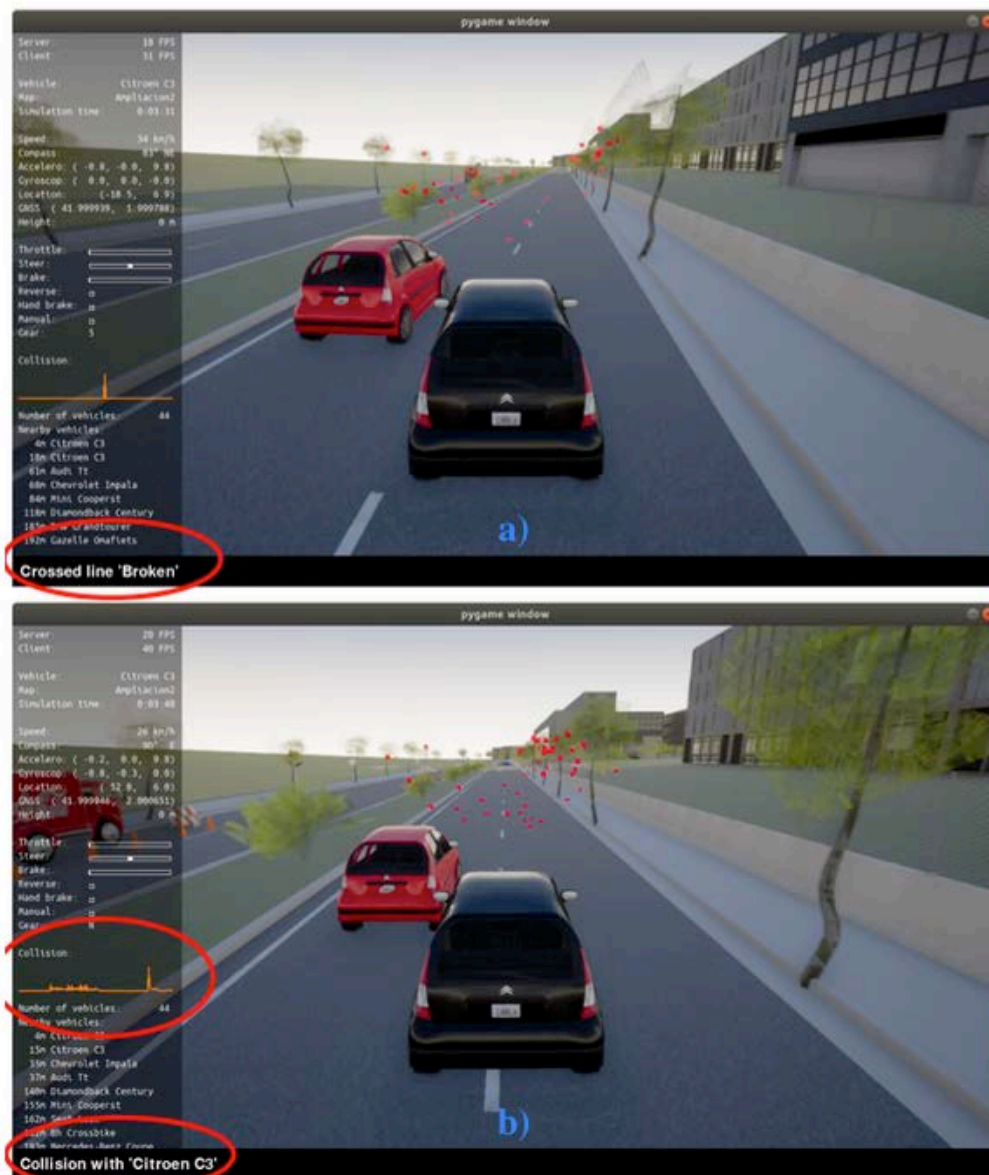


Figura 66. Simulación de una colisión

En la simulación se ha visto como el vehículo tiene un muy buen comportamiento respetando todas las señales de tráfico e independientemente de las condiciones ambientales y del entorno. El vehículo, adapta su comportamiento recogiendo gran cantidad de información del entorno y de otros vehículos que circulan por la ampliación circulando en todo momento de forma correcta y sin producirse colisiones con otros objetos o vehículos.

5.3. Estudio de los datos sensoriales

Para el almacenamiento y representación de los datos sensoriales se han realizado una serie de pruebas de la misma forma que la mencionada en el apartado anterior. Concretamente, se ha realizado una prueba en conducción autónoma en las que el vehículo ha circulado aleatoriamente por el entorno de la ampliación y una prueba en conducción manual. Esta segunda prueba se ha realizado ya que en modo autónomo rara vez se producen colisiones con otros vehículos u obstáculos, cambios bruscos de aceleración, etc. Con esta simulación con conducción manual, se ponen a prueba todos los sensores y se compararán los resultados con los obtenidos mediante conducción autónoma.

5.3.1. Prueba 1

En esta primera prueba, se ha realizado un pequeño y simple recorrido en el que no existen grandes variaciones. El recorrido, mostrado en la figura 67, que inicia en el punto marcado con un cuadrado y finaliza en el punto marcado con una X, consistirá en llegar a la rotonda de la Facultad de Ciencias de la Salud, dar una vuelta completa a la rotonda y continuar recto por el camino que llevaba el vehículo antes de entrar en la intersección circular.



Figura 67. Recorrido de la primera simulación en conducción autónoma

En primer lugar, gracias a los datos del sensor de localización y el sensor GNSS se puede comprobar el recorrido que ha seguido el vehículo. El sensor de localización recoge los datos de posición del vehículo del sistema de coordenadas de CARLA, mientras que el sensor GNSS obtiene datos de las coordenadas geográficas en las que se encuentra el vehículo en todo momento. Estas coordenadas no se corresponden con las coordenadas del mundo real en las que se situaría el entorno de la ampliación del Campus de Teatinos, pero con la variación de la posición GNSS y de las coordenadas de CARLA se puede obtener el recorrido realizado. El recorrido, recogido por los sensores (figura 68), es similar al indicado en la figura 67. El camino obtenido con las diferentes posiciones del vehículo durante la simulación coincide con el camino obtenido con los datos de latitud y longitud. En la figura 69 se puede observar el recorrido en 3D, en el que se observa como el trayecto comienza a 0m y aumenta hasta alcanzar algo menos de 4m de altitud.

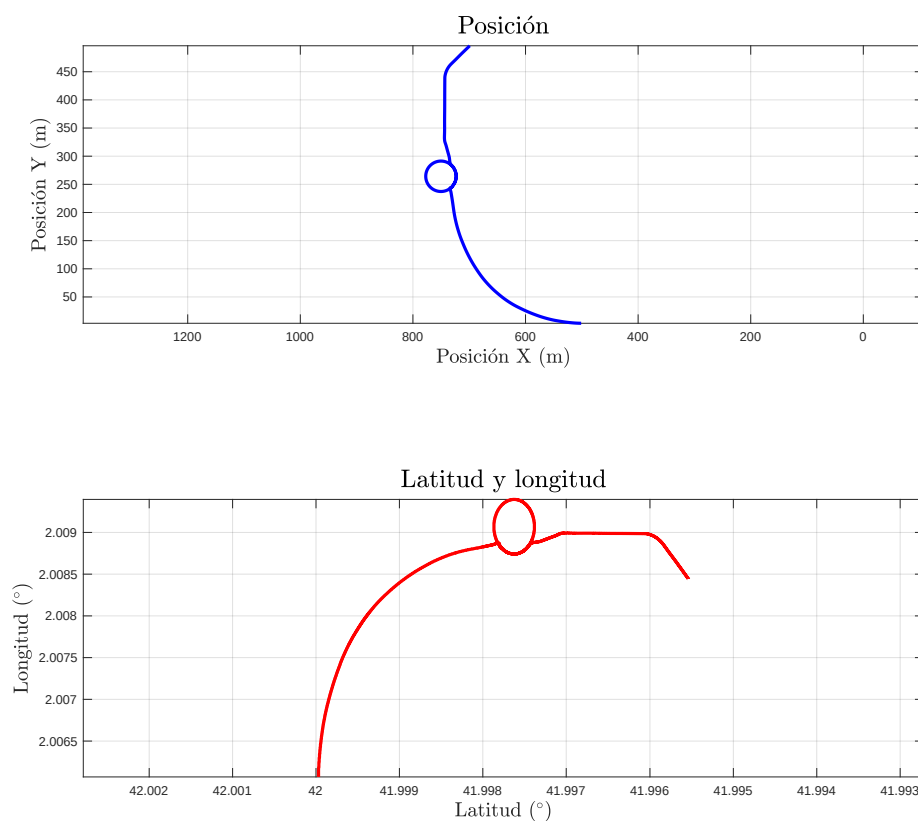


Figura 68. Recorrido del vehículo obtenido mediante la localización de CARLA (arriba) y las coordenadas geográficas

Recorrido 3D

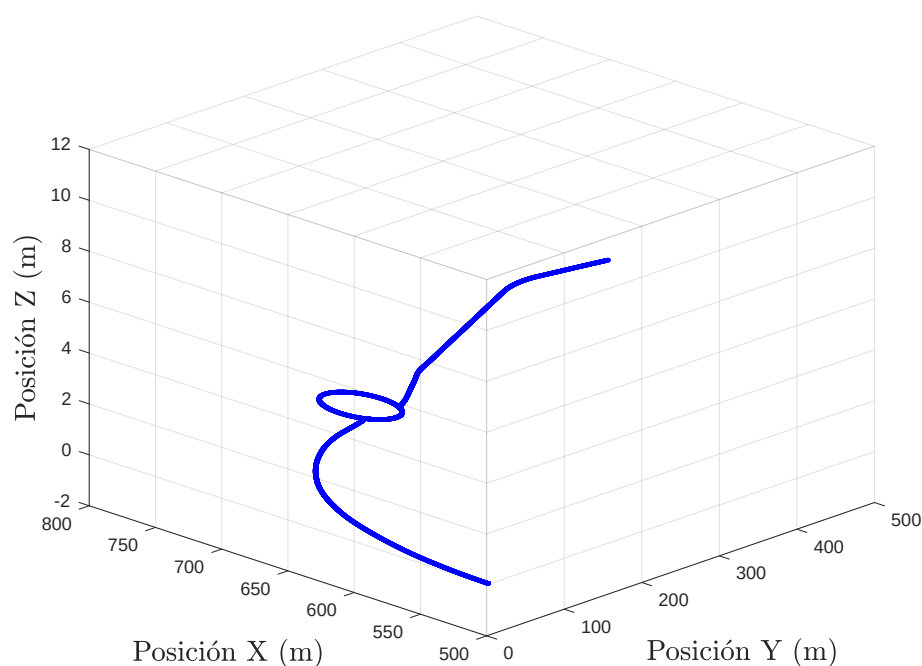


Figura 69. Recorrido de la simulación automática en 3D

En la figura 70 se representan la velocidad, la intensidad sobre los pedales de aceleración y freno y la aceleración lineal en X recogida por el acelerómetro. Inicialmente, se observa un periodo de tiempo en el que los sensores no detectan movimiento, el cual se corresponde al tiempo que tarda el simulador en iniciar. En el segundo 20 de simulación, aproximadamente, comienza a moverse el vehículo, aplicando una presión en el pedal de aceleración de 0.7 sobre 1, lo que lleva al vehículo a aumentar la velocidad hasta los 21 km/h, manteniéndose constante en esta velocidad hasta llegar a la rotonda. Debido a este aumento en la aceleración, el sensor inercial detecta una aceleración lineal en el eje X como se muestra en la gráfica inferior. En el segundo 50, momento en el que se detecta una señal en el que el límite de velocidad aumenta a 50 km/h (figura 71), el vehículo aumenta su velocidad aplicando mayor presión al pedal de aceleración, apareciendo un aumento en la aceleración lineal. Al llegar a la rotonda, el vehículo frena como se observa con el aumento de la intensidad de pedal de freno (gráfica naranja en la figura central), disminuyendo la velocidad y obtenido aceleraciones lineales negativas con oscilaciones hasta alcanzar una velocidad estable. El resto del recorrido la velocidad se mantiene estable sin grandes cambios detectados.

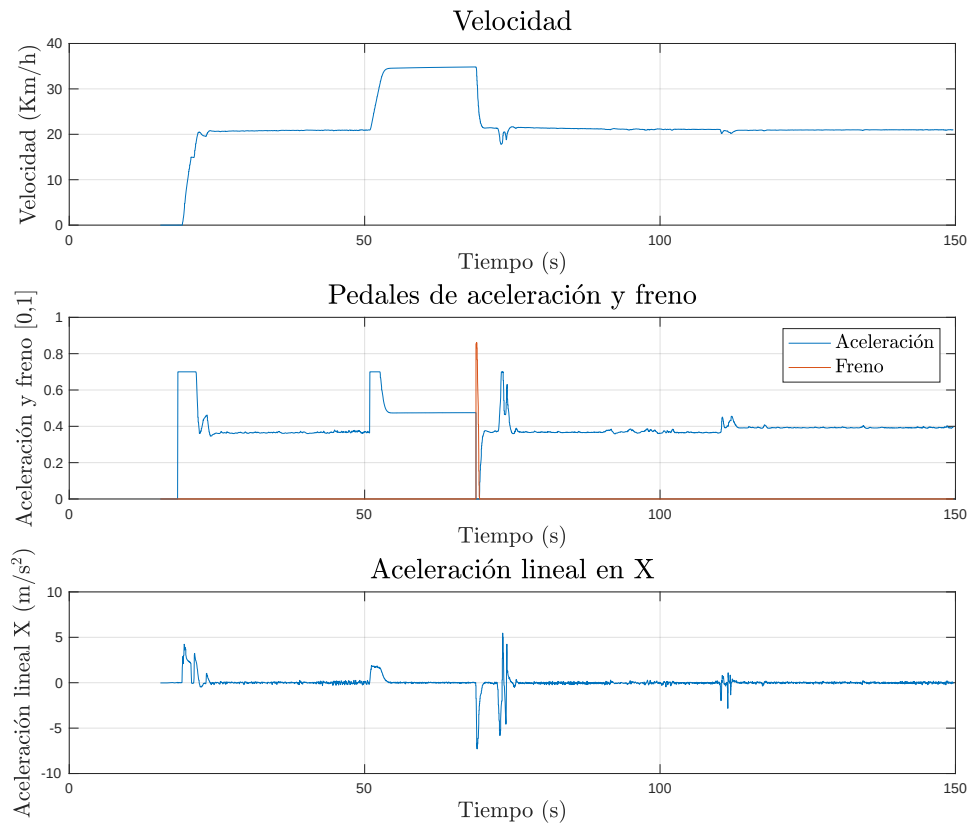


Figura 70. Velocidad (arriba), pedales de aceleración y freno (centro), aceleración lineal en X (abajo)



Figura 71. Momento en el que el vehículo detecta una señal de límite 50 km/h

En los primeros segundos de caída del vehículo, el acelerómetro detecta una caída de aceleración en el eje Z, por lo que para la representación de la figura 72 se han eliminado

los segundos iniciales. En este caso se observa como el acelerómetro, detecta cambios de aceleración en el eje X en los momentos de aceleración y desaceleración y en el eje Y cuando el vehículo gira. Sin embargo, la aceleración en el eje Z se mantiene constante prácticamente en todo momento al haber eliminado los momentos de caída iniciales. El giroscopio, detecta movimientos rotacionales sobre el eje vertical en los momentos en los que el vehículo gira como se observa en las dos gráficas inferiores de la figura 72. En torno al segundo 75 se detecta una gran oscilación que se corresponde con el momento de entrada en la rotonda (figura 73). El vehículo entra a una velocidad inadecuada a la rotonda por lo que necesita varios giros para enderezar la marcha y continuar dentro del carril. A partir de ese momento se mantiene constante un giro a la izquierda detectado como un movimiento rotacional constante negativo, que se mantiene en la vuelta completa a la rotonda. Al salir de ella, en el segundo 110 aproximadamente, se estabiliza el movimiento rotacional detectado por el giroscopio hasta el final de la simulación. Durante la vuelta a la rotonda el acelerómetro detecta aceleraciones lineales negativas en el eje transversal del vehículo (eje y), que se corresponde con el giro a la izquierda constante.

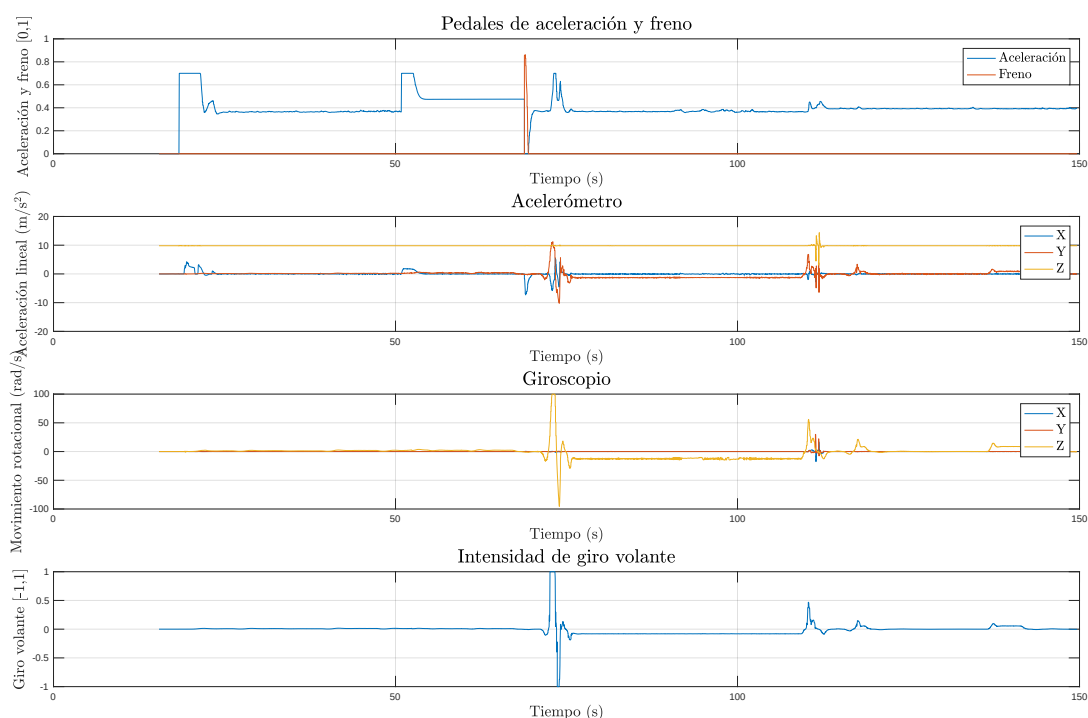


Figura 72. Intensidad pedales de aceleración y freno (grafica 1), aceleración lineal (gráfica 2), movimiento rotacional (gráfica 3) e intensidad de giro (gráfica 4)

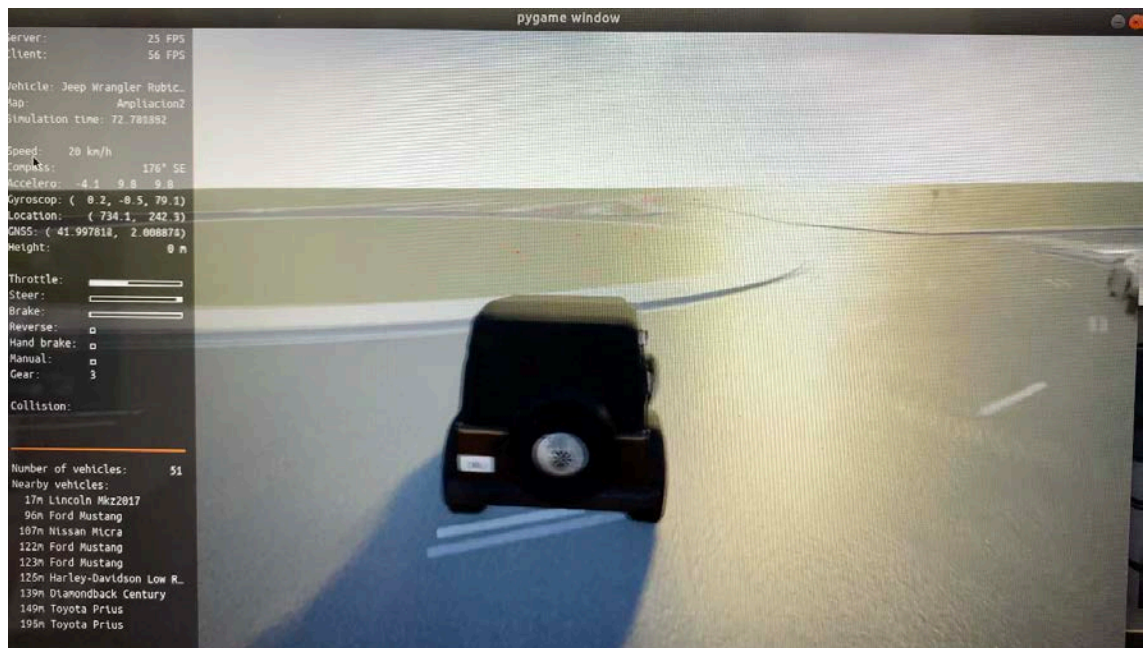


Figura 73. Momento en el que el vehículo entra en la rotonda

El siguiente sensor del que se pueden obtener datos es el radar. En este caso, los datos del radar no dan mucha información ya que durante la simulación el vehículo se encuentra con pocos coches, aunque se puede obtener una visión rápida del funcionamiento del radar. En la figura 74, se representa la orientación sobre la superficie (azimut) respecto al ángulo de altitud. Además, dependiendo de la profundidad (distancia del obstáculo hacia el sensor) varía el color de los puntos representados. Se puede ver como los ángulos son constantes durante la mayor parte de la simulación, sin embargo, se detecta un aumento de los puntos detectados en la zona izquierda ya que el radar detecta coches aparcados a una profundidad de entre 70 m y 90 m (como se puede ver por los diferentes colores de la barra de la figura 75).

En la figura 76 se representan los datos del radar, pero con gradiente de velocidad, en lugar de profundidad. En este caso, la velocidad hacia el sensor es constante la mayoría del tiempo al no encontrarse apenas obstáculos, sin embargo, se detecta un aumento de la velocidad hacia valores negativos por la zona de la izquierda que corresponde con el vehículo detectado, representado en la figura 77. El aumento de la velocidad negativa indica que el vehículo detectado circula a mayor velocidad y se aleja progresivamente. Estos valores se detectan en un verde más intenso en la gráfica.

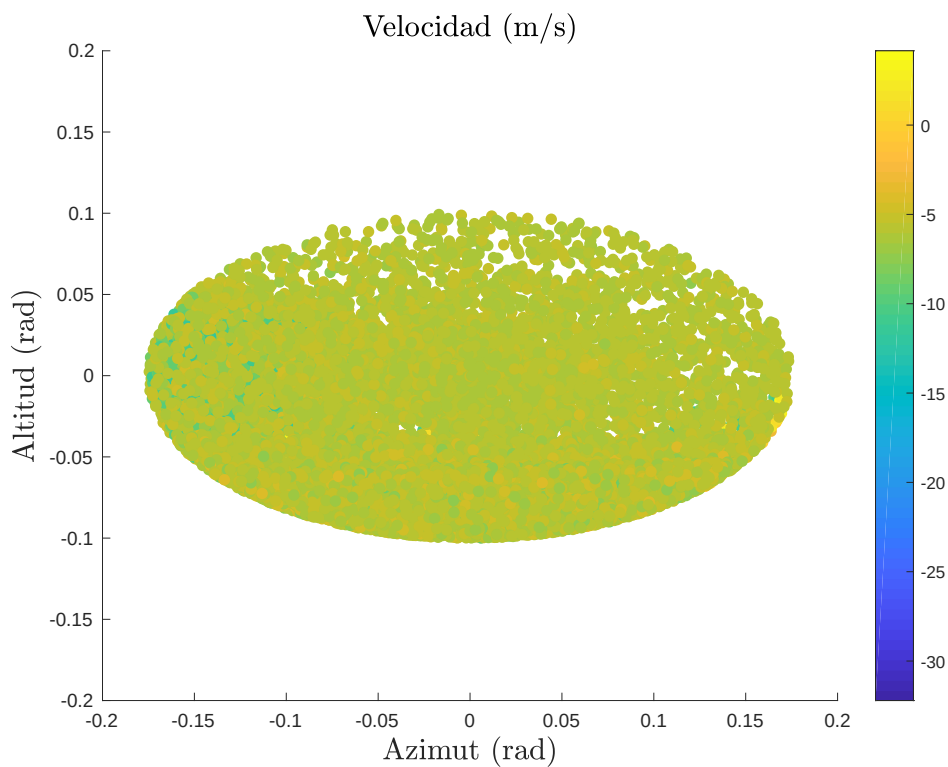


Figura 76. Azimut frente a altitud con gradiente de velocidad hacia el radar

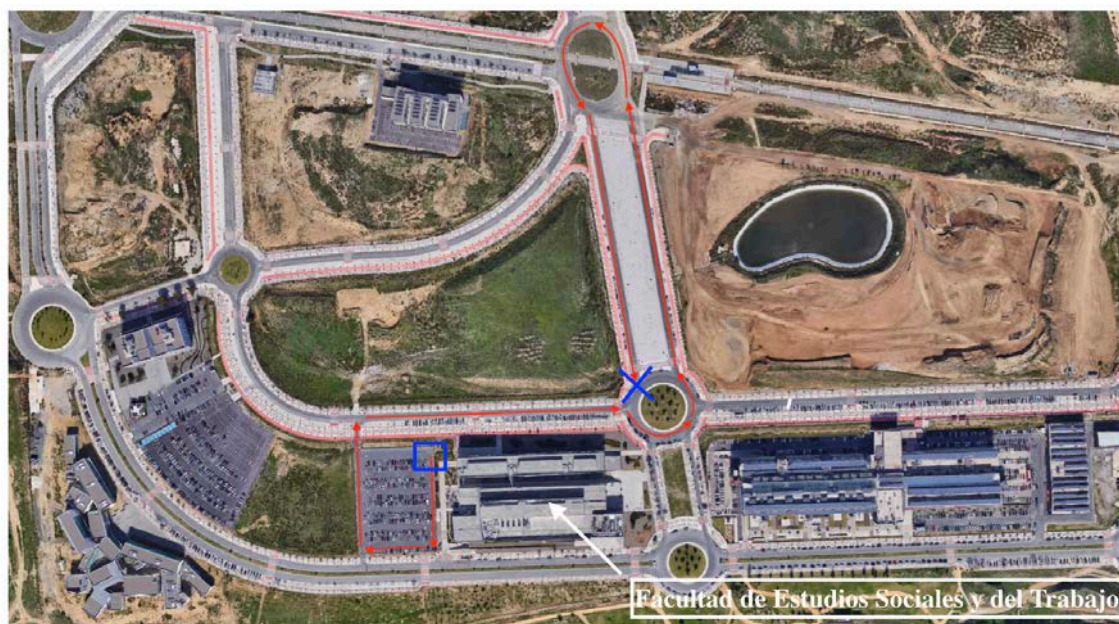


Figura 78. Recorrido en la prueba de conducción manual

En las figuras 79 y 80 se muestran los recorridos seguidos por el vehículo en 2D y 3D. En las gráficas 2D se obtiene el recorrido mediante la localización de CARLA y mediante las coordenadas geográficas, observando que ambos recorridos coinciden exactamente. En comparación con la simulación automática detallada en el apartado anterior, se observa como el recorrido no es tan lineal en este caso debido a la dificultad de mantener una trayectoria perfecta utilizando el modo manual.

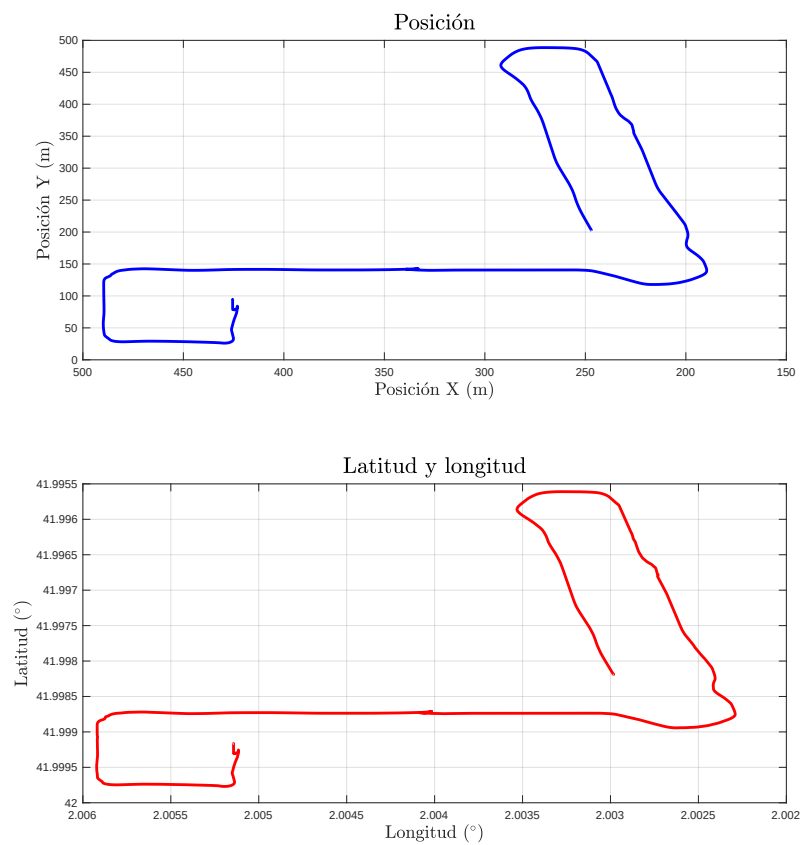


Figura 79. Recorridos de la prueba de simulación manual a partir de la localización (arriba) y las coordenadas geográficas (abajo)

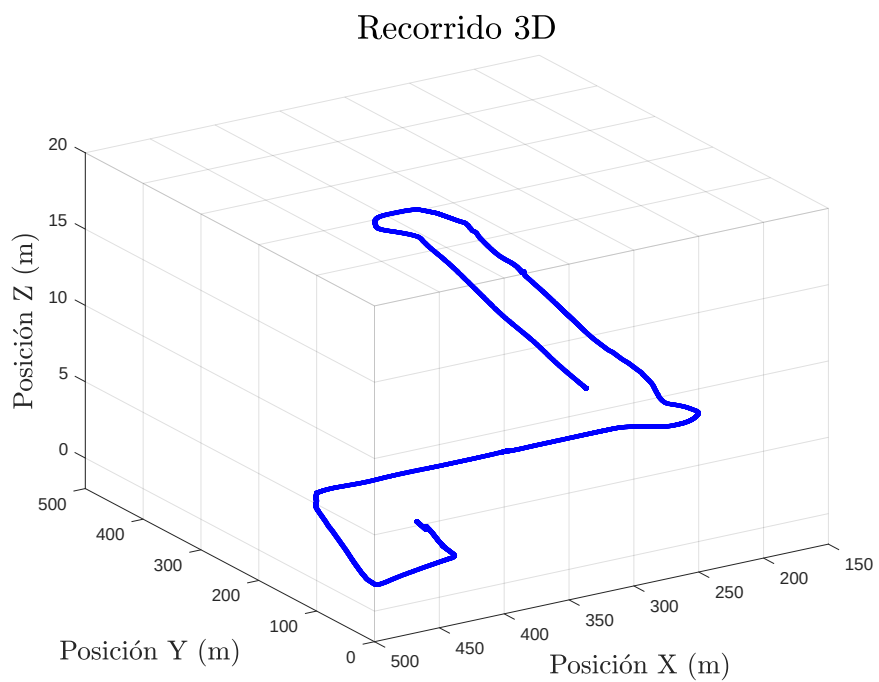


Figura 80. Recorrido de la simulación manual en 3D

En la figura 81 se representa la velocidad, intensidad de los pedales y aceleración lineal en X. En este caso la velocidad de los pedales es 0 o 1 debido a la imposibilidad de regular la intensidad, de ahí los cambios bruscos en la velocidad y las grandes oscilaciones detectadas por el acelerómetro en el eje longitudinal.

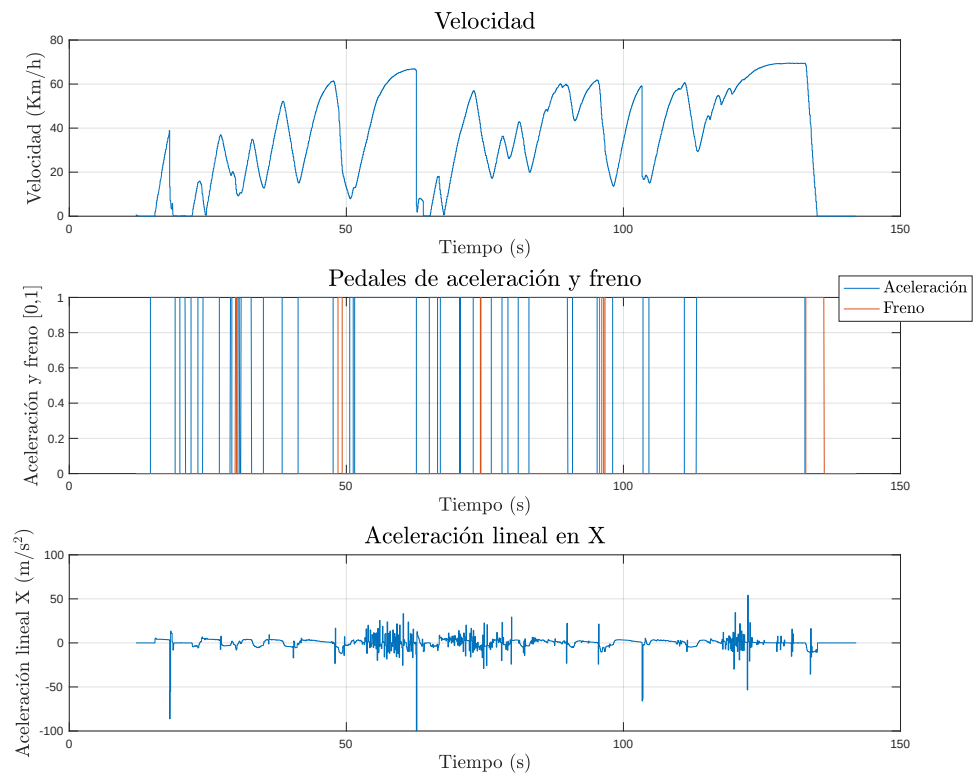


Figura 81. Velocidad (arriba), pedales de aceleración y freno (centro), aceleración lineal en X (abajo) en la prueba de conducción manual

En la gráfica 82 se muestran la intensidad de los pedales y el volante, así como los datos del sensor inercial. Los datos siguen teniendo un comportamiento oscilatorio como consecuencia de la brusquedad en el control de los mandos del vehículo. De las gráficas destacan como el movimiento rotacional en el eje z detectado por el giroscopio que tiene una gráfica similar a la intensidad de giro del volante y los cambios bruscos de aceleración negativa detectado por el acelerómetro como consecuencia de colisiones. Uno de estos casos se observa en el segundo 60 de simulación, aproximadamente, donde la aceleración lineal en X cae bruscamente como consecuencia de la colisión mostrada en la figura 83. En la gráfica de la velocidad de la figura 81 se observa este hecho donde la velocidad cae súbitamente desde 70 km/h hasta 0 km/h.

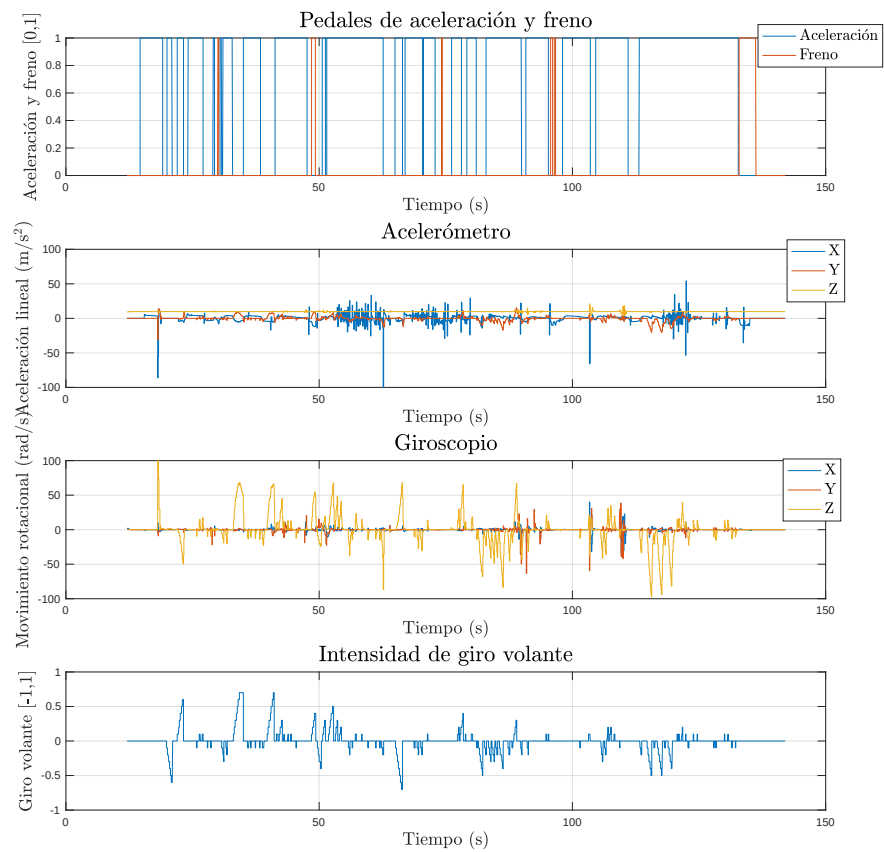


Figura 82. Intensidad pedales de aceleración y freno (gráfica 1), aceleración lineal (gráfica 2), movimiento rotacional (gráfica 3) e intensidad de giro (gráfica 4) en la prueba de simulación manual

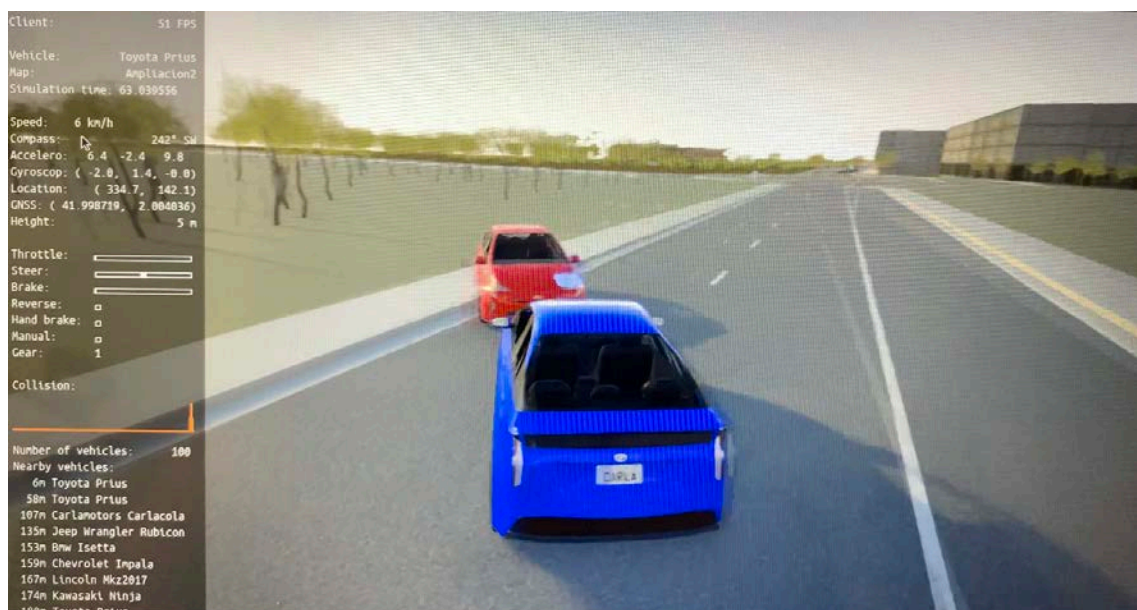


Figura 83. Momento de la colisión en el segundo 62 de la simulación de conducción manual

En cuanto a los datos obtenidos por el radar, en comparación con los obtenidos en la prueba de conducción autónoma, existen mayores cambios en cuanto a la profundidad y velocidad. El primer caso, se obtiene mayor cantidad de colores en los puntos lo que indica una mayor variabilidad en la distancia de otros obstáculos al vehículo. En la zona superior izquierda se observa gran variabilidad en la detección de obstáculos, desde objetos muy cercanos a otros con una distancia cercana a los 100m. En la zona de la derecha, hay gran cantidad de puntos a una distancia cercana debido a la gran cantidad de arboles y objetos situados en la zona derecha de la carretera por la que el vehículo circulaba. Estos objetos como se observa en la figura 84 no suponen peligro para el vehículo al tener velocidades nulas o negativas, lo que indica que no existe posibilidad inmediata de colisión. Destaca en ambas gráficas la presencia de acumulaciones de puntos en forma circular a una distancia media y con velocidad positiva que indica el acercamiento peligroso a un vehículo. Uno de los casos en los que el radar detecta estos puntos se muestra en la figura 83, donde momentos previos a la colisión se detecta un objeto muy cercano y con velocidad positiva peligrosa para el vehículo.

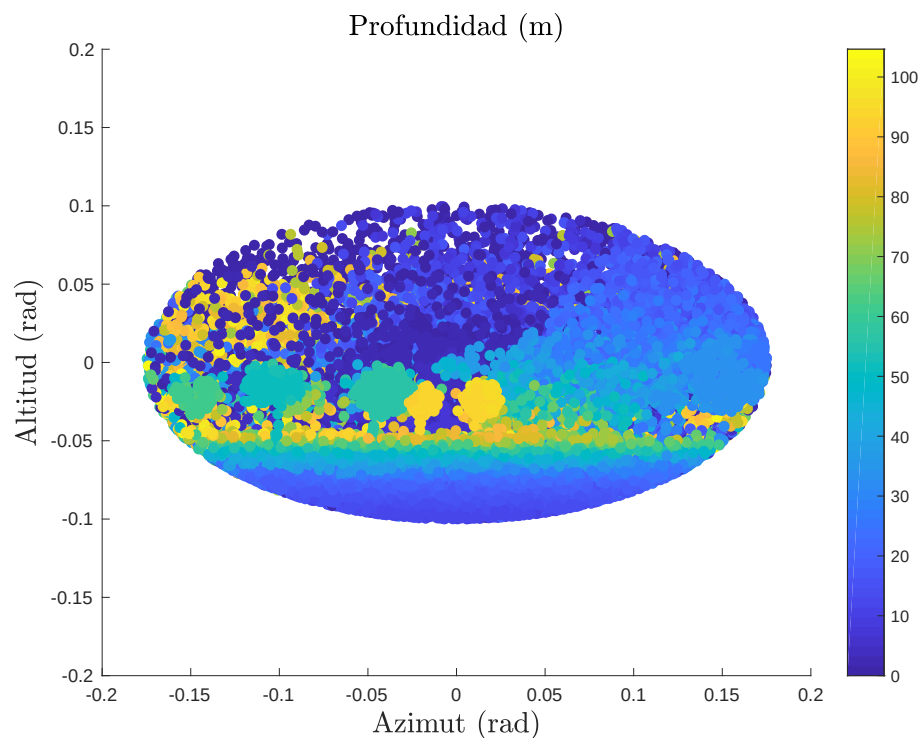


Figura 84. Azimut frente a altitud con gradiente de profundidad en la prueba de conducción manual

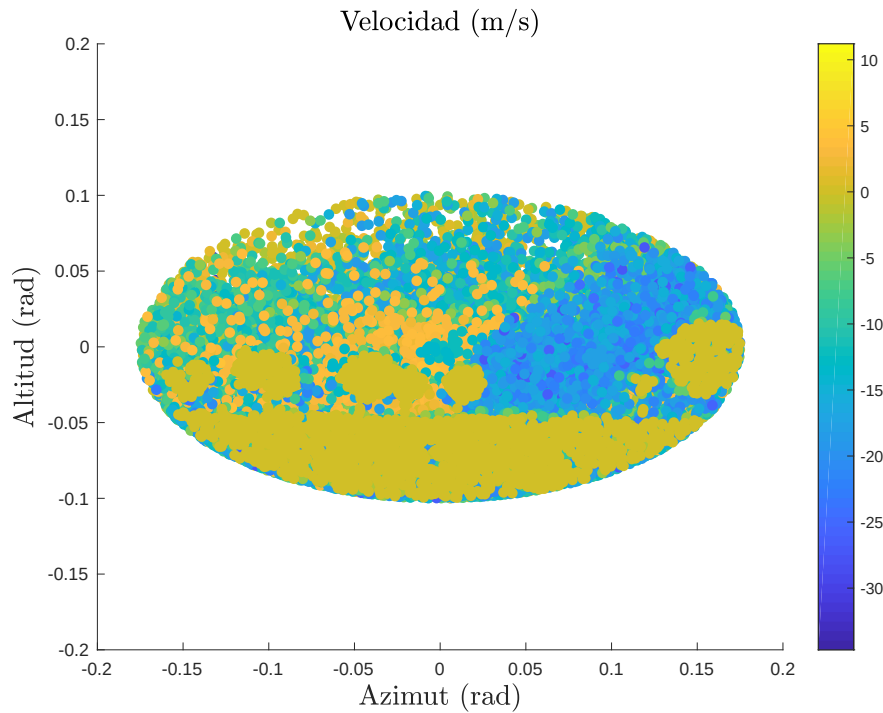


Figura 85. Azimut frente a altitud con gradiente de velocidad en la prueba de conducción manual

El sensor de colisiones es capaz de detectar la fuerza de la colisión en los diferentes ejes del vehículo y la intensidad de esta (ecuación 1). En la figura 86 se muestran las cuatro colisiones ocasionadas durante la simulación y que se estudiarán mediante los datos del sensor de colisión.

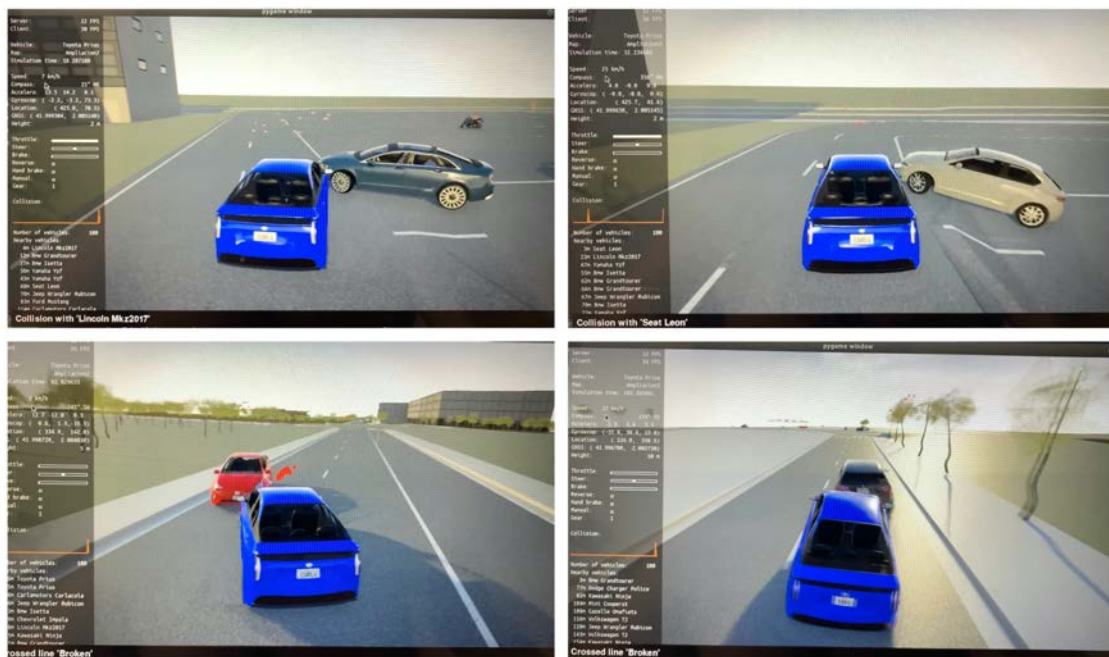


Figura 86. Colisiones ocurridas durante la simulación manual

En la figura 87 se detecta el impulso detectado en los diferentes ejes del vehículo y la intensidad. Se observan cuatro picos correspondientes a las colisiones mostradas en la figura 86. La primera colisión ocurre a baja velocidad con el otro coche detenido, por lo que la fuerza recibida en el eje longitudinal es negativa. Además, la colisión se produce en la zona delantera derecha del vehículo por lo que detecta un impulso en el eje Y. La segunda colisión se produce en el lateral derecho del vehículo por lo que el impulso se nota con mayor intensidad sobre el eje Y y en este caso negativo, lo que indica que el golpe se ha recibido por la derecha. La tercera colisión es la más fuerte ya que se circulaba a mayor velocidad y el otro vehículo circulaba en sentido contrario a la del vehículo simulado. El golpe es frontal por lo que se recibe una fuerza positiva mayor que el resto de las colisiones. En el último golpe, un vehículo circula delante y se colisiona con el por detrás. En este caso el vehículo se levanta un poco y las fuerzas se reciben en sentido contrario al tercer golpe, donde se produjo una colisión frontal. El impulso en el eje Z no varía en las primeras tres colisiones, sin embargo, en la última colisión al golpear el coche se levanta detectando un impulso en el eje vertical. La gráfica de la intensidad muestra como la tercera colisión, en la que se circulaba a alta velocidad y el choque es frontal, la intensidad es mucho mayor que en el resto de las colisiones. Los datos del sensor de colisión concuerdan con las variaciones inerciales y de velocidad detectadas con los otros sensores y detallados anteriormente.

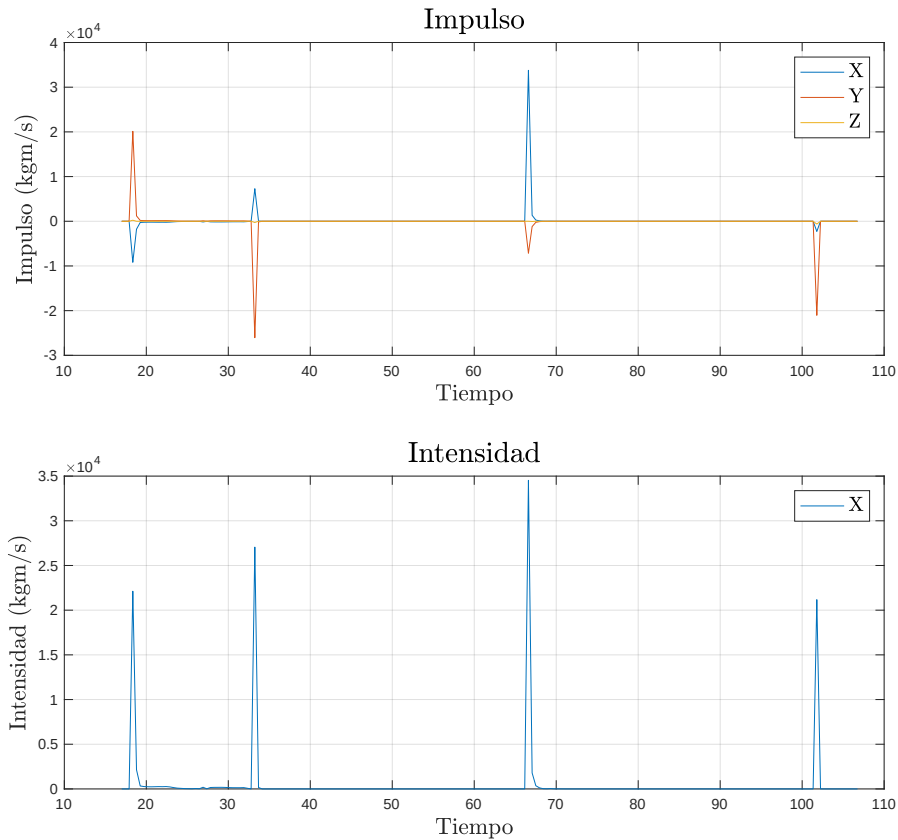


Figura 87. Impulso e intensidad detectada por el sensor de colisión

Durante esta simulación manual, se ha visto como el control del vehículo es mucho mas brusco detectando grandes oscilaciones en la mayoría de los sensores a diferencia de lo que ocurría con el control automático, donde tanto el comportamiento del vehículo como el comportamiento de los sensores era mucho más lineal.

5.3.3. Captura de imágenes y LIDAR

Además de los datos de los diferentes sensores detallados en los dos apartados anteriores, también se puede almacenar datos de las diferentes cámaras de las que dispone el vehículo y datos obtenidos por el sensor LIDAR. Las cámaras se mostraron en apartados anteriores por lo que este punto se centrará en las capturas del LIDAR.

Para comprobar el funcionamiento del LIDAR, se han simulado diferentes situaciones en diferentes escenarios en los que se compararán las imágenes capturadas con la cámara

RGB y el mapa creado por el LIDAR. El primer escenario ha sido la entrada al parking este de la Escuela de Ingenierías (figura 88). En las imágenes del LIDAR se obtiene una visión 360° en la que se pueden diferenciar los árboles del entorno e incluso las personas que cruzan por el paso de peatones (redondeado en rojo). Por tanto, en este caso se obtiene buena información del entorno con el mapa creado con el LIDAR.

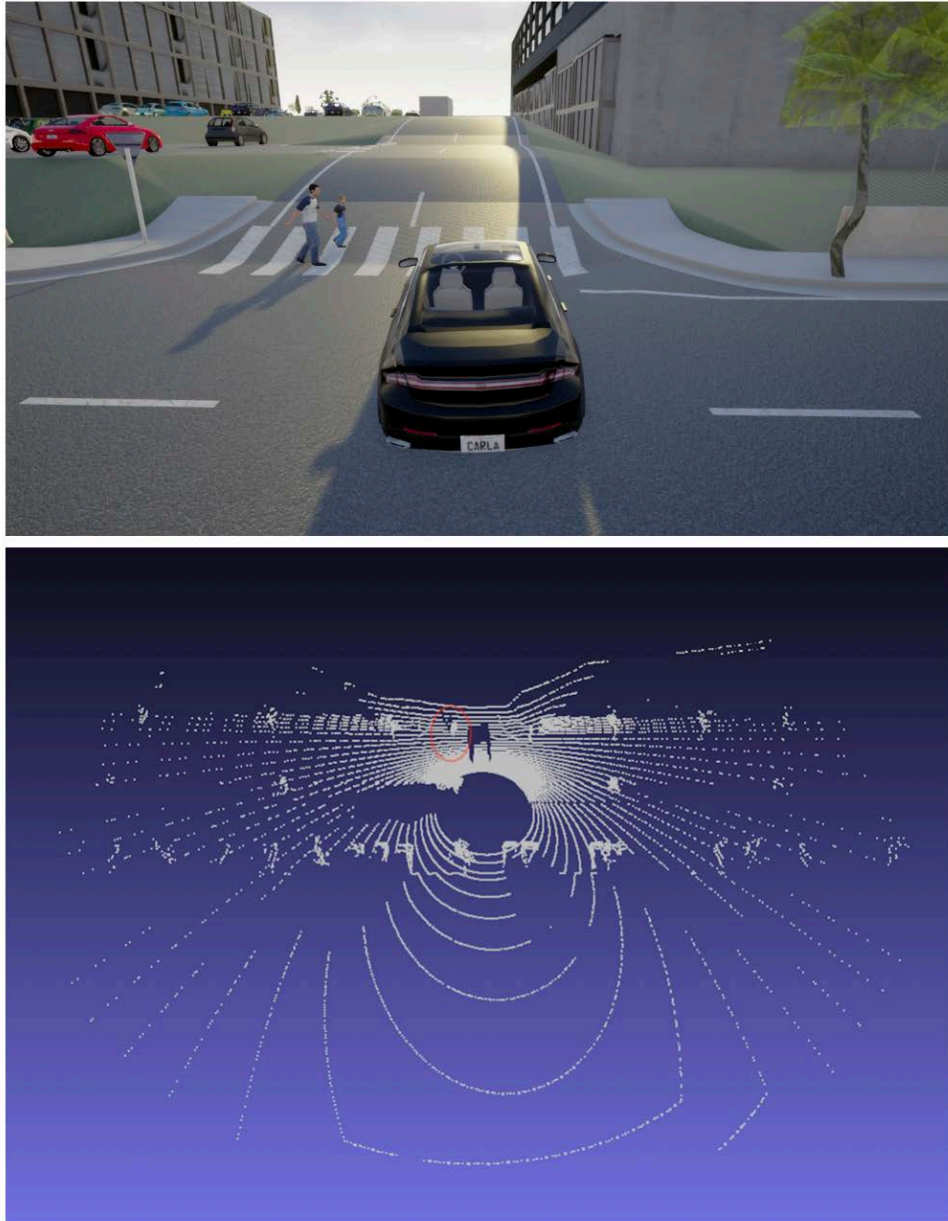


Figura 88. Captura de la cámara y mapa del LIDAR en la entrada al parking de la Escuela de Ingenierías

En segundo lugar, se ha capturado el trayecto frente a la facultad de Ciencias de la Salud dirección este (figura 89). En este caso, al circular también a mayor velocidad, se obtiene

un mapa menos claro que el anterior donde únicamente se pueden ver gran cantidad de puntos en la parte derecha del mapa que corresponde a los vehículos aparcados.

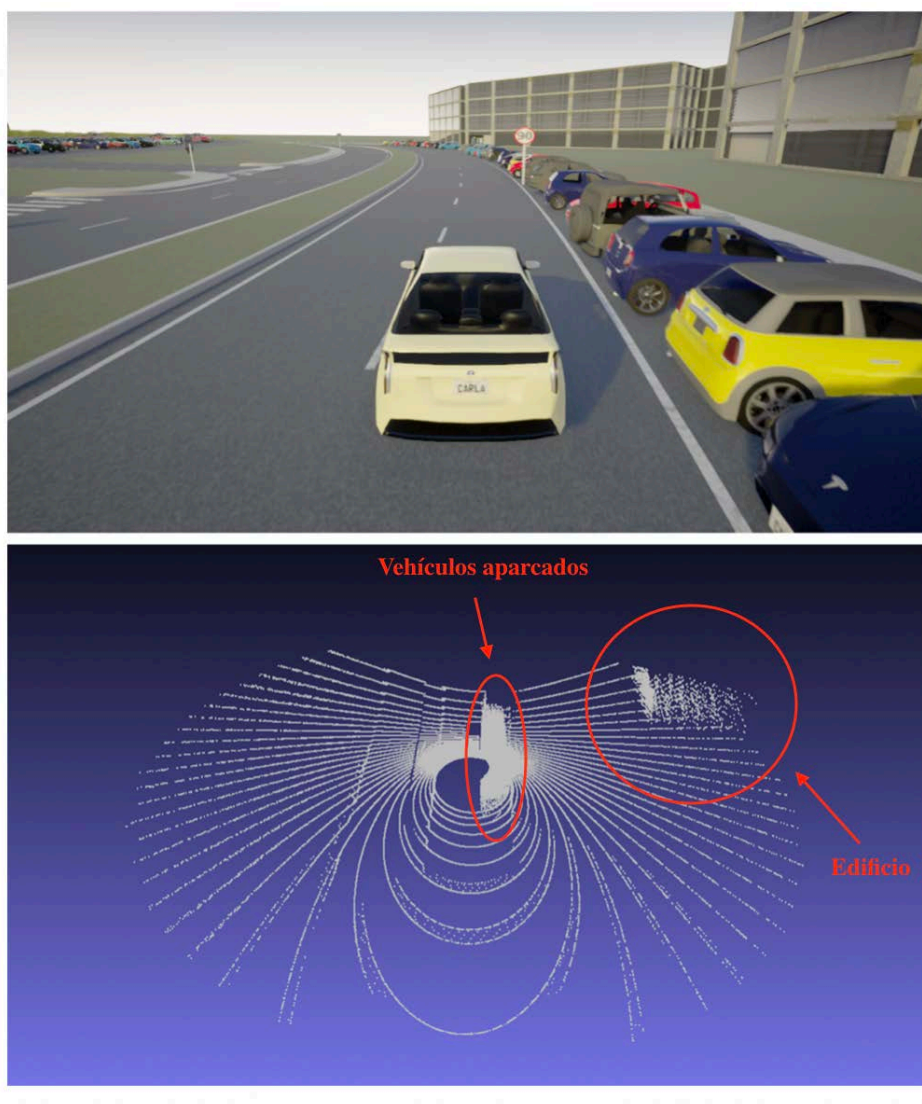


Figura 89. Captura de la cámara y mapa del LIDAR frente a la Facultad de Ciencias de la Salud

Por último, se ha recorrido la carretera anterior, pero en dirección contraria y con una colisión producida con una señal (figura 90). En este caso se puede distinguir en la parte trasera del vehículo una gran pared, que se corresponde con la facultad de ciencias de la salud, e incluso la señal con la que ha colisionado el vehículo.

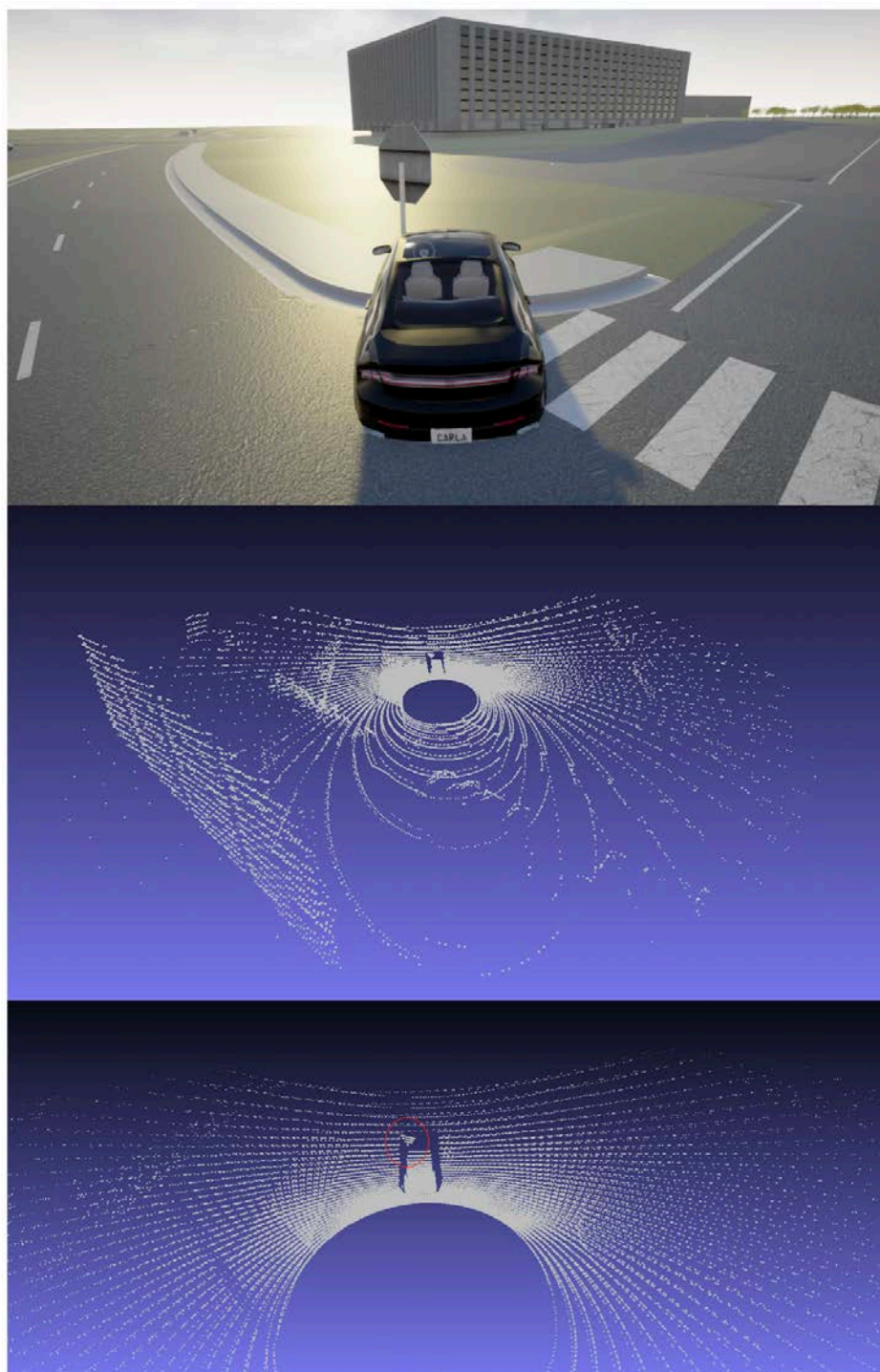


Figura 90. Captura de la cámara y mapa del LIDAR frente a la Facultad de Ciencias de la Salud en dirección oeste

5.4. Coste computacional

El coste computacional obtenido a partir del comando ‘top’ en Linux se muestra en la tabla 4. En cada momento de la simulación, se distinguen dos procesos con orden

UE4Editor y Python. El primero de ellos aparece al arrancar el motor Unreal Engine y se corresponde con el módulo *CARLA Simulator*, programado en C++, y se encarga de las tareas más complejas. La segunda orden aparece al ejecutar el programa de simulación (*manual_control.py*), del módulo *PythonAPI*, que permite la visualización y control del vehículo autónomo. Como se observa en los datos, el primer proceso (*UE4 Editor*) tiene un coste computacional mayor que el segundo (*python*), con un tiempo de CPU que aumenta con mayor velocidad que el otro proceso estudiado. Se observa como el uso de CPU aumenta en los momentos iniciales como consecuencia del comienzo del proceso de simulación, aumentando rápidamente el tiempo total de uso utilizado por la CPU. En ambos procesos, el uso de CPU supera el 180% en todo momento, ya que el programa de simulación requiere un alto nivel de procesamiento debido a su complejidad y a la gran cantidad de información que tiene que procesar. El hardware utilizado ha sido:

- Ordenador: Lenovo Legion Y530-15ICH
- GPU: NVIDIA GeForce GTX 1050
- CPU: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
- Disco duro: Samsung SSD 970 EVO 250GB
- RAM: 16GB (2xMemoria RAM de 8 GB (DDR4, 2666 MT/s, PC4-21300, Single Rank x 8, SODIMM, 260-Pin)

Los datos recogidos en la tabla del coste computacional son:

- PID: identificador del proceso.
- Usuario: usuario propietario del proceso.
- PR: prioridad del proceso.
- NI: asignación de la prioridad.
- VIRT: cantidad de memoria virtual usada por el proceso.
- RES: cantidad de memoria RAM utilizada por el proceso.
- SHR: memoria compartida.
- S: estado del proceso. (R = en ejecución, S = dormido)
- %CPU: porcentaje de CPU utilizado.
- %MEM: porcentaje de memoria física utilizada por el proceso.
- HORA+: tiempo total de CPU que ha utilizado el proceso desde su inicio.
- ORDEN: comando utilizado para iniciar el proceso.

Momento de la simulación	Datos hardware										
Arranque Unreal Engine	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	1207088	520392	395668	R	90,4	3,2	0:05.98 UE4Editor
Inicio de la simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	6995204	3,070g	645276	R	257,0	19,8	2:01.11 UE4Editor
	3042	eloy96vg	20	0	2509212	305696	34740	R	196,4	1,9	0:19.99 python
30s de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7152000	3,085g	649560	R	259,1	19,8	3:34.75 UE4Editor
	3042	eloy96vg	20	0	3222908	343884	34936	R	211,2	2,1	1:35.16 python
1 min de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7100208	3,032g	649596	S	262,5	19,5	3:42.65 UE4Editor
	3042	eloy96vg	20	0	3222908	343884	34936	R	209,0	2,1	1:41.45 python
1 min 30s de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7100208	3,041g	649600	R	251,7	19,6	4:52.83 UE4Editor
	3042	eloy96vg	20	0	3288444	344148	34936	R	190,1	2,1	2:37.04 python
2 min de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7165744	3,054g	651648	S	254,0	19,7	6:09.70 UE4Editor
	3042	eloy96vg	20	0	3288444	344148	34936	S	204,3	2,1	3:38.99 python
2 min 30s de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7161640	3,048g	651648	R	254,8	19,6	7:26.84 UE4Editor
	3042	eloy96vg	20	0	3288444	344148	34936	S	202,3	2,1	4:40.42 python
3 min de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7227176	3,048g	651648	R	244,7	19,6	8:42.50 UE4Editor
	3042	eloy96vg	20	0	3288444	344148	34936	S	187,1	2,1	5:41.40 python
3 min 30s de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7227176	3,049g	651648	R	253,3	19,6	9:57.95 UE4Editor
	3042	eloy96vg	20	0	3288444	344148	34936	R	206,0	2,1	6:42.75 python
4 min de simulación	PID	USUARIO	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	HORA+ ORDEN
	2760	eloy96vg	20	0	7227176	3,049g	651648	R	249,3	19,6	11:04.44 UE4Editor
	3042	eloy96vg	20	0	3288444	344148	34936	R	208,3	2,1	7:38.07 python

Tabla 4. Coste computacional

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

En este trabajo se ha llevado a cabo una revisión inicial de la historia y los fundamentos de los vehículos autónomos desde sus comienzos hasta la actualidad. Con esta base, el objetivo era crear un entorno en el que realizar pruebas de conducción autónoma en el simulador “CARLA Simulator”. El entorno construido ha sido la ampliación universitaria del Campus de Teatinos, en el que se han realizado numerosas pruebas de conducción y de las que se han sacado diversas conclusiones sobre su funcionamiento.

En el capítulo 2 se ha realizado una pequeña revisión de los antecedentes de los vehículos autónomos, desde sus primeras apariciones hasta la actualidad. Además, se han estudiado los diferentes niveles de autonomía que se consideran a la hora de clasificar la capacidad del vehículo para circular sin conductor y los diferentes sensores de los que disponen los vehículos autónomos actuales.

En el capítulo 3 se ha explicado detalladamente el proceso de creación del entorno de la ampliación en el programa RoadRunner y su posterior importación a CARLA. En este proceso se trata la adaptación del mapa de un programa a otro y la creación de los diferentes elementos que dotan de realidad al entorno.

En el capítulo 4 se ha descrito la implementación de la simulación en la que se explican los diferentes programas utilizados para simular la conducción y almacenar datos sensoriales.

En el capítulo 5, por último, se han estudiado los resultados obtenidos tras realizar las diferentes simulaciones, así como el estudio de los datos capturados tras su procesamiento en Matlab. El estudio ha mostrado:

1. El vehículo autónomo circula adecuadamente alrededor del entorno adaptándose a todas las situaciones en las que se ha puesto a prueba, adaptando sus movimientos al entorno como respuesta a los diferentes estímulos que detectan sus sensores. En la gran cantidad de pruebas realizadas, el vehículo apenas comete infracciones de tráfico o colisiones con otros vehículos u objetos.
2. El simulador de la zona de la ampliación universitaria del Campus de Teatinos se asemeja en gran medida al entorno real, por lo que se podría desarrollar cualquier entorno en el que realizar pruebas de conducción autónoma.
3. Los datos capturados por los sensores son acordes a la simulación realizada, por lo que los datos sensoriales son fiables. Gracias a estos datos el vehículo puede circular autónomamente y de forma adecuada por el entorno.
4. En la prueba realizada manualmente se observa un comportamiento mucho menos lineal que en la realizada autónomamente, con movimientos más bruscos y numerosas oscilaciones en los datos obtenidos por los sensores. Por tanto, resulta más adecuado el control automático que el control manual en este caso.
5. La estimación de coste computacional indica que el simulador consume gran cantidad de recursos con gran cantidad de CPU usada y un tiempo alto durante la simulación

Este estudio que abre una puerta a realizar pruebas de simulación en entornos conocidos que posteriormente se puedan llevar a cabo a la realidad. Con esto se puede limitar errores que pudieran aparecer en situaciones reales en las que no se han realizado pruebas anteriores. Con este método, al estudiar los datos de los sensores más importantes se pueden ajustar y adecuar en su adaptación al vehículo real, limitando entre otras cosas el coste que puede suponer realizar las pruebas en un entorno y vehículos reales directamente.

6.2. Trabajos futuros

Los resultados obtenidos en este trabajo son prometedores, ya que como se ha comentado, el simulador puede utilizarse como paso previo a la realización de pruebas en entornos reales, limitando posibles fallos y costes en las pruebas. Aunque la conducción autónoma está muy avanzada, aún queda mucho trabajo por hacer para lograr la autonomía total y simuladores como el utilizado en este trabajo pueden tener gran importancia para lograr este objetivo. Por tanto, uno de los posibles trabajos futuros puede ser la adaptación de los sensores utilizados para ser implementados en vehículos reales y en entornos conocidos como el de la ampliación.

También, se podría mejorar visualmente el entorno creado creando un mapa en alta definición y con los edificios más realistas, aunque en este caso se trataría de un trabajo más centrado en el diseño que en las pruebas de conducción.

Por último, con el simulador se podrían hacer pruebas en cualquier entorno deseado, creando previamente el mapa e implementándolo en CARLA. Esto tiene la ventaja de poder realizar pruebas en entornos en los que posteriormente se van a llevar a cabo pruebas reales y con los que se puedan comparar resultados.

Anexo

A.1. Instalación de CARLA en Ubuntu

Para instalar CARLA [24] en Linux se deben cumplir los siguientes requisitos:

1. Ubuntu 16.05 o Ubuntu 18.05.
2. 30 GB de espacio libre.
3. GPU de al menos 4 GB
4. Dos puertos TCP y buena conexión a internet.

Para la instalación del simulador se seguirán los siguientes pasos:

1. Instalación de dependencias para que CARLA pueda ejecutarse correctamente.

Para ello se ejecutarán en el terminal los siguientes comandos:

```
>> sudo apt-get update &&

sudo apt-get install wget software-properties-common &&

sudo add-apt-repository ppa:ubuntu-toolchain-r/test &&

wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add -
&&

sudo apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-
xenial-8 main" &&

sudo apt-get update

>> sudo apt-get install build-essential clang-8 lld-8 g++-7 cmake ninja-
build libvulkan1 python python-pip python-dev python3-dev python3-pip
libpng16-dev libtiff5-dev libjpeg-dev tzdata sed curl unzip autoconf
libtool rsync libxml2-dev &&

pip2 install --user setuptools &&

pip3 install --user setuptools
```

2. Creación de una cuenta en GitHub para poder acceder a los repositorios de Unreal Engine, los cuales son privados.
3. Instalación de Unreal Engine. Para ello se clonará Unreal Engine 4.22 en el ordenador.

```
>> git clone --depth=1 -b 4.22  
https://github.com/EpicGames/UnrealEngine.git ~/UnrealEngine_4.22
```

Posteriormente, se accederá a la carpeta donde se ha clonado Unreal Engine y se ejecutarán los siguientes comandos:

```
>> cd ~/UnrealEngine_4.22  
  
>> wget https://carla-releases.s3.eu-west-  
3.amazonaws.com/Linux/UE_Patch/430667-13636743-patch.txt  
~/430667-13636743-patch.txt patch --strip=4 < ~/430667-13636743-  
patch.txt  
  
>> ./Setup.sh && ./GenerateProjectFiles.sh && make
```

4. Instalación de CARLA. Para ello se clonará el repositorio mediante la siguiente línea de comando:

```
>> git clone https://github.com/carla-simulator/carla
```

Posteriormente, se accederá a la carpeta clonada y se ejecutará el comando que permite instalar la versión 0.9.7 de CARLA (el simulador ya tiene versiones superiores, pero para garantizar la correcta compatibilidad del entorno creado es recomendable instalar esta versión). Además, se ejecutará el script para obtener todos los elementos del simulador (mapas, vehículos, etc.).

```
>> cd ~/carla  
  
>> git checkout 9d068d2  
  
>> ./Update.sh
```

5. Para añadir el entorno de la ampliación universitaria al simulador se incluirán los plugins *RoadRunnerIntegration* y *RoadRunnerImporter* en la ruta “~/carla/Unreal/CarlaUE4/Plugins”. Además, el paquete que contiene el mapa, *Package1*, debe ser incluido en la ruta “~/carla/Unreal/CarlaUE4/Content”. Para

obtener el mapa y los plugins se clonará el repositorio que contiene dichos paquetes:

```
>> git clone https://github.com/eloy96vg/Ampliacion.git
```

Tras este paso, se debe copiar el paquete *Static*, que se encuentra en la ruta “~/carla/Unreal/CarlaUE4/Content/Carla”, y pegar en la ruta “~/carla/Unreal/CarlaUE4/Content/Package1”. Posteriormente, se clonará el repositorio que contiene los paquetes *Terrain* y *Terrain2*, los cuales se deben copiar en la ruta “~/carla/Unreal/CarlaUE4/Content/Package1/Static”.

```
>> git clone https://github.com/eloy96vg/Static.git
```

6. Para compilar el simulador y abrir Unreal Engine es necesario ejecutar previamente el comando que permite que CARLA encuentre Unreal Engine. Este comando será necesario cada vez que se vaya a abrir el simulador en Unreal Engine:

```
>> export UE4_ROOT=~/.UnrealEngine_4.22
```

Posteriormente, se ejecuta el commando para compilar el simulador:

```
>> make launch
```

Por ultimo, para compilar el cliente API se ejecutará el siguiente comando:

```
>> make PythonAPI
```

A.2. Manual_control.py

```
#!/usr/bin/env python
# Copyright (c) 2019 Computer Vision Center (CVC) at the Universitat Autònoma de
# Barcelona (UAB).
#
# This work is licensed under the terms of the MIT license.
# For a copy, see <https://opensource.org/licenses/MIT>.
#
# Allows controlling a vehicle with a keyboard. For a simpler and more
# documented example, please take a look at tutorial.py.
"""
```

Welcome to CARLA manual control.

Use ARROWS or WASD keys for control.

```
W          : throttle
S          : brake
AD         : steer
Q          : toggle reverse
Space      : hand-brake
P          : toggle autopilot
M          : toggle manual transmission
,/.        : gear up/down

TAB        : change sensor position
`          : next sensor
[1-9]      : change to sensor [1-9]
G          : toggle radar visualization
C          : change weather (Shift+C reverse)
Backspace  : change vehicle

R          : toggle recording images to disk

CTRL + R   : toggle recording of simulation (replacing any previous)
CTRL + P   : start replaying last recorded simulation
CTRL + +   : increments the start time of the replay by 1 second (+SHIFT = 10
seconds)
CTRL + -   : decrements the start time of the replay by 1 second (+SHIFT = 10
seconds)

F1         : toggle HUD
H/?        : toggle help
ESC        : quit
"""
```

```
from __future__ import print_function
```

```
# =====
# -- find carla module -----
# =====
```

```
import glob
import os
import sys
```

```
try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass
```

```
# =====
# -- imports -----
# =====
```

```
import carla
```

```
from carla import ColorConverter as cc
```

```
import argparse
import collections
import datetime
import logging
import math
import random
import re
import weakref
```

```
try:
    import pygame
    from pygame.locals import KMOD_CTRL
    from pygame.locals import KMOD_SHIFT
```

```

from pygame.locals import K_0
from pygame.locals import K_9
from pygame.locals import K_BACKQUOTE
from pygame.locals import K_BACKSPACE
from pygame.locals import K_COMMA
from pygame.locals import K_DOWN
from pygame.locals import K_ESCAPE
from pygame.locals import K_F1
from pygame.locals import K_LEFT
from pygame.locals import K_PERIOD
from pygame.locals import K_RIGHT
from pygame.locals import K_SLASH
from pygame.locals import K_SPACE
from pygame.locals import K_TAB
from pygame.locals import K_UP
from pygame.locals import K_a
from pygame.locals import K_c
from pygame.locals import K_g
from pygame.locals import K_d
from pygame.locals import K_h
from pygame.locals import K_m
from pygame.locals import K_p
from pygame.locals import K_q
from pygame.locals import K_r
from pygame.locals import K_s
from pygame.locals import K_w
from pygame.locals import K_MINUS
from pygame.locals import K_EQUALS
except ImportError:
    raise RuntimeError('cannot import pygame, make sure pygame package is installed')

try:
    import numpy as np
except ImportError:
    raise RuntimeError('cannot import numpy, make sure numpy package is installed')

# =====
# -- Global functions -----
# =====

def find_weather_presets():
    rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')
    name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))
    presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+', x)]
    return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]

def get_actor_display_name(actor, truncate=250):
    name = ' '.join(actor.type_id.replace('_', '.').title().split('.')[1:])
    return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name

# =====
# -- World -----
# =====

class World(object):
    def __init__(self, carla_world, hud, args):
        self.world = carla_world
        self.actor_role_name = args.rolename
        try:
            self.map = self.world.get_map()
        except RuntimeError as error:
            print('RuntimeError: {}'.format(error))
            print('  The server could not send the OpenDRIVE (.xodr) file:')
            print('  Make sure it exists, has the same name of your town, and is
correct.')
            sys.exit(1)
        self.hud = hud
        self.player = None
        self.collision_sensor = None
        self.lane_invasion_sensor = None

```

```

self.gnss_sensor = None
self.imu_sensor = None
self.radar_sensor = None
self.camera_manager = None
self._weather_presets = find_weather_presets()
self._weather_index = 0
self._actor_filter = args.filter
self._gamma = args.gamma
self.restart()
self.world.on_tick(hud.on_world_tick)
self.recording_enabled = False
self.recording_start = 0

def restart(self):
    self.player_max_speed = 1.589
    self.player_max_speed_fast = 3.713
    # Keep same camera config if the camera manager exists.
    cam_index = self.camera_manager.index if self.camera_manager is not None else 0
    cam_pos_index = self.camera_manager.transform_index if self.camera_manager is not None else 0
    # Get a random blueprint.
    blueprint =
random.choice(self.world.get_blueprint_library().filter(self._actor_filter))
    blueprint.set_attribute('role_name', self.actor_role_name)
    if blueprint.has_attribute('color'):
        color =
random.choice(blueprint.get_attribute('color').recommended_values)
        blueprint.set_attribute('color', color)
    if blueprint.has_attribute('driver_id'):
        driver_id =
random.choice(blueprint.get_attribute('driver_id').recommended_values)
        blueprint.set_attribute('driver_id', driver_id)
    if blueprint.has_attribute('is_invincible'):
        blueprint.set_attribute('is_invincible', 'true')
    # set the max speed
    if blueprint.has_attribute('speed'):
        self.player_max_speed =
float(blueprint.get_attribute('speed').recommended_values[1])
        self.player_max_speed_fast =
float(blueprint.get_attribute('speed').recommended_values[2])
    else:
        print("No recommended values for 'speed' attribute")
    # Spawn the player.
    if self.player is not None:
        spawn_point = self.player.get_transform()
        spawn_point.location.z += 2.0
        spawn_point.rotation.roll = 0.0
        spawn_point.rotation.pitch = 0.0
        self.destroy()
        self.player = self.world.try_spawn_actor(blueprint, spawn_point)
    while self.player is None:
        if not self.map.get_spawn_points():
            print('There are no spawn points available in your map/town.')
            print('Please add some Vehicle Spawn Point to your UE4 scene.')
            sys.exit(1)
        spawn_points = self.map.get_spawn_points()
        spawn_point = random.choice(spawn_points) if spawn_points else
carla.Transform()
        self.player = self.world.try_spawn_actor(blueprint, spawn_point)
    # Set up the sensors.
    self.collision_sensor = CollisionSensor(self.player, self.hud)
    self.lane_invasion_sensor = LaneInvasionSensor(self.player, self.hud)
    self.gnss_sensor = GnssSensor(self.player)
    self.imu_sensor = IMUSensor(self.player)
    self.camera_manager = CameraManager(self.player, self.hud, self._gamma)
    self.camera_manager.transform_index = cam_pos_index
    self.camera_manager.set_sensor(cam_index, notify=False)
    actor_type = get_actor_display_name(self.player)
    self.hud.notification(actor_type)

def next_weather(self, reverse=False):
    self._weather_index += -1 if reverse else 1
    self._weather_index %= len(self._weather_presets)
    preset = self._weather_presets[self._weather_index]

```

```

        self.hud.notification('Weather: %s' % preset[1])
        self.player.get_world().set_weather(preset[0])

    def toggle_radar(self):
        if self.radar_sensor is None:
            self.radar_sensor = RadarSensor(self.player)
        elif self.radar_sensor.sensor is not None:
            self.radar_sensor.sensor.destroy()
            self.radar_sensor = None

    def tick(self, clock):
        self.hud.tick(self, clock)

    def render(self, display):
        self.camera_manager.render(display)
        self.hud.render(display)

    def destroy_sensors(self):
        self.camera_manager.sensor.destroy()
        self.camera_manager.sensor = None
        self.camera_manager.index = None

    def destroy(self):
        if self.radar_sensor is not None:
            self.toggle_radar()
        actors = [
            self.camera_manager.sensor,
            self.collision_sensor.sensor,
            self.lane_invasion_sensor.sensor,
            self.gnss_sensor.sensor,
            self.imu_sensor.sensor,
            self.player]
        for actor in actors:
            if actor is not None:
                actor.destroy()

# =====
# -- KeyboardControl -----
# =====

class KeyboardControl(object):
    def __init__(self, world, start_in_autopilot):
        self._autopilot_enabled = start_in_autopilot
        if isinstance(world.player, carla.Vehicle):
            self._control = carla.VehicleControl()
            world.player.set_autopilot(self._autopilot_enabled)
        elif isinstance(world.player, carla.Walker):
            self._control = carla.WalkerControl()
            self._autopilot_enabled = False
            self._rotation = world.player.get_transform().rotation
        else:
            raise NotImplementedError("Actor type not supported")
        self._steer_cache = 0.0
        world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

    def parse_events(self, client, world, clock):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                return True
            elif event.type == pygame.KEYUP:
                if self._is_quit_shortcut(event.key):
                    return True
                elif event.key == K_BACKSPACE:
                    world.restart()
                elif event.key == K_F1:
                    world.hud.toggle_info()
                elif event.key == K_h or (event.key == K_SLASH and
pygame.key.get_mods() & KMOD_SHIFT):
                    world.hud.help.toggle()
                elif event.key == K_TAB:
                    world.camera_manager.toggle_camera()
                elif event.key == K_c and pygame.key.get_mods() & KMOD_SHIFT:
                    world.next_weather(reverse=True)

```

```

elif event.key == K_c:
    world.next_weather()
elif event.key == K_g:
    world.toggle_radar()
elif event.key == K_BACKQUOTE:
    world.camera_manager.next_sensor()
elif event.key > K_0 and event.key <= K_9:
    world.camera_manager.set_sensor(event.key - 1 - K_0)
elif event.key == K_r and not (pygame.key.get_mods() & KMOD_CTRL):
    world.camera_manager.toggle_recording()
elif event.key == K_r and (pygame.key.get_mods() & KMOD_CTRL):
    if (world.recording_enabled):
        client.stop_recorder()
        world.recording_enabled = False
        world.hud.notification("Recorder is OFF")
    else:
        client.start_recorder("manual_recording.rec")
        world.recording_enabled = True
        world.hud.notification("Recorder is ON")
elif event.key == K_p and (pygame.key.get_mods() & KMOD_CTRL):
    # stop recorder
    client.stop_recorder()
    world.recording_enabled = False
    # work around to fix camera at start of replaying
    currentIndex = world.camera_manager.index
    world.destroy_sensors()
    # disable autopilot
    self._autopilot_enabled = False
    world.player.set_autopilot(self._autopilot_enabled)
    world.hud.notification("Replaying file 'manual_recording.rec'")
    # replayer
    client.replay_file("manual_recording.rec", world.recording_start,
0, 0)

    world.camera_manager.set_sensor(currentIndex)
elif event.key == K_MINUS and (pygame.key.get_mods() & KMOD_CTRL):
    if pygame.key.get_mods() & KMOD_SHIFT:
        world.recording_start -= 10
    else:
        world.recording_start -= 1
    world.hud.notification("Recording start time is %d" %
(world.recording_start))
elif event.key == K_EQUALS and (pygame.key.get_mods() & KMOD_CTRL):
    if pygame.key.get_mods() & KMOD_SHIFT:
        world.recording_start += 10
    else:
        world.recording_start += 1
    world.hud.notification("Recording start time is %d" %
(world.recording_start))
if isinstance(self._control, carla.VehicleControl):
    if event.key == K_q:
        self._control.gear = 1 if self._control.reverse else -1
    elif event.key == K_m:
        self._control.manual_gear_shift = not
self._control.manual_gear_shift
        self._control.gear = world.player.get_control().gear
        world.hud.notification('%s Transmission' %
('Manual' if
self._control.manual_gear_shift else 'Automatic'))
    elif self._control.manual_gear_shift and event.key == K_COMMA:
        self._control.gear = max(-1, self._control.gear - 1)
    elif self._control.manual_gear_shift and event.key == K_PERIOD:
        self._control.gear = self._control.gear + 1
    elif event.key == K_p and not (pygame.key.get_mods() &
KMOD_CTRL):
        self._autopilot_enabled = not self._autopilot_enabled
        world.player.set_autopilot(self._autopilot_enabled)
        world.hud.notification('Autopilot %s' % ('On' if
self._autopilot_enabled else 'Off'))
    if not self._autopilot_enabled:
        if isinstance(self._control, carla.VehicleControl):
            self._parse_vehicle_keys(pygame.key.get_pressed(), clock.get_time())
            self._control.reverse = self._control.gear < 0
        elif isinstance(self._control, carla.WalkerControl):
            self._parse_walker_keys(pygame.key.get_pressed(), clock.get_time(),
world)

```

```

        world.player.apply_control(self._control)

    def _parse_vehicle_keys(self, keys, milliseconds):
        self._control.throttle = 1.0 if keys[K_UP] or keys[K_w] else 0.0
        steer_increment = 5e-4 * milliseconds
        if keys[K_LEFT] or keys[K_a]:
            if self._steer_cache > 0:
                self._steer_cache = 0
            else:
                self._steer_cache -= steer_increment
        elif keys[K_RIGHT] or keys[K_d]:
            if self._steer_cache < 0:
                self._steer_cache = 0
            else:
                self._steer_cache += steer_increment
        else:
            self._steer_cache = 0.0
        self._steer_cache = min(0.7, max(-0.7, self._steer_cache))
        self._control.steer = round(self._steer_cache, 1)
        self._control.brake = 1.0 if keys[K_DOWN] or keys[K_s] else 0.0
        self._control.hand_brake = keys[K_SPACE]

    def _parse_walker_keys(self, keys, milliseconds, world):
        self._control.speed = 0.0
        if keys[K_DOWN] or keys[K_s]:
            self._control.speed = 0.0
        if keys[K_LEFT] or keys[K_a]:
            self._control.speed = .01
            self._rotation.yaw -= 0.08 * milliseconds
        if keys[K_RIGHT] or keys[K_d]:
            self._control.speed = .01
            self._rotation.yaw += 0.08 * milliseconds
        if keys[K_UP] or keys[K_w]:
            self._control.speed = world.player_max_speed_fast if
pygame.key.get_mods() & KMOD_SHIFT else world.player_max_speed
        self._control.jump = keys[K_SPACE]
        self._rotation.yaw = round(self._rotation.yaw, 1)
        self._control.direction = self._rotation.get_forward_vector()

    @staticmethod
    def _is_quit_shortcut(key):
        return (key == K_ESCAPE) or (key == K_q and pygame.key.get_mods() &
KMOD_CTRL)

# =====
# -- HUD -----
# =====

class HUD(object):
    def __init__(self, width, height):
        self.dim = (width, height)
        font = pygame.font.Font(pygame.font.get_default_font(), 20)
        font_name = 'courier' if os.name == 'nt' else 'mono'
        fonts = [x for x in pygame.font.get_fonts() if font_name in x]
        default_font = 'ubuntumono'
        mono = default_font if default_font in fonts else fonts[0]
        mono = pygame.font.match_font(mono)
        self._font_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else 14)
        self._notifications = FadingText(font, (width, 40), (0, height - 40))
        self.help = HelpText(pygame.font.Font(mono, 24), width, height)
        self.server_fps = 0
        self.frame = 0
        self.simulation_time = 0
        self._show_info = True
        self._info_text = []
        self._server_clock = pygame.time.Clock()

    def on_world_tick(self, timestamp):
        self._server_clock.tick()
        self.server_fps = self._server_clock.get_fps()
        self.frame = timestamp.frame
        self.simulation_time = timestamp.elapsed_seconds

```

```

def tick(self, world, clock):
    self._notifications.tick(world, clock)
    if not self._show_info:
        return
    t = world.player.get_transform()
    v = world.player.get_velocity()
    c = world.player.get_control()
    compass = world.imu_sensor.compass
    heading = 'N' if compass > 270.5 or compass < 89.5 else ''
    heading += 'S' if 90.5 < compass < 269.5 else ''
    heading += 'E' if 0.5 < compass < 179.5 else ''
    heading += 'W' if 180.5 < compass < 359.5 else ''
    colhist = world.collision_sensor.get_collision_history()
    collision = [colhist[x + self.frame - 200] for x in range(0, 200)]
    max_col = max(1.0, max(collision))
    collision = [x / max_col for x in collision]
    vehicles = world.world.get_actors().filter('vehicle.*')
    self._info_text = [
        'Server:  % 16.0f FPS' % self.server_fps,
        'Client:  % 16.0f FPS' % clock.get_fps(),
        '',
        'Vehicle: % 20s' % get_actor_display_name(world.player, truncate=20),
        'Map:      % 20s' % world.map.name,
        'Simulation time: % 0s' %
datetime.timedelta(seconds=float(self.simulation_time)).total_seconds(),
        '',
        'Speed:    % 0.0f km/h' % (3.6 * math.sqrt(v.x ** 2 + v.y ** 2 + v.z **
2)),
        u'Compass:% 17.0f\N{DEGREE SIGN} % 2s' % (compass, heading),
        'Accelero: %5.1f %5.1f %5.1f' % (world.imu_sensor.accelerometer),
        'Gyroscop: (%5.1f,%5.1f,%5.1f)' % (world.imu_sensor.gyroscope),
        'Location: % 20s' % ('(% 5.1f, % 5.1f)' % (t.location.x, t.location.y)),
        'GNSS: % 24s' % ('(% 2.6f, % 3.6f)' % (world.gnss_sensor.lat,
world.gnss_sensor.lon)),
        'Height:  % 18.0f m' % t.location.z,
        ''
    ]

    datos = open("datos.txt", "a")
    datos.write('\n' +
str(datetime.timedelta(seconds=float(self.simulation_time)).total_seconds()) + " " +
str(
    3.6 * math.sqrt(v.x ** 2 + v.y ** 2 + v.z ** 2)) + " " +
str(t.location.x) + " " + str(
    t.location.y) + " " + str(t.location.z) + " " +
str(world.gnss_sensor.lat) + " " + str(
    world.gnss_sensor.lon) + " " + str(world.imu_sensor.accelerometer[0]) + "
" + str(
    world.imu_sensor.accelerometer[1]) + " " +
str(world.imu_sensor.accelerometer[2]) + " " + str(
    world.imu_sensor.gyroscope[0]) + " " + str(world.imu_sensor.gyroscope[1])
+ " " + str(
    world.imu_sensor.gyroscope[2]) + " " + str(world.imu_sensor.compass) + "
" + str(
    float(c.throttle)) + " " + str(float(c.steer)) + " " +
str(float(c.brake)))
    datos.close()

    if isinstance(c, carla.VehicleControl):
        self._info_text += [
            ('Throttle:', c.throttle, 0.0, 1.0),
            ('Steer:', c.steer, -1.0, 1.0),
            ('Brake:', c.brake, 0.0, 1.0),
            ('Reverse:', c.reverse),
            ('Hand brake:', c.hand_brake),
            ('Manual:', c.manual_gear_shift),
            'Gear:        %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)]
    elif isinstance(c, carla.WalkerControl):
        self._info_text += [
            ('Speed:', c.speed, 0.0, 5.556),
            ('Jump:', c.jump)]
    self._info_text += [
        '',
        'Collision:',
        collision,
        ''
    ]

```

```

        'Number of vehicles: % 8d' % len(vehicles)]

    if len(vehicles) > 1:
        self._info_text += ['Nearby vehicles:']
        distance = lambda l: math.sqrt(
            (l.x - t.location.x) ** 2 + (l.y - t.location.y) ** 2 + (l.z -
            t.location.z) ** 2)
        vehicles = [(distance(x.get_location()), x) for x in vehicles if x.id !=
        world.player.id]
        for d, vehicle in sorted(vehicles):
            if d > 200.0:
                break
            vehicle_type = get_actor_display_name(vehicle, truncate=22)
            self._info_text.append('% 4dm %s' % (d, vehicle_type))

    def toggle_info(self):
        self._show_info = not self._show_info

    def notification(self, text, seconds=2.0):
        self._notifications.set_text(text, seconds=seconds)

    def error(self, text):
        self._notifications.set_text('Error: %s' % text, (255, 0, 0))

    def render(self, display):
        if self._show_info:
            info_surface = pygame.Surface((220, self.dim[1]))
            info_surface.set_alpha(100)
            display.blit(info_surface, (0, 0))
            v_offset = 4
            bar_h_offset = 100
            bar_width = 106
            for item in self._info_text:
                if v_offset + 18 > self.dim[1]:
                    break
                if isinstance(item, list):
                    if len(item) > 1:
                        points = [(x + 8, v_offset + 8 + (1.0 - y) * 30) for x, y in
enumerate(item)]
                        pygame.draw.lines(display, (255, 136, 0), False, points, 2)
                        item = None
                        v_offset += 18
                    elif isinstance(item, tuple):
                        if isinstance(item[1], bool):
                            rect = pygame.Rect((bar_h_offset, v_offset + 8), (6, 6))
                            pygame.draw.rect(display, (255, 255, 255), rect, 0 if item[1]
else 1)
                            else:
                                rect_border = pygame.Rect((bar_h_offset, v_offset + 8),
(bar_width, 6))
                                pygame.draw.rect(display, (255, 255, 255), rect_border, 1)
                                f = (item[1] - item[2]) / (item[3] - item[2])
                                if item[2] < 0.0:
                                    rect = pygame.Rect((bar_h_offset + f * (bar_width - 6),
v_offset + 8), (6, 6))
                                else:
                                    rect = pygame.Rect((bar_h_offset, v_offset + 8), (f *
bar_width, 6))
                                pygame.draw.rect(display, (255, 255, 255), rect)
                                item = item[0]
                if item: # At this point has to be a str.
                    surface = self._font_mono.render(item, True, (255, 255, 255))
                    display.blit(surface, (8, v_offset))
                    v_offset += 18
            self._notifications.render(display)
            self.help.render(display)

# =====
# -- FadingText -----
# =====

class FadingText(object):
    def __init__(self, font, dim, pos):

```

```

        self.font = font
        self.dim = dim
        self.pos = pos
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)

    def set_text(self, text, color=(255, 255, 255), seconds=2.0):
        text_texture = self.font.render(text, True, color)
        self.surface = pygame.Surface(self.dim)
        self.seconds_left = seconds
        self.surface.fill((0, 0, 0, 0))
        self.surface.blit(text_texture, (10, 11))

    def tick(self, _, clock):
        delta_seconds = 1e-3 * clock.get_time()
        self.seconds_left = max(0.0, self.seconds_left - delta_seconds)
        self.surface.set_alpha(500.0 * self.seconds_left)

    def render(self, display):
        display.blit(self.surface, self.pos)

# =====
# -- HelpText -----
# =====

class HelpText(object):
    def __init__(self, font, width, height):
        lines = __doc__.split('\n')
        self.font = font
        self.dim = (680, len(lines) * 22 + 12)
        self.pos = (0.5 * width - 0.5 * self.dim[0], 0.5 * height - 0.5 *
self.dim[1])
        self.seconds_left = 0
        self.surface = pygame.Surface(self.dim)
        self.surface.fill((0, 0, 0, 0))
        for n, line in enumerate(lines):
            text_texture = self.font.render(line, True, (255, 255, 255))
            self.surface.blit(text_texture, (22, n * 22))
            self._render = False
        self.surface.set_alpha(220)

    def toggle(self):
        self._render = not self._render

    def render(self, display):
        if self._render:
            display.blit(self.surface, self.pos)

# =====
# -- CollisionSensor -----
# =====

class CollisionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self.history = []
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.collusion')
        self.sensor = world.spawn_actor(bp, carla.Transform(),
attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: CollisionSensor._on_collision(weak_self,
event))

    def get_collision_history(self):
        history = collections.defaultdict(int)
        for frame, intensity in self.history:

```

```

        history[frame] += intensity
    return history

    @staticmethod
    def _on_collision(weak_self, event):
        self = weak_self()
        if not self:
            return
        actor_type = get_actor_display_name(event.other_actor)
        self.hud.notification('Collision with %r' % actor_type)
        impulse = event.normal_impulse
        intensity = math.sqrt(impulse.x ** 2 + impulse.y ** 2 + impulse.z ** 2)
        self.history.append((event.frame, intensity))
        if len(self.history) > 4000:
            self.history.pop(0)
        collisiondata = open("collisiondata.txt", "a")
        collisiondata.write('\n' + str(impulse.x) + " " + str(impulse.y) + " " +
str(impulse.z) + " " + str(
            intensity) + " " + 'Collision with %r' % actor_type)
        collisiondata.close()

# =====
# -- LaneInvasionSensor -----
# =====

class LaneInvasionSensor(object):
    def __init__(self, parent_actor, hud):
        self.sensor = None
        self._parent = parent_actor
        self.hud = hud
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.lane_invasion')
        self.sensor = world.spawn_actor(bp, carla.Transform(),
attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: LaneInvasionSensor._on_invasion(weak_self,
event))

    @staticmethod
    def _on_invasion(weak_self, event):
        self = weak_self()
        if not self:
            return
        lane_types = set(x.type for x in event.crossed_lane_markings)
        text = ['%r' % str(x).split()[-1] for x in lane_types]
        self.hud.notification('Crossed line %s' % ' and '.join(text))
        lanedata = open("lanedata.txt", "a")
        lanedata.write('\n' + 'Crossed line %s' % ' and '.join(text))
        lanedata.close()

# =====
# -- GnssSensor -----
# =====

class GnssSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.lat = 0.0
        self.lon = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.gnss')
        self.sensor = world.spawn_actor(bp, carla.Transform(carla.Location(x=1.0,
z=2.8)), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda event: GnssSensor._on_gnss_event(weak_self, event))

```

```

@staticmethod
def _on_gnss_event(weak_self, event):
    self = weak_self()
    if not self:
        return
    self.lat = event.latitude
    self.lon = event.longitude
    gnssdata = open("gnssdata.txt", "a")
    gnssdata.write('\n' + str(event.latitude) + " " + str(event.longitude))
    gnssdata.close()

# =====
# -- IMUSensor -----
# =====

class IMUSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.accelerometer = ()
        self.gyroscope = ()
        self.compass = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.other.imu')
        self.sensor = world.spawn_actor(
            bp, carla.Transform(), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(
            lambda sensor_data: IMUSensor._IMU_callback(weak_self, sensor_data))

    @staticmethod
    def _IMU_callback(weak_self, sensor_data):
        self = weak_self()
        if not self:
            return
        limits = (-99.9, 99.9)
        self.accelerometer = (
            max(limits[0], min(limits[1], sensor_data.accelerometer.x)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.y)),
            max(limits[0], min(limits[1], sensor_data.accelerometer.z)))
        self.gyroscope = (
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.x))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.y))),
            max(limits[0], min(limits[1], math.degrees(sensor_data.gyroscope.z))))
        self.compass = math.degrees(sensor_data.compass)
        imudatag = open("imudatag.txt", "a")
        imudatag.write('\n' + str(sensor_data.gyroscope.x) + " " +
            str(sensor_data.gyroscope.y) + " " + str(
                sensor_data.gyroscope.z))
        imudatag.close()
        imudataa = open("imudataa.txt", "a")
        imudataa.write('\n' + str(sensor_data.accelerometer.x) + " " +
            str(sensor_data.accelerometer.y) + " " + str(
                sensor_data.accelerometer.z))
        imudataa.close()

# =====
# -- RadarSensor -----
# =====

class RadarSensor(object):
    def __init__(self, parent_actor):
        self.sensor = None
        self._parent = parent_actor
        self.velocity_range = 7.5 # m/s
        world = self._parent.get_world()
        self.debug = world.debug
        bp = world.get_blueprint_library().find('sensor.other.radar')
        bp.set_attribute('horizontal_fov', str(35))

```

```

bp.set_attribute('vertical_fov', str(20))
self.sensor = world.spawn_actor(
    bp,
    carla.Transform(
        carla.Location(x=2.8, z=1.0),
        carla.Rotation(pitch=5)),
    attach_to=self._parent)
# We need a weak reference to self to avoid circular reference.
weak_self = weakref.ref(self)
self.sensor.listen(
    lambda radar_data: RadarSensor._Radar_callback(weak_self, radar_data))

@staticmethod
def _Radar_callback(weak_self, radar_data):
    self = weak_self()
    if not self:
        return
    # To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:
    # points = np.frombuffer(radar_data.raw_data, dtype=np.dtype('f4'))
    # points = np.reshape(points, (len(radar_data), 4))

    current_rot = radar_data.transform.rotation
    for detect in radar_data:
        azi = math.degrees(detect.azimuth)
        alt = math.degrees(detect.altitude)
        # The 0.25 adjusts a bit the distance so the dots can
        # be properly seen
        fw_vec = carla.Vector3D(x=detect.depth - 0.25)
        carla.Transform(
            carla.Location(),
            carla.Rotation(
                pitch=current_rot.pitch + alt,
                yaw=current_rot.yaw + azi,
                roll=current_rot.roll)).transform(fw_vec)

        def clamp(min_v, max_v, value):
            return max(min_v, min(value, max_v))

        norm_velocity = detect.velocity / self.velocity_range # range [-1, 1]
        r = int(clamp(0.0, 1.0, 1.0 - norm_velocity) * 255.0)
        g = int(clamp(0.0, 1.0, 1.0 - abs(norm_velocity)) * 255.0)
        b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm_velocity)) * 255.0)
        self.debug.draw_point(
            radar_data.transform.location + fw_vec,
            size=0.075,
            life_time=0.06,
            persistent_lines=False,
            color=carla.Color(r, g, b))
        radardata = open("radardata.txt", "a")
        radardata.write(
            '\n' + str(azi) + " " + str(alt) + " " + str(detect.depth) + " " +
            str(detect.velocity) + " " + str(
                norm_velocity))
        radardata.close()

# =====
# -- CameraManager -----
# =====

class CameraManager(object):
    def __init__(self, parent_actor, hud, gamma_correction):
        self.sensor = None
        self.surface = None
        self._parent = parent_actor
        self.hud = hud
        self.recording = False
        bound_y = 0.5 + self._parent.bounding_box.extent.y
        Attachment = carla.AttachmentType
        self._camera_transforms = [
            (carla.Transform(carla.Location(x=-5.5, z=2.5),
            carla.Rotation(pitch=8.0)), Attachment.SpringArm),
            (carla.Transform(carla.Location(x=1.6, z=1.7)), Attachment.Rigid),
            (carla.Transform(carla.Location(x=5.5, y=1.5, z=1.5)),

```

```

Attachment.SpringArm),
    (carla.Transform(carla.Location(x=-8.0, z=6.0),
carla.Rotation(pitch=6.0)), Attachment.SpringArm),
    (carla.Transform(carla.Location(x=-1, y=-bound_y, z=0.5)),
Attachment.Rigid)]
self.transform_index = 1
self.sensors = [
    ['sensor.camera.rgb', cc.Raw, 'Camera RGB', {}],
    ['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)', {}],
    ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)', {}],
    ['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth (Logarithmic
Gray Scale)', {}],
    ['sensor.camera.semantic_segmentation', cc.Raw, 'Camera Semantic
Segmentation (Raw)', {}],
    ['sensor.camera.semantic_segmentation', cc.CityScapesPalette,
'Camera Semantic Segmentation (CityScapes Palette)', {}],
    ['sensor.lidar.ray_cast', None, 'Lidar (Ray-Cast)', {}],
    ['sensor.camera.rgb', cc.Raw, 'Camera RGB Distorted',
{'lens_circle_multiplier': '3.0',
'lens_circle_falloff': '3.0',
'chromatic_aberration_intensity': '0.5',
'chromatic_aberration_offset': '0'}]]
world = self._parent.get_world()
bp_library = world.get_blueprint_library()
for item in self.sensors:
    bp = bp_library.find(item[0])
    if item[0].startswith('sensor.camera'):
        bp.set_attribute('image_size_x', str(hud.dim[0]))
        bp.set_attribute('image_size_y', str(hud.dim[1]))
        if bp.has_attribute('gamma'):
            bp.set_attribute('gamma', str(gamma_correction))
            for attr_name, attr_value in item[3].items():
                bp.set_attribute(attr_name, attr_value)
    elif item[0].startswith('sensor.lidar'):
        bp.set_attribute('range', '50')
    item.append(bp)
self.index = None

def toggle_camera(self):
    self.transform_index = (self.transform_index + 1) %
len(self._camera_transforms)
    self.set_sensor(self.index, notify=False, force_respawn=True)

def set_sensor(self, index, notify=True, force_respawn=False):
    index = index % len(self.sensors)
    needs_respawn = True if self.index is None else \
        (force_respawn or (self.sensors[index][2] !=
self.sensors[self.index][2]))
    if needs_respawn:
        if self.sensor is not None:
            self.sensor.destroy()
            self.surface = None
        self.sensor = self._parent.get_world().spawn_actor(
            self.sensors[index][-1],
            self._camera_transforms[self.transform_index][0],
            attach_to=self._parent,
            attachment_type=self._camera_transforms[self.transform_index][1])
        # We need to pass the lambda a weak reference to self to avoid
        # circular reference.
        weak_self = weakref.ref(self)
        self.sensor.listen(lambda image: CameraManager._parse_image(weak_self,
image))
    if notify:
        self.hud.notification(self.sensors[index][2])
        self.index = index

def next_sensor(self):
    self.set_sensor(self.index + 1)

def toggle_recording(self):
    self.recording = not self.recording
    self.hud.notification('Recording %s' % ('On' if self.recording else 'Off'))

def render(self, display):
    if self.surface is not None:

```

```

        display.blit(self.surface, (0, 0))

    @staticmethod
    def _parse_image(weak_self, image):
        self = weak_self()
        if not self:
            return
        if self.sensors[self.index][0].startswith('sensor.lidar'):
            points = np.frombuffer(image.raw_data, dtype=np.dtype('f4'))
            points = np.reshape(points, (int(points.shape[0] / 3), 3))
            lidar_data = np.array(points[:, :2])
            lidar_data *= min(self.hud.dim) / 100.0
            lidar_data += (0.5 * self.hud.dim[0], 0.5 * self.hud.dim[1])
            lidar_data = np.fabs(lidar_data) # pylint: disable=E1111
            lidar_data = lidar_data.astype(np.int32)
            lidar_data = np.reshape(lidar_data, (-1, 2))
            lidar_img_size = (self.hud.dim[0], self.hud.dim[1], 3)
            lidar_img = np.zeros((lidar_img_size), dtype=int)
            lidar_img[tuple(lidar_data.T)] = (255, 255, 255)
            self.surface = pygame.surfarray.make_surface(lidar_img)
        else:
            image.convert(self.sensors[self.index][1])
            array = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
            array = np.reshape(array, (image.height, image.width, 4))
            array = array[:, :, :3]
            array = array[:, :, :-1]
            self.surface = pygame.surfarray.make_surface(array.swapaxes(0, 1))
        if self.recording:
            image.save_to_disk('camara/%08d' % image.frame)
            # image.save_to_disk('_palette/%08d' % image.frame,)
            # self.sensor.listen(lambda LidarMeasurement:
            LidarMeasurement.save_to_disk('_lidar/%06d.ply' % LidarMeasurement.frame_number))

# =====
# -- game_loop() -----
# =====

def game_loop(args):
    pygame.init()
    pygame.font.init()
    world = None

    try:
        client = carla.Client(args.host, args.port)
        client.set_timeout(2.0)

        display = pygame.display.set_mode(
            (args.width, args.height),
            pygame.HWSURFACE | pygame.DOUBLEBUF)

        hud = HUD(args.width, args.height)
        world = World(client.get_world(), hud, args)
        controller = KeyboardControl(world, args.autopilot)

        clock = pygame.time.Clock()
        while True:
            clock.tick_busy_loop(60)
            if controller.parse_events(client, world, clock):
                return
            world.tick(clock)
            world.render(display)
            pygame.display.flip()

    finally:
        if (world and world.recording_enabled):
            client.stop_recorder()

        if world is not None:
            world.destroy()

    pygame.quit()

```

```

# =====
# -- main() -----
# =====

def main():
    argparser = argparse.ArgumentParser(
        description='CARLA Manual Control Client')
    argparser.add_argument(
        '-v', '--verbose',
        action='store_true',
        dest='debug',
        help='print debug information')
    argparser.add_argument(
        '--host',
        metavar='H',
        default='127.0.0.1',
        help='IP of the host server (default: 127.0.0.1)')
    argparser.add_argument(
        '-p', '--port',
        metavar='P',
        default=2000,
        type=int,
        help='TCP port to listen to (default: 2000)')
    argparser.add_argument(
        '-a', '--autopilot',
        action='store_true',
        help='enable autopilot')
    argparser.add_argument(
        '--res',
        metavar='WIDTHxHEIGHT',
        default='1280x720',
        help='window resolution (default: 1280x720)')
    argparser.add_argument(
        '--filter',
        metavar='PATTERN',
        default='vehicle.*',
        help='actor filter (default: "vehicle.*")')
    argparser.add_argument(
        '--rolename',
        metavar='NAME',
        default='hero',
        help='actor role name (default: "hero")')
    argparser.add_argument(
        '--gamma',
        default=2.2,
        type=float,
        help='Gamma correction of the camera (default: 2.2)')
    args = argparser.parse_args()

    args.width, args.height = [int(x) for x in args.res.split('x')]

    log_level = logging.DEBUG if args.debug else logging.INFO
    logging.basicConfig(format='%(levelname)s: %(message)s', level=log_level)

    logging.info('listening to server %s:%s', args.host, args.port)

    print(__doc__)

    try:
        game_loop(args)
    except KeyboardInterrupt:
        print('\nCancelled by user. Bye!')

if __name__ == '__main__':
    main()

```

Código 13. Programa manual_control.py

A.3. WheeledVehicleAIController.cpp

```
// Copyright (c) 2017 Computer Vision Center (CVC) at the Universitat Autònoma
// de Barcelona (UAB).
//
// This work is licensed under the terms of the MIT license.
// For a copy, see <https://opensource.org/licenses/MIT>.

#include "Carla.h"
#include "WheeledVehicleAIController.h"

#include "MapGen/RoadMap.h"
#include "Vehicle/CarlaWheeledVehicle.h"

#include "EngineUtils.h"
#include "GameFramework/Pawn.h"
#include "WheeledVehicleMovementComponent.h"

// =====
// -- Static local methods -----
// =====

static bool RayTrace(const AActor &Actor, const FVector &Start, const FVector &End) {
    FHitResult OutHit;
    static FName TraceTag = FName(TEXT("VehicleTrace"));
    FCollisionQueryParams CollisionParams(TraceTag, true);
    CollisionParams.AddIgnoredActor(&Actor);

    const bool Success = Actor.GetWorld()->LineTraceSingleByObjectType(
        OutHit,
        Start,
        End,
        FCollisionObjectQueryParams(FCollisionObjectQueryParams::AllDynamicObjects),
        CollisionParams);

    return Success && OutHit.bBlockingHit;
}

static bool IsThereAnObstacleAhead(
    const ACarlaWheeledVehicle &Vehicle,
    const float Speed,
    const FVector &Direction)
{
    const auto ForwardVector = Vehicle.GetVehicleOrientation();
    const auto VehicleBounds = Vehicle.GetVehicleBoundingBoxExtent();

    const float Distance = std::max(50.0f, Speed * Speed); // why?

    const FVector StartCenter = Vehicle.GetActorLocation() + (ForwardVector * (250.0f +
        VehicleBounds.X / 2.0f)) + FVector(0.0f, 0.0f, 50.0f);
    const FVector EndCenter = StartCenter + Direction * (Distance + VehicleBounds.X /
        2.0f);

    const FVector StartRight = StartCenter + (FVector(ForwardVector.Y, -
        ForwardVector.X, ForwardVector.Z) * 100.0f);
    const FVector EndRight = StartRight + Direction * (Distance + VehicleBounds.X /
        2.0f);

    const FVector StartLeft = StartCenter + (FVector(-ForwardVector.Y, ForwardVector.X,
        ForwardVector.Z) * 100.0f);
    const FVector EndLeft = StartLeft + Direction * (Distance + VehicleBounds.X /
        2.0f);

    return
        RayTrace(Vehicle, StartCenter, EndCenter) ||
        RayTrace(Vehicle, StartRight, EndRight) ||
        RayTrace(Vehicle, StartLeft, EndLeft);
}

template <typename T>
```

```

static void ClearQueue(std::queue<T> &Queue)
{
    std::queue<T> EmptyQueue;
    Queue.swap(EmptyQueue);
}

// =====
// -- Constructor and destructor -----
// =====

AWheeledVehicleAIController::AWheeledVehicleAIController(const FObjectInitializer&
ObjectInitializer) :
    Super(ObjectInitializer)
{
    RandomEngine = CreateDefaultSubobject<URandomEngine>(TEXT("RandomEngine"));

    RandomEngine->Seed(RandomEngine->GenerateRandomSeed());

    PrimaryActorTick.bCanEverTick = true;
    PrimaryActorTick.TickGroup = TG_PrePhysics;
}

AWheeledVehicleAIController::~~AWheeledVehicleAIController() {}

// =====
// -- APlayerController -----
// =====

void AWheeledVehicleAIController::Possess(APawn *aPawn)
{
    Super::Possess(aPawn);

    if (IsPossessingAVehicle())
    {
        UE_LOG(LogCarla, Error, TEXT("Controller already possessing a vehicle!"));
        return;
    }
    Vehicle = Cast<ACarlaWheeledVehicle>(aPawn);
    check(Vehicle != nullptr);
    MaximumSteerAngle = Vehicle->GetMaximumSteerAngle();
    check(MaximumSteerAngle > 0.0f);
    ConfigureAutopilot(bAutopilotEnabled);

    if (RoadMap == nullptr)
    {
        TActorIterator<ACityMapGenerator> It(GetWorld());
        RoadMap = (It ? It->GetRoadMap() : nullptr);
    }
}

void AWheeledVehicleAIController::UnPossess()
{
    Super::UnPossess();

    Vehicle = nullptr;
}

void AWheeledVehicleAIController::Tick(const float DeltaTime)
{
    Super::Tick(DeltaTime);

    if (!IsPossessingAVehicle())
    {
        return;
    }

    TickAutopilotController();

    if (bAutopilotEnabled)
    {
        Vehicle->ApplyVehicleControl(AutopilotControl);
    }
}

// =====

```

```

// -- Autopilot -----
// =====

void AWheeledVehicleAIController::ConfigureAutopilot(const bool Enable)
{
    bAutopilotEnabled = Enable;
    // Reset state.
    Vehicle->SetSteeringInput(0.0f);
    Vehicle->SetThrottleInput(0.0f);
    Vehicle->SetBrakeInput(0.0f);
    Vehicle->SetReverse(false);
    Vehicle->SetHandbrakeInput(false);
    TrafficLightState = ETrafficLightState::Green;
    ClearQueue(TargetLocations);
    Vehicle->SetAIVehicleState(
        bAutopilotEnabled ?
            ECarlaWheeledVehicleState::FreeDriving :
            ECarlaWheeledVehicleState::AutopilotOff);
}

// =====
// -- Traffic -----
// =====

void AWheeledVehicleAIController::SetFixedRoute(
    const TArray<FVector> &Locations,
    const bool bOverwriteCurrent)
{
    if (bOverwriteCurrent) {
        ClearQueue(TargetLocations);
    }
    for (auto &Location : Locations) {
        TargetLocations.emplace(Location);
    }
}

// =====
// -- AI -----
// =====

void AWheeledVehicleAIController::TickAutopilotController()
{
    #if WITH_EDITOR
    if (Vehicle == nullptr) { // This happens in simulation mode in editor.
        bAutopilotEnabled = false;
        return;
    }
    #endif // WITH_EDITOR

    check(Vehicle != nullptr);

    if (RoadMap == nullptr) {
        return;
    }

    FVector Direction;

    float Steering;
    if (!TargetLocations.empty()) {
        Steering = GoToNextTargetLocation(Direction);
    } else {
        Steering = CalcSteeringValue(Direction);
    }

    // Speed in km/h.
    const auto Speed = Vehicle->GetVehicleForwardSpeed() * 0.036f;

    float Throttle;
    if (TrafficLightState != ETrafficLightState::Green) {
        Vehicle->SetAIVehicleState(ECarlaWheeledVehicleState::WaitingForRedLight);
        Throttle = Stop(Speed);
    } else if (IsThereAnObstacleAhead(*Vehicle, Speed, Direction)) {
        Vehicle->SetAIVehicleState(ECarlaWheeledVehicleState::ObstacleAhead);
        Throttle = Stop(Speed);
    } else {

```

```

        Throttle = Move(Speed);
    }

    if (Throttle < 0.001f) {
        AutopilotControl.Brake = 1.0f;
        AutopilotControl.Throttle = 0.0f;
    } else {
        AutopilotControl.Brake = 0.0f;
        AutopilotControl.Throttle = Throttle;
    }
    AutopilotControl.Steer = Steering;
}

float AWheeledVehicleAIController::GoToNextTargetLocation(FVector &Direction)
{
    // Get middle point between the two front wheels.
    const auto CurrentLocation = [&]() {
        const auto &Wheels = Vehicle->GetVehicleMovementComponent()->Wheels;
        check((Wheels.Num() > 1) && (Wheels[0u] != nullptr) && (Wheels[1u] != nullptr));
        return (Wheels[0u]->Location + Wheels[1u]->Location) / 2.0f;
    }();

    const auto Target = [&]() {
        const auto &Result = TargetLocations.front();
        return FVector{Result.X, Result.Y, CurrentLocation.Z};
    }();

    if (Target.Equals(CurrentLocation, 200.0f)) {
        TargetLocations.pop();
        if (!TargetLocations.empty()) {
            return GoToNextTargetLocation(Direction);
        } else {
            return CalcSteeringValue(Direction);
        }
    }

    Direction = (Target - CurrentLocation).GetSafeNormal();

    const FVector &Forward = GetPawn()->GetActorForwardVector();

    float dirAngle = Direction.UnitCartesianToSpherical().Y;
    float actorAngle = Forward.UnitCartesianToSpherical().Y;

    dirAngle *= (180.0f / PI);
    actorAngle *= (180.0 / PI);

    float angle = dirAngle - actorAngle;

    if (angle > 180.0f) { angle -= 360.0f; } else if (angle < -180.0f) {
        angle += 360.0f;
    }

    float Steering = 0.0f;
    if (angle < -MaximumSteerAngle) {
        Steering = -1.0f;
    } else if (angle > MaximumSteerAngle) {
        Steering = 1.0f;
    } else {
        Steering += angle / MaximumSteerAngle;
    }

    Vehicle->SetAIVehicleState(ETCarlaWheeledVehicleState::FollowingFixedRoute);
    return Steering;
}

float AWheeledVehicleAIController::CalcSteeringValue(FVector &direction)
{
    float steering = 0;
    FVector BoxExtent = Vehicle->GetVehicleBoundingBoxExtent();
    FVector forward = Vehicle->GetActorForwardVector();

    FVector rightSensorPosition(BoxExtent.X / 2.0f, (BoxExtent.Y / 2.0f) + 100.0f,
0.0f);
    FVector leftSensorPosition(BoxExtent.X / 2.0f, -(BoxExtent.Y / 2.0f) - 100.0f,
0.0f);

```

```

float forwardMagnitude = BoxExtent.X / 2.0f;

float Magnitude = (float) sqrt(pow((double) leftSensorPosition.X, 2.0) +
pow((double) leftSensorPosition.Y, 2.0));

//same for the right and left
float offset = FGenericPlatformMath::Acos(forwardMagnitude / Magnitude);

float actorAngle = forward.UnitCartesianToSpherical().Y;

float sinR = FGenericPlatformMath::Sin(actorAngle + offset);
float cosR = FGenericPlatformMath::Cos(actorAngle + offset);

float sinL = FGenericPlatformMath::Sin(actorAngle - offset);
float cosL = FGenericPlatformMath::Cos(actorAngle - offset);

rightSensorPosition.Y = sinR * Magnitude;
rightSensorPosition.X = cosR * Magnitude;

leftSensorPosition.Y = sinL * Magnitude;
leftSensorPosition.X = cosL * Magnitude;

FVector rightPositon = GetPawn()->GetActorLocation() +
FVector(rightSensorPosition.X, rightSensorPosition.Y, 0.0f);
FVector leftPosition = GetPawn()->GetActorLocation() +
FVector(leftSensorPosition.X, leftSensorPosition.Y, 0.0f);

FRoadMapPixelData rightRoadData = RoadMap->GetDataAt(rightPositon);
if (!rightRoadData.IsRoad()) { steering -= 0.2f;}

FRoadMapPixelData leftRoadData = RoadMap->GetDataAt(leftPosition);
if (!leftRoadData.IsRoad()) { steering += 0.2f;}

FRoadMapPixelData roadData = RoadMap->GetDataAt(GetPawn()->GetActorLocation());
if (!roadData.IsRoad()) {
    steering = 0.0f;
} else if (roadData.HasDirection()) {

    direction = roadData.GetDirection();
    FVector right = rightRoadData.GetDirection();
    FVector left = leftRoadData.GetDirection();

    forward.Z = 0.0f;

    float dirAngle = direction.UnitCartesianToSpherical().Y;
    float rightAngle = right.UnitCartesianToSpherical().Y;
    float leftAngle = left.UnitCartesianToSpherical().Y;

    dirAngle *= (180.0f / PI);
    rightAngle *= (180.0 / PI);
    leftAngle *= (180.0 / PI);
    actorAngle *= (180.0 / PI);

    float min = dirAngle - 90.0f;
    if (min < -180.0f) { min = 180.0f + (min + 180.0f);}

    float max = dirAngle + 90.0f;
    if (max > 180.0f) { max = -180.0f + (max - 180.0f);}

    if (dirAngle < -90.0 || dirAngle > 90.0) {
        if (rightAngle < min && rightAngle > max) { steering -= 0.2f;}
        if (leftAngle < min && leftAngle > max) { steering += 0.2f;}
    } else {
        if (rightAngle < min || rightAngle > max) { steering -= 0.2f;}
        if (leftAngle < min || leftAngle > max) { steering += 0.2f;}
    }

    float angle = dirAngle - actorAngle;

    if (angle > 180.0f) { angle -= 360.0f;} else if (angle < -180.0f) {
        angle += 360.0f;
    }

    if (angle < -MaximumSteerAngle) {

```

```

        steering = -1.0f;
    } else if (angle > MaximumSteerAngle) {
        steering = 1.0f;
    } else {
        steering += angle / MaximumSteerAngle;
    }
}

Vehicle->SetAIVehicleState(ETCarlaWheeledVehicleState::FreeDriving);
return steering;
}

float AWheeledVehicleAIController::Stop(const float Speed) {
    return (Speed >= 1.0f ? -Speed / SpeedLimit : 0.0f);
}

float AWheeledVehicleAIController::Move(const float Speed) {
    if (Speed >= SpeedLimit) {
        return Stop(Speed);
    } else if (Speed >= SpeedLimit - 10.0f) {
        return 0.5f;
    } else {
        return 1.0f;
    }
}

```

Código 14. Control autónomo del vehículo [25]

A.4. RoutePlanner.cpp

```

// Copyright (c) 2017 Computer Vision Center (CVC) at the Universitat Autònoma
// de Barcelona (UAB).
//
// This work is licensed under the terms of the MIT license.
// For a copy, see <https://opensource.org/licenses/MIT>.

#include "Carla.h"
#include "RoutePlanner.h"

#include "Util/RandomEngine.h"
#include "Vehicle/CarlaWheeledVehicle.h"
#include "Vehicle/WheeledVehicleAIController.h"

#include "Engine/CollisionProfile.h"
#include "DrawDebugHelpers.h"

static bool IsSplineValid(const USplineComponent *SplineComponent)
{
    return (SplineComponent != nullptr) &&
        (SplineComponent->GetNumberOfSplinePoints() > 1);
}

static AWheeledVehicleAIController *GetVehicleController(AActor *Actor)
{
    auto *Vehicle = (Actor->IsPendingKill() ? nullptr :
        Cast<ACarlaWheeledVehicle>(Actor));
    return (Vehicle != nullptr ?
        Cast<AWheeledVehicleAIController>(Vehicle->GetController()) :
        nullptr);
}

static const USplineComponent *PickARoute(
    URandomEngine &RandomEngine,
    const TArray<USplineComponent*> &Routes,
    const TArray<float> &Probabilities)
{
    check(Routes.Num() > 0);
}

```

```

    if (Routes.Num() == 1)
    {
        return Routes[0];
    }

    auto Index = RandomEngine.GetIntWithWeight(Probabilities);
    check((Index >= 0) && (Index < Routes.Num()));
    return Routes[Index];
}

ARoutePlanner::ARoutePlanner(const FObjectInitializer &ObjectInitializer)
: Super(ObjectInitializer)
{
    RootComponent =
        ObjectInitializer.CreateDefaultSubobject<USceneComponent>(this,
TEXT("SceneRootComponent"));
    RootComponent->SetMobility(ECComponentMobility::Static);

    TriggerVolume = CreateDefaultSubobject<UBoxComponent>(TEXT("TriggerVolume"));
    TriggerVolume->SetupAttachment(RootComponent);
    TriggerVolume->SetHiddenInGame(true);
    TriggerVolume->SetMobility(ECComponentMobility::Static);
    TriggerVolume->SetCollisionProfileName(FName("OverlapAll"));
    TriggerVolume->SetGenerateOverlapEvents(true);

    // Do not change default value here, our autopilot depends on this.
    TriggerVolume->SetBoxExtent(FVector{32.0f, 32.0f, 32.0f});
}

void ARoutePlanner::BeginDestroy()
{
    CleanRoute();
    Super::BeginDestroy();
}

#ifdef WITH_EDITOR
void ARoutePlanner::PostEditChangeProperty(FPropertyChangedEvent
&PropertyChangedEvent)
{
    Super::PostEditChangeProperty(PropertyChangedEvent);
    const auto Size = Routes.Num();
    if (PropertyChangedEvent.Property && (Size != Probabilities.Num()))
    {
        Probabilities.Reset(Size);
        for (auto i = 0; i < Size; ++i)
        {
            Probabilities.Add(1.0f / static_cast<float>(Size));
            if (Routes[i] == nullptr)
            {
                Routes[i] = NewObject<USplineComponent>(this);
                Routes[i]->SetupAttachment(RootComponent);
                Routes[i]->SetHiddenInGame(true);
                Routes[i]->SetMobility(ECComponentMobility::Static);
                Routes[i]->RegisterComponent();
            }
        }
    }
}
#endif // WITH_EDITOR

void ARoutePlanner::AddRoute(float probability, const TArray<FVector> &routePoints)
{
    USplineComponent *NewSpline = NewObject<USplineComponent>(this);
    NewSpline->bHiddenInGame = true;

#ifdef WITH_EDITOR
    NewSpline->EditorUnselectedSplineSegmentColor = FLinearColor(1.f, 0.15f, 0.15f);
#endif // WITH_EDITOR

    NewSpline->SetLocationAtSplinePoint(0, routePoints[0],
ESplineCoordinateSpace::World, true);
    NewSpline->SetLocationAtSplinePoint(1, routePoints[1],
ESplineCoordinateSpace::World, true);

    for (int i = 2; i < routePoints.Num(); ++i)

```

```

    {
        NewSpline->AddSplinePoint(routePoints[i], ESplineCoordinateSpace::World, true);
    }

    Routes.Add(NewSpline);
    Probabilities.Add(probability);
}

void ARoutePlanner::CleanRoute()
{
    Routes.Empty();
    Probabilities.Empty();
}

void ARoutePlanner::AssignRandomRoute(AWheeledVehicleAIController &Controller) const
{
    if (!Controller.IsPendingKill() && (Controller.GetRandomEngine() != nullptr))
    {
        auto *RandomEngine = Controller.GetRandomEngine();
        auto *Route = PickARoute(*RandomEngine, Routes, Probabilities);

        TArray<FVector> WayPoints;
        const auto Size = Route->GetNumberOfSplinePoints();
        if (Size > 1)
        {
            WayPoints.Reserve(Size);
            for (auto i = 1; i < Size; ++i)
            {
                WayPoints.Add(Route->GetLocationAtSplinePoint(i,
                    ESplineCoordinateSpace::World));
            }

            Controller.SetFixedRoute(WayPoints);
        }
        else
        {
            UE_LOG(LogCarla, Error, TEXT("ARoutePlanner '%s' has a route with zero way-
points."), *GetName());
        }
    }
}

void ARoutePlanner::Init()
{
    if (Routes.Num() < 1)
    {
        UE_LOG(LogCarla, Warning, TEXT("ARoutePlanner '%s' has no route assigned."),
            *GetName());
        return;
    }

    for (auto &&Route : Routes)
    {
        if (!IsSplineValid(Route))
        {
            UE_LOG(LogCarla, Error, TEXT("ARoutePlanner '%s' has a route with zero way-
points."), *GetName());
            return;
        }
    }

    // Register delegate on begin overlap.
    if (!TriggerVolume->OnComponentBeginOverlap.IsAlreadyBound(this,
        &ARoutePlanner::OnTriggerBeginOverlap))
    {
        TriggerVolume->OnComponentBeginOverlap.AddDynamic(this,
            &ARoutePlanner::OnTriggerBeginOverlap);
    }
}

void ARoutePlanner::BeginPlay()
{
    Super::BeginPlay();
    Init();
}

```

```

}

void ARoutePlanner::EndPlay(const EEndPlayReason::Type EndPlayReason)
{
    // Deregister the delegate.
    if (TriggerVolume->OnComponentBeginOverlap.IsAlreadyBound(this,
&ARoutePlanner::OnTriggerBeginOverlap))
    {
        TriggerVolume->OnComponentBeginOverlap.RemoveDynamic(this,
&ARoutePlanner::OnTriggerBeginOverlap);
    }

    Super::EndPlay(EndPlayReason);
}

void ARoutePlanner::OnTriggerBeginOverlap(
    UPrimitiveComponent * /*OverlappedComp*/,
    AActor *OtherActor,
    UPrimitiveComponent * /*OtherComp*/,
    int32 /*OtherBodyIndex*/,
    bool /*bFromSweep*/,
    const FHitResult & /*SweepResult*/)
{
    auto *Controller = GetVehicleController(OtherActor);
    if (Controller != nullptr)
    {
        AssignRandomRoute(*Controller);
    }
}

void ARoutePlanner::DrawRoutes()
{
    #if WITH_EDITOR
        for (int i = 0, lenRoutes = Routes.Num(); i < lenRoutes; ++i)
        {
            for (int j = 0, lenNumPoints = Routes[i]->GetNumberOfSplinePoints() - 1; j <
lenNumPoints; ++j)
            {
                const FVector p0 = Routes[i]->GetLocationAtSplinePoint(j + 0,
ESplineCoordinateSpace::World);
                const FVector p1 = Routes[i]->GetLocationAtSplinePoint(j + 1,
ESplineCoordinateSpace::World);

                static const float MinThickness = 3.f;
                static const float MaxThickness = 15.f;

                const float Dist = (float) j / (float) lenNumPoints;
                const float OneMinusDist = 1.f - Dist;
                const float Thickness = OneMinusDist * MaxThickness + MinThickness;

                if (bIsIntersection)
                {
                    // from blue to black
                    DrawDebugLine(
                        GetWorld(), p0, p1, FColor(0, 0, 255 * OneMinusDist),
                        true, -1.f, 0, Thickness);
                }
                else
                {
                    // from green to black
                    DrawDebugLine(
                        GetWorld(), p0, p1, FColor(0, 255 * OneMinusDist, 0),
                        true, -1.f, 0, Thickness);
                }
            }
        }
    #endif
}

```

A.5. Código Matlab

```
%% Simulación Modo automático 1
clc
clear all

% Cargamos fichero con los datos y asignamos variables a cada columna de
% datos

file = load('datosprius.txt');
tiempo = file(:,1);
velocidad = file(:,2);
posx = file(:,3);
posy = file(:,4);
posz = file(:,5);
latitud = file(:,6);
longitud = file(:,7);
acelerometro_x = file(:,8);
acelerometro_y = file(:,9);
acelerometro_z = file(:,10);
giroscopio_x = file(:,11);
giroscopio_y = file(:,12);
giroscopio_z = file(:,13);
brujula = file(:,14);
aceleracion = file(:,15);
giro = file(:,16);
freno = file(:,17);

% Velocidad, pedal aceleración y aceleración lineal en X
figure;
subplot(3,1,1);
plot(tiempo,velocidad);
title('Velocidad','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Velocidad (Km/h)','Interpreter','Latex','FontSize',16);
grid on;
subplot(3,1,2);
plot(tiempo,aceleracion,tiempo,freno)
title('Pedales de aceleración y freno','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Aceleración y freno [0,1]','Interpreter','Latex','FontSize',16);
legend({'Aceleración','Freno'},'Interpreter','Latex','FontSize',14)
grid on;
subplot(3,1,3);
plot(tiempo,acelerometro_x);
title('Aceleración lineal en X','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Aceleración lineal X (m/s^2)','Interpreter','Latex','FontSize',16);
grid on;

% Posición, datos IMU completos y giro volante
figure;
subplot(4,1,1);
plot(tiempo,aceleracion,tiempo,freno)
title('Pedales de aceleración y freno','Interpreter','Latex','FontSize',20);
xlabel('Tiempo (s)','Interpreter','Latex','FontSize',16);
ylabel('Aceleración y freno [0,1]','Interpreter','Latex','FontSize',16);
legend({'Aceleración','Freno'},'Interpreter','Latex','FontSize',14)
grid on;
grid on;
subplot(4,1,2);
plot(tiempo,acelerometro_x,tiempo,acelerometro_y,tiempo,acelerometro_z);
title('Acelerómetro','Interpreter','Latex','FontSize',20);
```

```

xlabel('Tiempo (s)', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Aceleraci\on lineal
(m/s$^{2}$)', 'Interpreter', 'Latex', 'FontSize', 16);
legend({'X', 'Y', 'Z'}, 'Interpreter', 'Latex', 'FontSize', 14)
grid on;
subplot(4,1,3);
plot(tiempo,giroscopiox,tiempo,giroscopioy,tiempo,giroscopioz);
title('Giroscopio', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Tiempo (s)', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Movimiento rotacional (rad/s)', 'Interpreter', 'Latex', 'FontSize', 16);
legend({'X', 'Y', 'Z'}, 'Interpreter', 'Latex', 'FontSize', 14)
grid on;
subplot(4,1,4);
plot(tiempo,giro);
title('Intensidad de giro volante', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Tiempo (s)', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Giro volante [-1,1]', 'Interpreter', 'Latex', 'FontSize', 16);
grid on;

% Posicion mediante localizaci0n y latitud/longitud
figure;
subplot(2,1,1)
plot(posx,posy, 'b', 'LineWidth', 2.5);
title('Posici\on', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Posici\on X (m)', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Posici\on Y (m)', 'Interpreter', 'Latex', 'FontSize', 16);
grid on;
subplot(2,1,2)
plot(latitud,longitud, 'r', 'LineWidth', 2.5);
title('Latitud y longitud', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Latitud ($^{\circ}$)', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Longitud ($^{\circ}$)', 'Interpreter', 'Latex', 'FontSize', 16);
grid on;

figure;
plot3(posx, posy, posz, 'b', 'LineWidth', 2.5);
title('Recorrido 3D', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Posici\on X (m)', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Posici\on Y (m)', 'Interpreter', 'Latex', 'FontSize', 16);
zlabel('Posici\on Z (m)', 'Interpreter', 'Latex', 'FontSize', 16);
zlim([-2 20]);

grid on;

% Acelerometro y giroscopio

figure;
subplot(2,1,1)
plot(tiempo,acelerometrox,tiempo,acelerometroy,tiempo,acelerometroz);
title('Aceler\ometro', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Tiempo', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Aceleraci\on lineal
(m/s$^{2}$)', 'Interpreter', 'Latex', 'FontSize', 16);
legend({'X', 'Y', 'Z'}, 'Interpreter', 'Latex', 'FontSize', 14)
grid on;
subplot(2,1,2)
plot(tiempo,giroscopiox,tiempo,giroscopioy,tiempo,giroscopioz);
title('Giroscopio', 'Interpreter', 'Latex', 'FontSize', 20);
xlabel('Tiempo', 'Interpreter', 'Latex', 'FontSize', 16);
ylabel('Movimiento rotacional (rad/s)', 'Interpreter', 'Latex', 'FontSize', 16);
legend({'X', 'Y', 'Z'}, 'Interpreter', 'Latex', 'FontSize', 14)
grid on;

%% Radar

```

```

clc
clear all

fileradar = load('radardataprius.txt');
azi = fileradar(:,1);
alt = fileradar(:,2);
depth = fileradar(:,3);
velocidad = fileradar(:,4);
normvel = fileradar(:,5);

% Create some sample data:
x = azi/100; % X data
y = alt/100; % Y data
z = depth; % "Color" data
v = velocidad;

% Profundidad
figure;
scatter(x,y,[],z,'fill');
title('Profundidad (m)','Interpreter','Latex','FontSize',16);
xlabel('Azimut (rad)','Interpreter','Latex','FontSize',16);
ylabel('Altitud (rad)','Interpreter','Latex','FontSize',16);
xlim([-0.2 0.2]);
ylim([-0.2 0.2]);
colorbar;
% Profundidad, altitud y azimuth
figure;
plot(x,y,'o');
title('Radar','Interpreter','Latex','FontSize',20);
xlabel('Azimut (rad)','Interpreter','Latex','FontSize',16);
ylabel('Azimuth (rad)','Interpreter','Latex','FontSize',16);

% Velocidad
figure;
scatter(x,y,[],v,'fill');
title('Velocidad (m/s)','Interpreter','Latex','FontSize',16);
xlabel('Azimut (rad)','Interpreter','Latex','FontSize',16);
ylabel('Altitud (rad)','Interpreter','Latex','FontSize',16);
xlim([-0.2 0.2]);
ylim([-0.2 0.2]);
colorbar;

%% Colisiones
clc
clear all

filecollision = load('collisiondataprius.txt');
tiempo = 17:0.451:107
impulsox = filecollision(:,1);
impulsoy = filecollision(:,2);
impulsoz = filecollision(:,3);
intensidad = filecollision(:,4);

figure;
subplot(2,1,1)
plot(tiempo,impulsox,tiempo,impulsoy,tiempo,impulsoz);
title('Impulso','Interpreter','Latex','FontSize',20);
xlabel('Tiempo','Interpreter','Latex','FontSize',16);
ylabel('Impulso (kgm/s)','Interpreter','Latex','FontSize',16);
legend({'X', 'Y', 'Z'},'Interpreter','Latex','FontSize',14)
grid on;
subplot(2,1,2)
plot(tiempo,intensidad);
title('Intensidad','Interpreter','Latex','FontSize',20);
xlabel('Tiempo','Interpreter','Latex','FontSize',16);
ylabel('Intensidad (kgm/s)','Interpreter','Latex','FontSize',16);

```

```
legend({'X', 'Y', 'Z'}, 'Interpreter', 'Latex', 'FontSize', 14)  
grid on;
```

Código 15. Programa de procesamiento de datos en Matlab

Referencias

- [1] Carla Team 2019. CARLA. (2019). Recuperado de www.carla.org.
- [2] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017). CARLA: An open urban driving simulator.
- [3] Marchand, R. (1992). The Designers Go to the Fair II: Norman Bel Geddes, The General Motors "Futurama," and the Visit to the Factory Transformed. *Design Issues*, 8(2), 23-40.
- [4] The Prometheus project: The story behind one of AV's greatest developments, <https://www.autonomousvehicleinternational.com/features/the-prometheus-project.html>, acceso 16 Junio de 2019.
- [4] Seppelt, B. D., & Victor, T. W. (2016). Potential solutions to human factors challenges in road vehicle automation. In *Road vehicle automation 3* (pp. 131-148). Springer, Cham
- [5] 2GETTHERE PRESENTS ON PLATOONING AT THE FUTURE OF BUSES, <https://www.2getthere.eu/tag/parkshuttle/>, acceso 16 Junio de 2019.
- [6] Schoettle, B., & Sivak, M. (2015). A preliminary analysis of real-world crashes involving self-driving vehicles. *University of Michigan Transportation Research Institute*.
- [7] Ingle, S., & Phute, M. (2016). Tesla autopilot: semi autonomous driving, an uptick for future autonomy. *International Research Journal of Engineering and Technology*, 3(9)

- [8] Fusing Sensors for the Automated Driving Future, Terry Costlow, <https://saemobilus.sae.org/automated-connected/feature/2019/02/fusing-sensors-for-the-automateddriving-future> , acceso 3 de Julio de 2020.
- [9] Millimeter Wave, Automotive Radar, <https://www.youtube.com/watch?v=uLEqCzaGTLA> , acceso 17 de Junio de 2019.
- [10] LIDAR: how self-driving cars ‘see’ where they’re going. <https://cosmosmagazine.com/technology/how-self-driving-cars-can-see-where-they-re-going>, acceso 16 Junio de 2019.
- [11] Autopilot, https://www.tesla.com/es_ES/autopilot, acceso 16 Junio de 2019.
- [12] National Highway Traffic Safety Administration. (2013). Preliminary statement of policy concerning automated vehicles. *Washington, DC*, 1-14.
- [13] SAE Standars News: J3016 automated-driving graphic update, Jennifer Shuttleworth, <https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-gra>, acceso 3 de Julio de 2020.
- [14] CARLA Documentation, <https://carla.readthedocs.io/en/latest/>, acceso 20 de Enero de 2020.
- [15] Lidar raycast sensor, https://carla.readthedocs.io/en/latest/ref_sensors/#lidar-raycast-sensor, acceso 20 de Enero de 2020.
- [16] Louw, T., Kountouriotis, G., Carsten, O., & Merat, N. (2015). Driver inattention during vehicle automation: How does driver engagement affect resumption of control?. In *4th International Conference on Driver Distraction and Inattention (DDI2015)*, Sydney: proceedings. ARRB Group.
- [17] Dikmen, M., & Burns, C. M. (2016, October). Autonomous driving in the real world: Experiences with tesla autopilot and summon. In *Proceedings of the 8th international conference on automotive user interfaces and interactive vehicular applications*(pp. 225-228). ACM.
- [18] CARLA Simulator, <https://github.com/carla-simulator/carla> acceso el 12 de Junio de 2019.

- [19] Haerri, J., Fiore, M., Filali, F., Bonnet, C., Chiasserini, C. F., & Casetti, C. (2005). *A realistic mobility simulator for vehicular ad hoc networks*. Eurécom Technical Report, Institut Eurécom, France.
- [20] D. Göhring, M. Wang, M. Schnürmacher and T. Ganjineh, "Radar/Lidar sensor fusion for car-following on highways," *The 5th International Conference on Automation, Robotics and Applications*, Wellington, 2011, pp. 407-412.
- [21] Li, J., Bao, H., Han, X. *et al.* Real-time self-driving car navigation and obstacle avoidance using mobile 3D laser scanner and GNSS. *Multimed Tools Appl* **76**, 23017–23039 (2017).
- [22] Latitude, <https://www.nasa.gov/audience/forstudents/k-4/dictionary/Latitude.html>, acceso el 14 de Abril de 2020.
- [23] Longitude, <https://www.nasa.gov/audience/forstudents/k-4/dictionary/Longitude.html>, acceso el 15 de Abril de 2020
- [24] Linux build, https://github.com/carla-simulator/carla/blob/master/Docs/build_linux.md, acceso 23 de Junio de 2020.
- [25] WheeledVehicleAIController, <https://github.com/carla-simulator/carla/blob/0.9.2/Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Vehicle/WheeledVehicleAIController.cpp>, acceso el 6 de Julio de 2020
- [26] RoutePlanner, <https://github.com/carla-simulator/carla/blob/0.9.7/Unreal/CarlaUE4/Plugins/Carla/Source/Carla/Traffic/RoutePlanner.cpp>, acceso el 6 de Julio de 2020