UNIVERSIDAD
DE MÁLAGA

ESCUELA DE INGENIERÍAS INDUSTRIALES

DPTO. INGENIERÍA DE SISTEMAS Y AUTOMÁTICA
DPTO. ARQUITECTURA DE COMPUTADORES

TRABAJO FIN DE GRADO

DETECCIÓN DE AGENTES DEL TRÁFICO EN ENTORNOS URBANOS EN SIMULACIÓN UTILIZANDO CARLA Y ROS2

Grado en

INGENIERÍA ELECTRÓNICA, ROBÓTICA Y MECATRÓNICA

AUTOR: ALBERTO GARCÍA GUILLÉN

TUTOR: JESÚS MORALES RODRÍGUEZ
COTUTOR: FRANCISCO MANUEL CASTRO PAYÁN

MÁLAGA, ENERO DE 2024

ESCUELA DE INGENIERÍAS INDUSTRIALES, UNIVERSIDAD DE MÁLAGA

DETECCIÓN DE AGENTES DEL TRÁFICO EN ENTORNOS URBANOS EN SIMULACIÓN UTILIZANDO CARLA Y ROS2

Autor: Alberto García Guillén.

Tutor: Jesús Morales Rodríguez.

Cotutor: Francisco Manuel Castro Payán.

Departamento: Ingeniería de Sistemas y Automática / Arquitectura de Computadores.

Titulación: Grado en Ingeniería Electrónica, Robótica y Mecatrónica.

Palabras clave: CARLA Simulator, ROS2, cámara RGB, LiDAR 3D, segmentación semántica, detección de obstáculos, redes neuronales, CNN, DETR, conducción autónoma.

Resumen

Este trabajo aborda el procesamiento de las imágenes y nubes de puntos proporcionados por una cámara RGB y un LiDAR 3D abordo de un vehículo para la detección de obstáculos en entornos urbanos en el marco del proyecto de investigación PREMOVE [1]. Dicha tarea se realizará en un entorno digital simulado, que permitirá recrear situaciones realistas minimizando los costes de experimentación y acelerando el proceso de desarrollo.

El simulador en cuestión es CARLA Simulator, que ofrece múltiples escenarios y una gran variedad de herramientas con las que poder experimentar. Para ello, se hará uso del modelo digital del vehículo realizado por Daniel Gamba en el TFG predecesor a este [2]. Dicho modelo incluye además del vehículo, los sensores que lleva abordo. Entre ellos se destacan múltiples cámaras RGB y un sensor LiDAR 3D que recrean los modelos reales que se utilizarán en el proyecto PREMOVE.

Se someterá a dicho automóvil a distintos escenarios, donde el tráfico y el estado climatológico jugarán un papel esencial. Por otro lado, se incluirán agentes del tráfico como peatones o ciclistas que pondrán a prueba de forma crítica el sistema de detección de obstáculos para la conducción segura.

Para facilitar el manejo, agrupación y procesamiento de dichos datos, se hará uso del software ROS2, una interfaz especializada en el desarrollo de programas orientados a la robótica.

Por último, para desarrollar la interfaz encargada de la detección de obstáculos, se acudirá a un modelo preentrenado de red neuronal especializado en dicha tarea. El estado del arte actual sugiere tanto modelos más clásicos como las redes CNN, así como las nuevas redes *Transformer* especializadas en la detección (DETR). Por tanto, un análisis profundo se realizará para determinar la mejor opción con la que puede contar este proyecto.

E.T.S. de Ingeniería de Telecomunicación, Universidad de Málaga

Detection of traffic agents in urban environments in simulation using CARLA and ROS2

Author: Alberto García Guillén.

Supervisor: Jesús Morales Rodríguez.

Co-supervisor: Francisco Manuel Castro Payán.

Department: Systems and Automation Engineering / Computer Architecture.

Degree: Electronics, Robotics and Mechatronics Engineering.

Keywords: CARLA Simulator, ROS2, RGB camera, 3D LiDAR, semantic segmentation, obstacle detection, neural networks, CNN, DETR, autonomous driving.

Abstract

This work deals with the processing of images and point clouds provided by an RGB camera and a 3D LiDAR on board a vehicle for obstacle detection in urban environments in the framework of the PREMOVE research project [1]. This task will be performed in a simulated digital environment, which will allow recreating realistic situations minimizing experimentation costs and accelerating the development process.

The simulator in question is CARLA Simulator, which offers multiple scenarios and a wide variety of tools with which to experiment. For this purpose, use will be made of the digital model of the vehicle made by Daniel Gamba in the TFG previous to this one [2]. This model includes not only the vehicle, but also the sensors on board. Among them are multiple RGB cameras and a 3D LiDAR sensor that recreate the real models that will be used in the PREMOVE proyect.

The car will be subjected to different scenarios, where traffic and weather conditions will play an essential role. On the other hand, traffic agents such as

pedestrians or cyclists will be included to critically test the obstacle detection system for safe driving.

To facilitate the handling, grouping and processing of this data, use will be made of ROS2 software, an interface specialized in the development of robotics-oriented programs.

Finally, a pre-trained neural network model specialized in this task will be used to develop the obstacle detection interface. The current state of the art suggests both more classical models such as CNN networks, as well as the new *Transformer* networks specialized in detection (DETR). Therefore, an in-depth analysis will be carried out to determine the best option for this project.

Dedicado a mi familia,
porque sin ellos no sería quien soy.
En especial a mi hermana,
que es mi orgullo y guía de mi vida.

Acrónimos

API	*Application Programming Interface*
CARLA	*CAR Learning to Act*
CNN	*Convolutional Neural Networks*
CPU	*Central Processing Unit*
DETR	*DEtection TRansformer*
EII	Escuela de Ingenierías Industriales
GPU	*Graphics Processing Unit*
LiDAR	*Light Detection and Ranging*
NPC	*Non Player Character*
PREMOVE	Predicción del movimiento de los participantes del tráfico para la integración segura del vehículo autónomo en áreas urbanas
RGB	*Red, Green and Blue*
ROS2	*Robot Operating System 2*
SAE	*Society of Automotive Engineers*
2D	Bidimensional
3D	Tridimensional

Índice

Resumen III

Abstract V

Acrónimos IX

I Introducción 1

1 Introducción y visión general 3

1.1 Motivación 3
1.2 Marco de realización 5
1.3 Objetivo 5
1.4 Fases de trabajo 5
1.5 Estructura del documento 6

II Desarrollo del proyecto 9

2 Descripción de las herramientas utilizadas 11

2.1 Introducción 11
2.2 Ordenador de laboratorio 12
2.3 CARLA Simulator 12
2.3.1 CARLA Simulator 0.9.14 15
2.3.2 CARLA Simulator 0.9.15 16

2.4 ROS2 17
2.4.1 *Rviz2* 18
2.4.2 ROS2 *Bag* 19
2.5 *Carla-ros-bridge* 20
2.6 *PyTorch* 20
2.7 *OpenPCDet* 21
2.8 *Pcdet_ros2* 21
2.9 TFG de Daniel Gamba 22

3 Desarrollo de la interfaz de CARLA Simulator 23
3.1 Introducción 24
3.2 Incorporación del modelo digital 24
3.2.1 Descripción del modelo 24
3.2.2 Modificación del modelo 25
3.3 Control del vehículo a través de rutas programadas 27
3.4 Control manual del vehículo con volante y pedales 29
3.5 Pruebas de rendimiento e instalación de mapas adicionales 31
3.5.1 Análisis sin sensores semánticos 32
3.5.2 Análisis con sensores semánticos 34
3.5.3 Análisis con sensores semánticos en CARLA Simulator 0.9.15 34
3.6 Generación de agentes del tráfico externos 35
3.6.1 Vehículos NPCs 35
3.6.2 Peatones 36
3.6.3 Agentes externos en CARLA Simulator 0.9.15 37

4 ROS2 para combinar CARLA Simulator y redes neuronales 39
4.1 Introducción a las redes neuronales para la detección de obstáculos 39
4.1.1 Redes CNN 40
4.1.2 Redes DETR 41
4.2 Redes candidatas para procesar datos 43
4.2.1 Redes para procesamiento de imágenes RGB 43

4.2.2 Redes para procesamiento de nubes de puntos 3D 44
4.3 Incorporación en CARLA Simulator a través de ROS2 46
4.3.1 Nodo para redes YOLO 47
4.3.2 Nodo para redes DETR 47
4.3.3 Nodo para redes DETR en ROS2 Humble 48
4.3.4 Nodo para redes de detección 3D 49

III Análisis final y conclusiones 51

5 Puesta en marcha y análisis de resultados 53
5.1 Introducción 53
5.2 Realización de simulaciones 54
5.2.1 Condiciones de conducción seleccionadas 54
5.2.2 Grabación de datos con ROS2 *Bag* 55
5.3 Resultados del procesamiento de datos 55
5.3.1 Simulación 1 55
5.3.2 Simulación 2 63
5.3.3 Simulación 3 69

6 Conclusiones y líneas futuras 75
6.1 Conclusiones 75
6.2 Líneas futuras 76

IV Apéndices 79

A Manual de instalación de Software 81
A.1 Contextualización 81
A.2 CARLA Simulator distribución *package* 82
A.2.1 Descarga del paquete 82
A.2.2 Instalación de la librería del cliente de CARLA Simulator . . 82

A.2.3 Instalación de mapas adicionales 83
A.3 ROS2 Foxy 84
A.3.1 Añadir repositorio 84
A.3.2 Instalación del paquete ROS2 84
A.3.3 Descarga de paquetes externos para este TFG 85
A.4 *Carla-ros-bridge* 88
A.5 Librerías y código base para uso de redes neuronales 88
A.5.1 CUDA 12.1 89
A.5.2 *PyTorch* 89
A.5.3 *Ultralytics* 90
A.5.4 *Transformers* 90
A.5.5 *OpenPCDet* 90

B Manual de Usuario **93**
B.1 Introducción y repositorio de *GitHub* 93
B.2 CARLA Simulator 94
B.2.1 Crear modelo digital dentro del simulador 94
B.2.2 Modificar modelo digital 95
B.2.3 Algoritmo para crear y utilizar rutas programadas 96
B.2.4 Algoritmo para utilizar el control manual 97
B.2.5 Generación de agentes del tráfico externos 98
B.2.6 Otros comandos básicos importantes 99
B.3 ROS2 101
B.3.1 Modificar, compilar y aplicar cambios en un paquete 101
B.3.2 Puesta en marcha del LiDAR fusión normal 102
B.3.3 Puesta en marcha del LiDAR fusión semántico 102
B.3.4 ROS2 *Bag* y *rviz2* 102
B.3.5 Uso de los nodos de procesamiento de datos 103
B.4 Ejemplo de algoritmo para realización de simulaciones 104
B.5 Ejemplo de algoritmo para procesar datos recopilados 107

Bibliografía **112**

Índice de figuras

1.1 Niveles de automatización según SAE. Fuente [3]. 4

2.1 GPU usada en el equipo. Fuente [4]. 12
2.2 Ejemplo de simulación hecha en CARLA Simulator. Fuente [5]. . . 14
2.3 Visualización de las nuevas clases semánticas. Fuente [6]. 15
2.4 Vista preliminar de *Town12*. Fuente [7]. 16
2.5 Vista preliminar de *Town15*. 16
2.6 Visualización de imágenes estéreo y nubes de puntos 3D con *rviz2*. 19
2.7 Distribución de los sensores en el vehículo. Fuente [2]. 22

3.1 Modelo digital del Toyota Prius en CARLA Simulator. 25
3.2 Nube de puntos generada por un LiDAR semántico. Fuente [8]. . . 26
3.3 Representación de los *spawn points*. 28
3.4 Ruta programada en *Town10HD_Opt*. 29
3.5 Ruta programada en *Town03_Opt*. 30
3.6 Equipo Logitech G27. 31
3.7 Sección de *Town06_Opt*. Fuente [9]. 33
3.8 Vista en planta de *Town07_Opt*. 33
3.9 Ejemplos de vehículos NPCs creados. 36
3.10 Ejemplos de peatones creados. 37

4.1 Arquitectura CNN. Fuente [10]. 41
4.2 Arquitectura DETR. Fuente [11]. 42
4.3 Arquitectura de PV-RCNN. Fuente [12]. 45

4.4 Arquitectura de Part-A2-Net. Fuente [13]. 46
4.5 Estructura de *pcdet_ros2* visto desde el repositorio en *Github*. . . 50

5.1 Condiciones de las simulaciones en *Town10HD_Opt*. Fuente [14]. 54
5.2 Procesamiento de *YOLOv8* en la simulación 1. 56
5.3 Procesamiento de *detr-resnet-50* en la simulación 1. 57
5.4 Procesamiento de *detr-resnet-50-dc5* en la simulación 1. 57
5.5 Procesamiento de *detr-resnet-101* en la simulación 1. 58
5.6 Procesamiento de *detr-resnet-101-dc5* en la simulación 1. 58
5.7 Detección de vehículos grandes de PV-RCNN en simulación 1. . . 60
5.8 Detección de vehículos grandes de Part-A2-Free en simulación 1. 60
5.9 Detección de bicicletas de PV-RCNN en la simulación 1. 61
5.10 Detección de bicicletas de Part-A2-Free en la simulación 1. 61
5.11 Detección de peatones de PV-RCNN en la simulación 1. 62
5.12 Detección de peatones de Part-A2-Free en la simulación 1. 62
5.13 Peatón captado por cámara RGB en la simulación 1. 63
5.14 Procesamiento de *YOLOv8* en la simulación 2. 64
5.15 Procesamiento de *detr-resnet-50* en la simulación 2. 64
5.16 Procesamiento de *detr-resnet-50-dc5* en la simulación 2. 65
5.17 Procesamiento de *detr-resnet-101* en la simulación 2. 65
5.18 Procesamiento de *detr-resnet-101-dc5* en la simulación 2. 66
5.19 Ejemplo de buena detección 3D de PV-RCNN en simulación 2. . . 67
5.20 Ejemplo de buena detección 3D de Part-A2-Free en simulación 2. 67
5.21 Detección de peatones de PV-RCNN en la simulación 2. 68
5.22 Detección de peatones de Part-A2-Free en la simulación 2. 68
5.23 Peatón captado por cámara RGB en la simulación 2. 69
5.24 Procesamiento de *YOLOv8* en la simulación 3. 70
5.25 Procesamiento de *detr-resnet-50* en la simulación 3. 71
5.26 Procesamiento de *detr-resnet-50-dc5* en la simulación 3. 71
5.27 Procesamiento de *detr-resnet-101* en la simulación 3. 72

5.28 Procesamiento de *detr-resnet-101-dc5* en la simulación 3. 72
5.29 Detección de PV-RCNN de múltiples objetos en la simulación 3. . 73
5.30 Detección de Part-A2-Free de múltiples objetos en la simulación 3. 74

A.1 Archivos para CARLA Simulator *package* 0.9.14. 82
A.2 Archivos *.whl* disponibles para CARLA Simulator 0.9.14. 83
A.3 Ejemplo de selección de características para CUDA 12.1. 89
A.4 Ejemplo de selección de *PyTorch*. 90

B.1 Ejemplo de actores principales en un archivo *.json*. 95
B.2 Ejemplo de sensores creados para el vehículo sensorizado en un archivo *.json*. 96
B.3 Ejemplo de especificación de NPCs a generar. 97
B.4 Ejemplo de especificar una cantidad nula de NPCs a generar. . . . 98
B.5 Interfaz de *rviz2* con todos los sensores activos. 103
B.6 Ventana con control manual del vehículo sensorizado. 105
B.7 Vista previa del terminal con todos los comandos. 107
B.8 Resultados de detecciones 3D en *rviz2*. 108

Índice de Tablas

4.1 Modelos destacables de *Facebook* en *Hugging Face* 44
4.2 Precisión de los modelos citados en la detección 3D de agentes . 46

Parte I

Introducción

Capítulo 1

Introducción y visión general

Contenido

1.1 Motivación 3
1.2 Marco de realización 5
1.3 Objetivo 5
1.4 Fases de trabajo 5
1.5 Estructura del documento 6

1.1. Motivación

Hoy en día, con la introducción de la industria 4.0, la automatización de todo tipo de cadenas de producción y de los mismos robots y máquinas es una realidad. No es una excepción en la industria de la automoción y el transporte, que además sigue siendo una de las más relevantes a nivel mundial. Los vehículos autónomos se encuentran en plena fase de desarrollo y cada vez ganan más terreno.

Coches autónomos como el Tesla Model 3, uno de los más populares en Europa, llevan incorporados multitud de sensores, un software de detección de obstáculos, entre otros componentes. Estos lo permiten situarse en el nivel 3 de conducción autónoma de los 6 (ver Fig. 1.1) que define la Sociedad de Ingenieros de la Automoción (SAE) [3].

Ahora bien, se están investigando cada día nuevas tecnologías que puedan aportar avances en esta industria. Por ejemplo, la incorporación de sensores LiDAR 3D es una de ellas. Otra que cada vez gana más voz es la incorporación

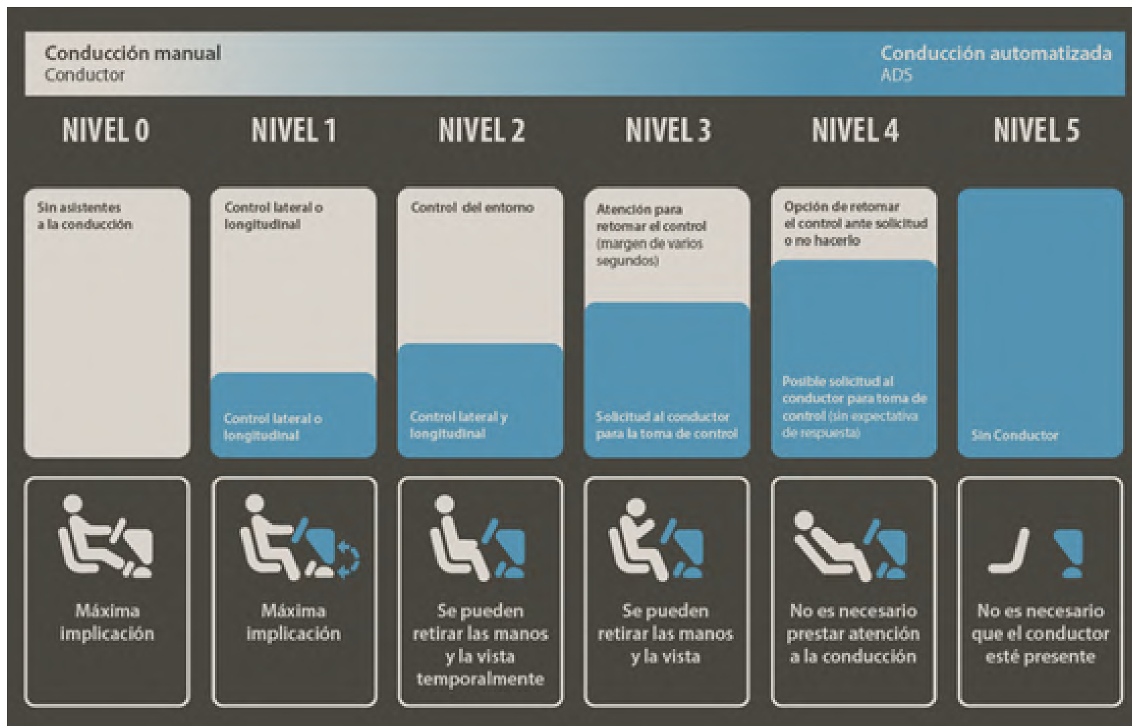

Figura 1.1: Niveles de automatización según SAE. Fuente [3].

de los simuladores en el desarrollo de estos vehículos. Permiten aminorar los costes materiales y de tiempo, así como realizar pruebas más seguras. Ofrecen una gran versatilidad para, por ejemplo, poder ajustar las condiciones de la conducción que se quieran probar, modificar y comprobar la distribución de los sensores, combinarlo con el modelado del vehículo que se quiera utilizar, etc.

Se acentúa aún más la necesidad de incorporar estos simuladores con la introducción de las redes neuronales para la detección de obstáculos. Tanto las más clásicas redes de convolución (CNN) [15] como las novedosas Transformer aplicadas a la detección (DETR) [11] requieren una gran cantidad de datos para poder ser correctamente entrenadas, verificadas e implementadas. Los simuladores ofrecen la posibilidad de generar grandes cantidades de datos a un coste muy reducido. Además, como ya se ha mencionado, el hecho de poder modificar las condiciones de conducción al antojo del usuario permite optimizar y mejorar la calidad de estos datos.

1.2. Marco de realización

El TFG que se menciona se enmarca en las actividades de investigación del departamento de Ingeniería de Sistemas y Automática. Específicamente, se está desarrollando en el proyecto PREMOVE [1] también conocido como *Predicción del movimiento de los participantes del tráfico para la integración segura del vehículo autónomo en áreas urbanas*. El objetivo del proyecto es desarrollar técnicas para detectar y predecir el movimiento de los participantes del tráfico, con el fin de evitar colisiones y facilitar la integración segura de los vehículos autónomos en áreas urbanas.

1.3. Objetivo

El propósito de este trabajo es desarrollar un software de detección de agentes del tráfico para la futura navegación autónoma del vehículo en cuestión.

Para ello se ha marcado cuatro diferentes objetivos específicos. En primer lugar, utilizar el simulador *CAR Learning to ACT* (CARLA) *Simulator* [16], a partir del cual se desarrollará la interfaz que creará las situaciones del tráfico realistas.

En segundo lugar, hacer un estudio del arte en tareas de detección en base a imágenes RGB y nubes de puntos provenientes de LiDAR 3D. En tercer lugar, integrar el software Robot Operating System 2 (ROS2) [17] para aplicar las estrategias de detección estudiadas a las simulaciones realizadas.

Finalmente, utilizar el puente *carla-ros-bridge* [18] para enlazar ROS2 y CARLA Simulator con el objetivo de facilitar la transmisión y el manejo de datos.

1.4. Fases de trabajo

En primer lugar, es necesario preparar el entorno de trabajo. Esto incluye instalar CARLA Simulator, ROS2, y carla-ros-bridge. Este último componente está desarrollado para funcionar con la distribución de *Linux Ubuntu 20.04 Focal* LTS y ROS2 Foxy. En cuanto a la versión del simulador, se escogerá la número 0.9.14, la más reciente, que además tiende a funcionar mejor con Ubuntu.

Una vez el entorno principal esté implementado, el siguiente paso será analizar, probar y modificar los archivos que componen el modelo digital del vehículo desarrollados en el TFG previo [2]. Estos incluyen al automóvil, así como a sus sensores, que buscan mimetizar unas cámaras de profundidad ZED2i, unas cá-

maras térmicas Seek CompactPRO, un sensor LiDAR 3D RS-Helios 5515 y otros sensores de bajo coste. Una vez el modelo esté adecuado al objetivo de este TFG, será necesario aprender a utilizar el simulador para poder generar situaciones de tráfico similares a las que se pueden llegar a producir en la ampliación de Teatinos de Málaga. Esto implica un entorno urbano con numerosos agentes del tráfico como otros autocares, motocicletas, bicicletas o peatones. Por otro lado, para el manejo del vehículo autónomo, se considerará tanto la programación de rutas concretas como su control manual. Además, se hará un estudio del rendimiento del simulador en distintos mapas soportando todos estos factores junto con la generación de datos de los sensores.

Seguidamente se utilizará el software ROS2 con el objetivo de manejar estos datos. Por una parte, se recurrirá a la herramienta *rviz2* [19] para visualizar y comprobar la información producida. También será necesario ROS2 *Bag* [20] para su guardado en archivos de formato *.bag*, que permitán luego reproducir la simulación exacta en cualquier momento, teniendo así una amplia base de datos. Además, se hará uso del nodo de ROS2 creado en el TFG previo [2], que pone en funcionamiento el modelo de LiDAR 3D que fue diseñado en aquel proyecto.

Cabe destacar que ambas interfaces, CARLA y ROS2, estarán estrechamente relacionadas entre sí durante todo el desarrollo del trabajo. Se empleará para ello el paquete de ROS2 *carla-ros-bridge*, con el que se realizarán algunas funcionalidades como crear el modelo digital dentro del simulador o sincronizar con ROS2 los datos generados desde CARLA.

Paralelamente, se efectuará un estudio del estado del arte de las distintas estrategias para la detección de obstáculos aplicada a la conducción autónoma. Las alternativas más llamativas incluyen el uso de las clásicas CNN como de las más nuevas DETR para la detección de los agentes del tráfico. Con el objetivo de elegir a los mejores candidatos para la tarea, se tendrá en consideración el tamaño de la red, su efectividad y la viabilidad de su aplicación en tiempo real.

Finalmente, se realizará una puesta en marcha de toda la interfaz generada y se analizarán los resultados producidos por las distintas redes neuronales probadas.

1.5. Estructura del documento

La estructura del proyecto es la siguiente:

- El primer capítulo va dedicado a la introducción al proyecto y a una visión general de este.

- En el segundo capítulo se enumeran y describen las diferentes herramientas, softwares y librerías utilizadas a lo largo del TFG.
- El tercer capítulo consiste en la presentación de todo el trabajo y código desarrollado para el simulador de CARLA. Además, se incluyen detalladamente los aportes de *carla-ros-bridge* para crear el modelo digital del vehículo en el servidor.
- El quinto capítulo comprende por un lado la descripción de las distintas estrategias para la detección de obstáculos consideradas. Por otra parte, también el desarrollo de la interfaz de ROS2 para poder integrar distintos modelos de red al resto del sistema.
- El sexto capítulo comprende el análisis y validación de los resultados obtenidos en los datos procesados.
- El capítulo siete presenta las conclusiones y trabajos futuros.
- La última parte del documento comprende los anexos para instalación y manual de usuario, así como referencias bibliográficas.

Parte II

Desarrollo del proyecto

Capítulo 2

Descripción de las herramientas utilizadas

Contenido

2.1 Introducción **11**
2.2 Ordenador de laboratorio **12**
2.3 CARLA Simulator **12**
2.3.1 CARLA Simulator 0.9.14 15
2.3.2 CARLA Simulator 0.9.15 16
2.4 ROS2 . . . . **17**
2.4.1 *Rviz2* 18
2.4.2 ROS2 *Bag* 19
2.5 *Carla-ros-bridge* **20**
2.6 *PyTorch* **20**
2.7 *OpenPCDet* **21**
2.8 *Pcdet_ros2* **21**
2.9 TFG de Daniel Gamba **22**

2.1. Introducción

En este capítulo se realiza una descripción de todos los elementos de trabajo utilizados durante el proyecto. Estos son CARLA Simulator, ROS2 y sus herramientas *rviz2* y ROS2 *Bag*, *carla-ros-bridge*, *PyTorch* [21], *OpenPCDet* [22] y el

TFG de Daniel Gamba. Además, se expondrán los componentes que contiene el ordenador de laboratorio con el que se ha trabajado.

A continuación, se desarrolla con más profundidad una explicación de cada uno de ellos en sus correspondientes apartados.

2.2. Ordenador de laboratorio

Las especificaciones del equipo utilizado durante este proyecto están listadas a continuación:

- **Procesador**: Intel i7-13700F a 5.2GHz con 16 cores y 24 MB L2 Cache.
- **Memoria**: DDR5 2X16GB 5600MHz.
- **Tarjeta gráfica**: GIGABYTE RTX 4070 TI GAMING OC con 12GB de memoria (ver Fig. 2.1).

Figura 2.1: GPU usada en el equipo. Fuente [4].

2.3. CARLA Simulator

CARLA es un simulador de código abierto especializado en el desarrollo, entrenamiento y validación de sistemas de conducción autónoma. Fue desarrollado

por investigadores y estudiantes de la Universidad Autónoma de Barcelona junto con el laboratorio de *Intel Computing Visual*. Es un entorno virtual basado en Unreal Engine altamente personalizable en el cual se disponen de herramientas para, por ejemplo, crear agentes del tráfico, tener control total de todos los actores dinámicos o generar mapas desde cero.

Algunas de las características más notables de este simulador son las siguientes:

- **Escalabilidad a través de una arquitectura servidor multi-cliente:** Multitud de clientes en un mismo o en distintos nodos permiten controlar todos los actores que se desee.
- **Sensores de conducción autónoma:** Esto da la posibilidad de configurar distintos tipos de sensores como LiDARs, cámaras, sensores de profundidad, GPS, entre otros.
- **Modo sin renderizado:** De esta forma puedes deshabilitar el renderizado y realizar simulaciones de tráfico en las que los gráficos no sean necesarios a una alta velocidad.
- **ScenarioRunner:** Se trata de una herramienta de CARLA especializada en la generación de situaciones concretas del tráfico.
- **Generación de mapas:** Los usuarios pueden fácilmente crear su propios mapas siguiendo los estándares de *OpenDrive* utilizando herramientas como RoadRunner. Además, CARLA ofrece la posibilidad de elegir entre un abanico de mapas ya creados por los desarrolladores.
- ***Application Programming Interface* (API) flexible:** CARLA Simulator proporciona una extensa interfaz de programación de aplicaciones que facilita el control de todos los aspectos de la simulación
- **Integración con ROS2:** A través de su puente *carla-ros-bridge*, se abre la puerta de integrar el simulador con otros entornos de desarrollo.

CARLA Simulator cuenta con dos distribuciones: source y package. La primera, más amplia, pesada y en continuo desarrollo, es la que realmente permite acceder a todas las funcionalidades del simulador, como crear y modificar partes del mapa y controlar todas las capas de los distintos agentes simulados. Por otro lado la distribución package, que es más ligera y sencilla de instalar, si bien posee más limitaciones para editar y modificar el entorno, es más que suficiente para lo que se quiere realizar en este trabajo.

Este simulador está basado en *Unreal Engine* (ver Fig. 2.2). Siguiendo este motor, la base de los distintos actores son los *Blueprints*, que poseen distintas propiedades llamadas *Attributes*. A través de la documentación de CARLA se puede consultar todos estos modelos y características con detalle.

Figura 2.2: Ejemplo de simulación hecha en CARLA Simulator. Fuente [5].

Para la gestión del tráfico, el simulador utiliza la herramienta llamada *traffic manager*. Está diseñada para el control aleatorio o planificado de los vehículos que se estén simulando. Permite mover estos agentes dinámicos para crear situaciones comprometedoras para el vehículo autónomo en cuestión. Sin embargo, para crear escenarios más concretos y detallados, se recomienda el uso de la herramienta ScennarioRunner, especialmente desarrollada para esta tarea. Esto surge del comportamiento errático del manejador de tráfico de CARLA, que como se verá más adelante en el proyecto, llevará a la conclusión de utilizarlo únicamente para generar rutas genéricas y aleatorias de múltiples *Non Played Characters* (NPCs) por todo el mapa.

Este TFG se ha llevado a cabo utilizando en su mayoría la versión CARLA 0.9.14 publicada el 23 de diciembre de 2022. Sin embargo, durante el desarrollo de la interfaz para incorporar redes neuronales al sistema se decidió actualizar a la reciente versión 0.9.15 publicada el 10 de noviembre de 2023. La motivación para este cambio está documentada en la Sección 2.7. Todo el documento sobre el proyecto está escrito para la versión CARLA Simulator 0.9.14, que será igual para CARLA Simulator 0.9.15 a no ser que se indique lo contrario.

A continuación, se explican las características más importantes que aportan

cada una de las versiones mencionadas.

2.3.1. CARLA Simulator 0.9.14

CARLA Simulator 0.9.14 contiene notables mejoras para el simulador como la adición de clases semánticas para diferenciar más tipos de vehículos (ver Fig. 2.3). También ahora el motor de CARLA soporta vehículos con más de cuatro de ruedas, que permite el diseño e implementación de mayor variedad de agentes del tráfico.

Figura 2.3: Visualización de las nuevas clases semánticas. Fuente [6].

Por otro lado, otra incorporación en CARLA Simulator 0.9.14 es el mapa adicional *Town12*. Se trata del primer mapa gigante decorado, que describe una ciudad con escenarios de rascacielos, residenciales y rurales (ver Fig. 2.4).

Sin embargo, la novedad de esta versión que más puede favorecer al proyecto PREMOVE en un futuro es que ahora CARLA soporta múltiples GPUs. Ahora el simulador puede ser distribuido por varios de estos dispositivos para poder ejecutar diferentes instancias sincronizadas en cada uno de ellos. Esto supone una gran mejora en escalabilidad para tener estaciones de trabajo de alto rendimiento.

Figura 2.4: Vista preliminar de *Town12*. Fuente [7].

2.3.2. CARLA Simulator 0.9.15

Una de las nuevas propiedades de esta versión de CARLA Simulator, fue la adición de gemelos digitales para la creación automática de mapas personalizados a partir de la selección de zonas reales.

Por otro lado, se abrió la posibilidad de instalar nuevos mapas adicionales: *Town13* y *Town15*. El primero describe una ciudad muy extensa con escenarios tanto urbanos como rurales. Por otro lado, *Town15* está basado en la Universidad Autónoma de Barcelona (ver Fig. 2.5).

Figura 2.5: Vista preliminar de *Town15*.

Finalmente, con el desarrollo de CARLA Simulator 0.9.15, se ha incluido también en la documentación oficial [23] el *CARLA Catalogue*. Se trata de un repertorio que favorece la rápida búsqueda de información de aquellos actores, mapas u objetos que se quieran utilizar en las simulaciones.

2.4. ROS2

ROS2 es una plataforma de código abierto diseñada para el desarrollo de sistemas robóticos. Es la evolución de ROS o ROS1, con el objetivo de abordar las limitaciones de la versión anterior y proporcionar una base más sólida y flexible para la construcción de robots y sistemas autónomos. Algunas de sus características principales son las siguientes:

- **Arquitectura modular:** ROS2 se basa en una arquitectura modular que permite a los desarrolladores crear sistemas robóticos altamente personalizados y flexibles. Los componentes de software se organizan en paquetes que pueden ser fácilmente compartidos y reutilizados.

- **Soporte para múltiples plataformas:** A diferencia de la versión anterior de ROS, que estaba principalmente orientada a Linux, ROS2 es más agnóstico en cuanto al sistema operativo y es compatible con una variedad de plataformas, incluyendo Linux, Windows y macOS.

- **Mayor enfoque en tiempo real:** ROS2 se diseñó para soportar sistemas en tiempo real y sistemas críticos en cuanto a la seguridad. Esto lo hace más adecuado para aplicaciones que requieren una alta precisión temporal y seguridad, como robots autónomos en entornos industriales.

- **Comunicación mejorada:** ROS2 utiliza el *Middleware de Comunicación ROS* (RCM) para facilitar la comunicación entre nodos. Este middleware es más eficiente y permite una comunicación más confiable y segura, lo que es fundamental para aplicaciones robóticas en tiempo real.

- **Ecosistema en crecimiento:** A medida que ROS2 ha ido madurando, ha ganado un creciente conjunto de paquetes y bibliotecas, lo que facilita el desarrollo de una amplia variedad de aplicaciones robóticas.

- **Herramientas y soporte de desarrollo:** ROS2 viene con un conjunto de herramientas de desarrollo y depuración que facilitan la construcción, pruebas y depuración de aplicaciones robóticas. También cuenta con una comunidad activa y un sólido soporte en línea.

- **Compatibilidad con ROS1:** Aunque ROS2 es una evolución, no una simple actualización de ROS1, se ha hecho un esfuerzo para permitir que los usuarios migren gradualmente de ROS1 a ROS2. Esto significa que los usuarios pueden aprovechar las ventajas de ROS2 sin necesidad de reescribir completamente su código existente.

En ROS2, los nodos son los puntos de entrada más básicos para la comunicación en un sistema. Entre algunos de los modelos de comunicación del sistema, uno bastante característico es el basado en la arquitectura publicador-subscriptor. Estos dos componentes, creados en los mismos nodos, extraen e importan datos a unos topics, canales de información unidireccionales fundamentales.

Por otro lado, en este TFG se ha llevado a cabo utilizando en su mayoría la distribución ROS2 Foxy publicada el 5 de junio de 2020. Sin embargo, durante el desarrollo de la interfaz para incorporar redes neuronales al sistema se decidió actualizar a la más moderna distribución ROS Humble publicada el 23 de mayo de 2022. La motivación para este cambio está documentada en la Sección 2.7. Todo el documento sobre el proyecto está escrito para la distribución ROS2 Foxy, que será igual para ROS2 Humble a no ser que se indique lo contrario.

Finalmente, existen en ROS2 múltiples herramientas muy útiles en el desarrollo de sistemas robóticos como *rviz2* y ROS2 *Bag*, las cuales se describen a continuación.

2.4.1. *Rviz2*

Esta herramienta de ROS2 es vital para la visualización y comprensión de los datos contenidos en los topics. Es una interfaz gráfica sencilla que permite seleccionar aquella información que se quiera mostrar en la pantalla. Permite también adecuar o modificar visualmente estos datos para general resultados más agradables o entendibles.

Como se aprecia en la Figura 2.6, a través de esta herramienta se puede comprobar el correcto funcionamiento de sensores RGB, LiDAR 3D y sus homólogos semánticos utilizados en este TFG. Es importante destacar que *rviz2* no es un simulador, no posee ningún motor capaz de realizar tareas relacionadas con cálculos propios de un sistema robótico.

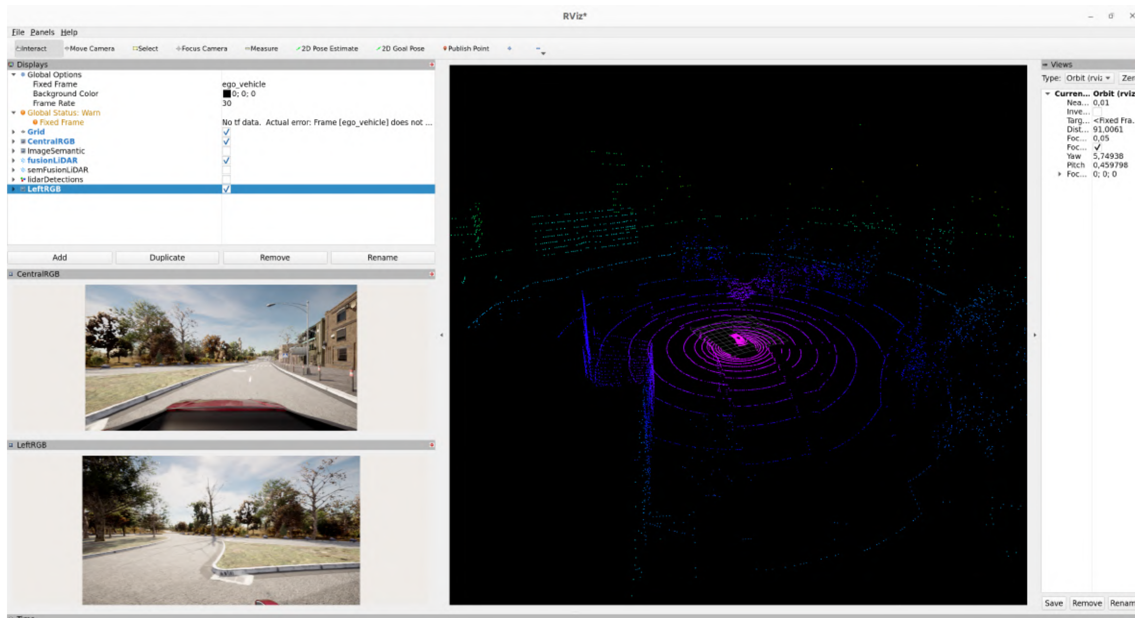

Figura 2.6: Visualización de imágenes estéreo y nubes de puntos 3D con *rviz2*.

2.4.2. ROS2 *Bag*

ROS2 *Bag* es una herramienta de línea de comandos que permite grabar y reproducir datos de ROS2 en forma de archivos *.bag*. Estos archivos pueden contener una variedad de datos, como mensajes de sensores, comandos de control, datos de odometría, etc.

Una de sus grandes ventajas es que permite sincronizar datos de múltiples fuentes y utilizarlos para analizar el comportamiento de un sistema robótico en su conjunto. Aplicado a este trabajo significa poder sincronizar fácilmente los datos aportados por las cámaras RGB y por el LiDAR 3D. También es posible generar una base de datos la cual puede ser reproducida para obtener una exacta réplica de todos los parámetros recogidos por ROS2 durante una simulación o tarea. Además, el formato de estos archivos así como su difusión son bastante sencillos y manejables.

Por último, ROS2 *Bag* es útil también para depurar problemas después de que ocurran. Se puede examinar el estado del sistema en un punto en el tiempo y analizar qué ocurrió.

2.5. *Carla-ros-bridge*

El puente de CARLA con ROS2 es un conjunto de paquetes de este último software que permite la comunicación bidireccional entre ambos sistemas. La información de CARLA se clasifica y publica en los correspondientes topics de ROS2. Por otro lado, los comandos ejecutados desde ROS2 se traducen en acciones reflejadas en el simulador.

La mayor parte de los paquetes que conforman esta herramienta están escritos en el lenguaje de programación *Python* [24] y se ejecutan con los típicos comandos de ROS2. En este trabajo se han utilizado dos de estos paquetes:

- **CARLA ROS Bridge:** Es el nodo principal que se debe ejecutar para iniciar el puente y las comunicaciones entre ambos sistemas. Se puede iniciar en modo pasivo, de forma que será el servidor de CARLA Simulator quien marque el tiempo de simulación.
- **CARLA Spawn Objects:** Conforma una herramienta genérica que da la posibilidad de generar actores dentro del servidor de CARLA a través de archivos de formato json previamente codificados. Es una buena opción para tener varias distribuciones y modelos digitales del vehículo a simular de forma ordenada.

Este TFG se ha llevado a cabo utilizando en su mayoría el puente *carla-ros-bridge* publicado en *ROS Bridge Documentation* [18] por CARLA Simulator el 22 de julio de 2022. Esta versión está diseñada para trabajar con ROS2 Foxy y CARLA Simulator 0.9.11 o mayor y *Python 3.8*.

Sin embargo, durante el desarrollo de la interfaz para incorporar redes neuronales al sistema se decidió actualizar a la versión publicada en el repositorio de *Github ttgamage / carla-ros-bridge* [25]. Esta versión está diseñada para trabajar con ROS2 Humble, CARLA Simulator 0.9.15 y *Python 3.10*. La motivación para este cambio está documentada en la Sección 2.7.

Todo el documento sobre el proyecto está escrito para la versión publicada en *ROS Bridge Documentation*, que será igual para la versión *ttgamage / carla-ros-bridge* a no ser que se indique lo contrario.

2.6. *PyTorch*

PyTorch es una biblioteca de aprendizaje profundo de código abierto desarrollado por Facebook's AI Research lab (FAIR) y escrita en *Python*. Es ampliamente

utilizado en el campo del *deep learning* y de la inteligencia artificial para la construcción y entrenamiento de modelos de redes neuronales.

A través de *PyTorch* es posible cargar modelos de redes neuronales en el código que se desee. Se puede optar por modelos preentrenados u otros que se quieran entrenar a posteriori. Por otro lado, esta bilioteca está diseñada para aprovechar el potencial de las unidades de procesamiento gráfico (GPU), lo que acelera el entrenamiento de estas redes así como la inferencia que se quiera realizar en ellas.

2.7. *OpenPCDet*

OpenPCDet es un proyecto en código abierto desarrollado especialmente para la detección de obstáculos en nubes de puntos tridimensionales generadas por sensores LiDAR. Esta interfaz desarrollada por *OpenMMLab* es ampliamente utilizada en diversos proyectos de investigación como resultado de su disponibilidad y eficacia.

Se trata de una base de código desarrollada en *Python* que actualmente soporta múltiples métodos para la detección de obstáculos 3D. Entre ellos se pueden destacar *VoxelNet*, *PointPillar*, *PV-RCNN* o *Part-A2-Net*.

Esta interfaz está preparada no sólo para el uso de múltiples modelos de redes neuronales, sino también para la evaluación de distintos formatos de datos de entrada. Tanto los típicos estandarizados como *KITTI*, así como otros completamente personalizables.

2.8. *Pcdet_ros2*

Pcdet_ros2 es un paquete de ROS2 publicado en febrero de 2023 [26] que sirve de puente entre *OpenPCDet* y ROS2.

Permite adaptar el software de *OpenPCDet* para usar modelos de redes neuronales 3D en nodos de ROS2. Consta de un nodo principal que lee sus parámetros de entrada del archivo *launch* que contiene el mismo paquete. Este aporta información tanto del tópico de entrada del que la red va a leer las nubes de puntos como del tópico de salida en el que publica los resultados de la detección.

Esta interfaz fue desarrollada para trabajar con *Ubuntu 22.04*, ROS2 Humble y *Python 3.10*, lo cual la hacía incompatible para utilizar con el resto del sistema original del proyecto.

Sin embargo, con la salida de CARLA Simulator 0.9.15 el 10 de noviembre de 2023 así como el puente *ttgamage / carla-ros-bridge* [25], esto dejaba de ser un problema. Esta nueva versión del puente trabajaba con CARLA Simulator 0.9.15 y ROS2 Humble, que a su vez requería de *Python 3.10*. Todas estas condiciones hacían que el uso de *pcdet_ros2* en el proyecto fuese posible, así que se instaló en el equipo de laboratorio una partición de *Ubuntu 22.04* con todo el nuevo software mencionado.

2.9. TFG de Daniel Gamba

El TFG de Daniel Gamba [10] es el predecesor a este trabajo. A raíz de él se posee el modelo digital del vehículo autónomo y de la distribución de sus sensores (ver Fig. 2.7). Además, constituye una valiosa fuente de información para aprender a utilizar CARLA Simulator. Este TFG fue finalizado y presentado en junio de 2023, utilizando las mismas versiones de todos los softwares involucrados en este trabajo.

Por otro lado también aportó una interfaz de ROS2 desde la que partir, con la cual realizaba la fusión de todos los componentes que había modelado previamente para poder simular en CARLA el LiDAR 3D RS-Helios 5515. Finalmente, el espacio de trabajo desarrollado por Daniel contiene una recopilación de código y utilidades que facilitaron algunas tareas durante este proyecto.

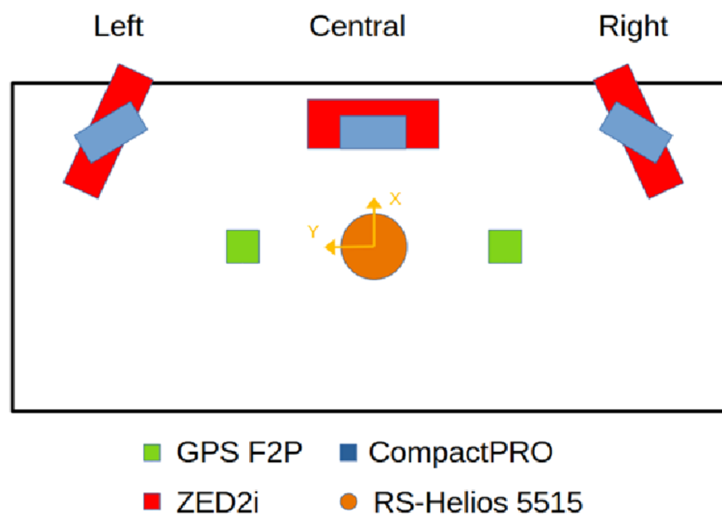

Figura 2.7: Distribución de los sensores en el vehículo. Fuente [2].

Capítulo 3

Desarrollo de la interfaz de CARLA Simulator

Contenido

3.1 Introducción 24

3.2 Incorporación del modelo digital 24

3.2.1 Descripción del modelo 24

3.2.2 Modificación del modelo 25

3.3 Control del vehículo a través de rutas programadas 27

3.4 Control manual del vehículo con volante y pedales 29

3.5 Pruebas de rendimiento e instalación de mapas adicionales 31

3.5.1 Análisis sin sensores semánticos 32

3.5.2 Análisis con sensores semánticos 34

3.5.3 Análisis con sensores semánticos en CARLA Simulator 0.9.15 34

3.6 Generación de agentes del tráfico externos 35

3.6.1 Vehículos NPCs 35

3.6.2 Peatones 36

3.6.3 Agentes externos en CARLA Simulator 0.9.15 37

3.1. Introducción

En este capítulo se realiza una descripción del proceso de desarrollo de la interfaz de CARLA Simulator para la generación de simulaciones realistas con el vehículo sensorizado.

Se aborda en primer lugar la descripción, incorporación y modificación del modelo digital del coche. Seguidamente se explica el análisis sobre las distintas opciones para el manejo del vehículo durante las simulaciones. Además se exponen las pruebas de rendimiento realizadas en los distintos mapas desarrollados por CARLA Simulator. Finalmente, se detalla la generación de los agentes del tráfico externos como vehículos NPCs y peatones.

El código fuente asociado a esta interfaz de CARLA Simulator está accesible y disponible para revisión en el repositorio de *GitHub* correspondiente. Aquellos interesados pueden acceder al código en [27].

3.2. Incorporación del modelo digital

El primer paso fue cargar el modelo digital del vehículo sensorizado que había sido desarrollado en el TFG anterior. Los archivos que contienen toda esta información son de formato *.json* , desde los cuales se ha cargado directamente el automóvil con sus sensores en el simulador. A continuación, se expone una breve descripción de su contenido.

3.2.1. Descripción del modelo

El modelo del vehículo utilizado de las librerías de CARLA Simulator es el Toyota Prius (ver Fig. 3.1), aquel con las medidas físicas más semejantes al del automóvil Nissan Leaf, que será el utilizado en las pruebas reales de PREMOVE.

Por otro lado tiene adjuntos múltiples sensores. Posee seis cámaras RGB que simulan las tres ZED2i que llevará equipadas el vehículo real. Además, lleva incorporado ocho sensores LiDAR con unas características muy concretas para que, al fusionarlos, describan un comportamiento similar al del LiDAR 3D RS-Helios 5515. Cabe destacar otras cámaras RGB incorporadas con sus atributos modificados para que traten de simular las cámaras térmicas Seek CompactPRO en términos de resolución y campo de visión.

Figura 3.1: Modelo digital del Toyota Prius en CARLA Simulator.

Realmente los archivos *.json* desarrollados por Daniel Steven Gamba contienen información de múltiples agentes, entre ellos el vehículo mencionado, que luego son creados dentro del servidor de CARLA Simulator. Dichos agentes pueden ser sensores sueltos, otros vehículos que sirven de tráfico genérico o peatones.

Una vez analizado el modelo, el siguiente paso que se ha realizado es modificarlo para adecuarlo a las necesidades de este trabajo.

3.2.2. Modificación del modelo

Como ya se ha mencionado, el objetivo final de este TFG es implementar una estrategia de detección de obstáculos utilizando para ello redes neuronales especializadas en la tarea. Además, los datos de entrada a dichas redes han sido en todo caso las imágenes proporcionadas por las cámaras ZED2i y las nube de puntos que genera el LiDAR 3D. Por ende, para este proyecto, el resto de sensores que lleva incorporados el vehículo pueden ser descartados.

Ahondando un poco más, como se documenta en la descripción del modelo, cada cámara ZED2i viene representada por dos cámaras RGB de CARLA Simulator. Cada una de estas están colocadas muy cerca, una al lado de la otra, para recrear la distribución del sensor real, con la cual se logra obtener información tridimensional en las imágenes relacionada con la profundidad de cada píxel. Sin embargo, la información 3D que se le proporcionará a las redes neuronales vendrá dada por la nube de puntos del LiDAR. Es decir, se puede prescindir de una cámara RGB de cada pareja que simula una ZED2i en el modelo digital.

Por otro lado, también han sido incorporados sensores homólogos a los ya presentes con una capacidad extra semántica. En CARLA Simulator existen un tipo de sensores semánticos, tanto para cámaras RGB como para LiDARs. Estos

generan información directamente etiquetada, lo cual puede ser muy interesante para entrenar o estudiar los datos procesados por las redes neuronales a posteriori. Como resultado se incorporaron tres cámaras RGB de segmentación semántica en las mismas posiciones y con características muy similares a las originales. Lo mismo ocurrió para los ocho LiDARs semánticos generados. En la Fig. 3.2 se muestra un ejemplo de una nube de puntos generada por un LiDAR 3D semántico, donde cada color representa una clase.

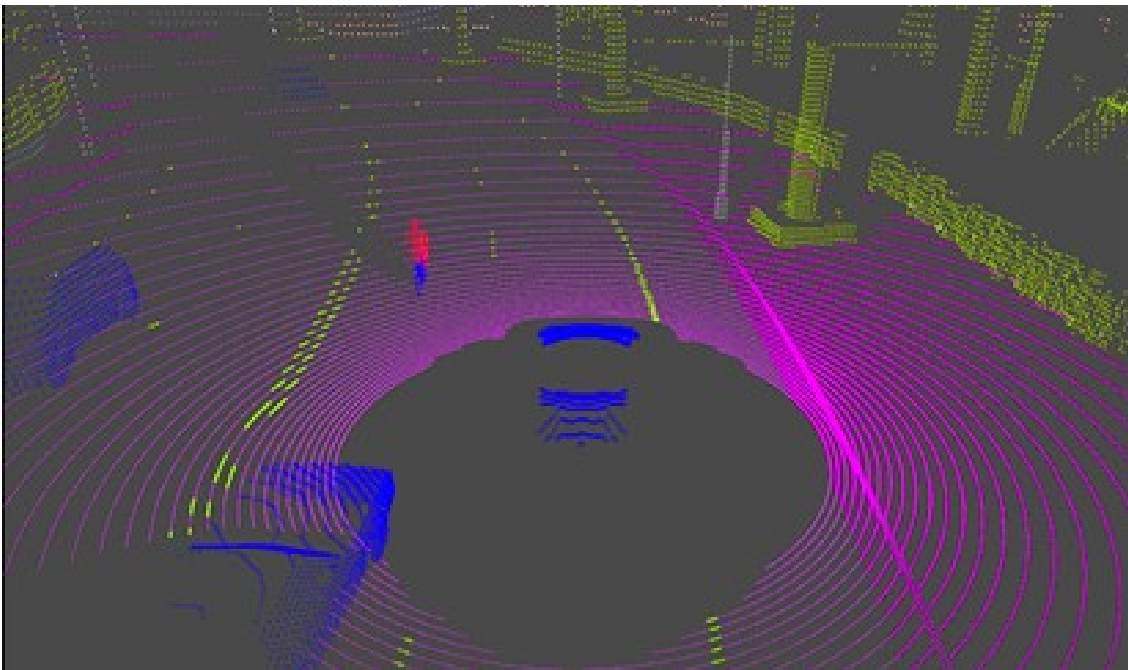

Figura 3.2: Nube de puntos generada por un LiDAR semántico. Fuente [8].

Por último, se decidió prescindir de todos los NPCs de los archivos *.json*. Esta forma de crear agentes del tráfico es muy detallada en comparación a las simulaciones de tráfico genérico que se quieren realizar, donde no hará falta tanta precisión por lo menos para los vehículos. En el caso de los peatones, se decidió dejar la tarea de crearlos en el mismo archivo en el que son controlados, dejando todo más ordenado y conciso.

Una vez el modelo hubo sido adecuado, se procedió a programar el control del vehículo. Para ello se evaluaron dos opciones: programación de rutas y control manual, las cuales se detallan a continuación.

3.3. Control del vehículo a través de rutas programadas

En un principio, para someter al vehículo autónomo a los escenarios de interés dentro del simulador se pensó en utilizar un enfoque similar al empleado en el TFG previo. Dicho método consistía en especificar una ruta en forma de puntos que el vehículo seguiría con ayuda de la herramienta *traffic manager*. Estos son los llamados *spawn points*, que en CARLA Simulator son etiquetas numéricas que representan ciertas localizaciones dentro del mapa situadas en zonas para el tráfico de vehículos.

Cada mapa en el simulador posee sus propios *spawn points* y es importante saber que para una misma localización, la etiqueta de dichos puntos varía según qué distribución de CARLA Simulator se esté utilizando (*package* o *source*).

El orden de trabajo utilizado para diseñar estas rutas fue similar independientemente del mapa por el que se quisiera hacer conducir al automóvil. La primera ciudad puesta a prueba fue *Town10HD_Opt* y con el siguiente procedimiento:

En primer lugar, se iniciaba el servidor de CARLA Simulator y se utilizaba el cliente *spawn_points.py* escrito en *Python* para mostrar sobre la misma ciudad las etiquetas de dichos puntos del mapa (ver Fig. 3.3). Este archivo debía ser ejecutado desde la carpeta de utilidades de este TFG que lo contiene. Tras esto, el siguiente paso era navegar por el mapa a través del espectador con las flechas del teclado, buscar los *spawn points* de interés y anotarlos.

A continuación se creaba un cliente de CARLA Simulator y se programaba un array con todos los puntos seleccionados secuencialmente. Después se modificaba la configuración del *traffic manager* dentro del mismo cliente, al cual posteriormente se le especificaba la ruta previamente definida. Por útlimo se le cedía el control del vehículo al *traffic manager* para que calculara y efectuara la ruta que el vehículo debería seguir en función de los *spawn points* elegidos.

Aunque sobre el papel pareciera un algoritmo robusto y sencillo, se percibió que el comportamiento de *traffic manager* resultaba errático. No seguía las rutas definidas y eran recurrentes las ocasiones en las que comenzaba a hacer circular al vehículo en bucle alrededor de una pequeña zona.

Para contrarrestar este inconveniente, se añadían al cliente creado una serie de líneas de código que, en las zonas de la ruta más problemáticas, recuperaba el control del vehículo para manejarlo manualmente con comandos de la API de *Python* de CARLA Simulator. Sin embargo este parche no resolvía por completo la situación, ya que dichos comandos eran tediosos de utilizar y sólo cubrían fá-

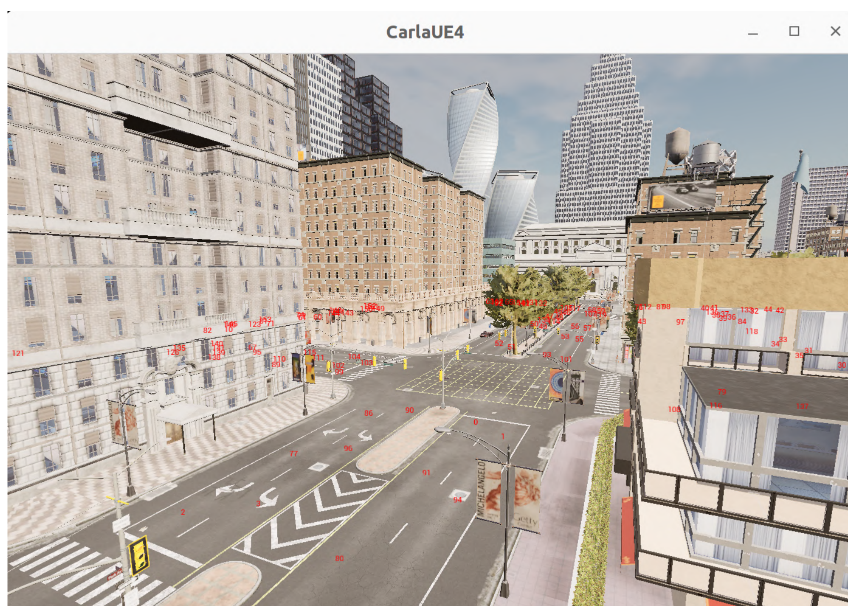

Figura 3.3: Representación de los *spawn points*.

cilmente maniobras para cambiar de carril y continuar recto en algunos cruces. La inestabilidad de esta herramienta provocó que sólo se pudieran hacer dos rutas medianamente completas: una para *Town10HD_Opt* y otra para *Town03_Opt* (ver Figs. 3.4 y 3.5).

El resultado son dos rutas largas y con múltiples escenarios que servirán para poner a prueba al vehículo. Sin embargo, su creación ha sido bastante tediosa, lo cual ha llevado entre otras cosas a que estos sean prácticamente los dos únicos itinerarios posibles. Por supuesto, se intentó seguir el algoritmo previamente descrito con el resto de mapas proporcionados por CARLA Simulator, pero en ninguno de ellos se ha logrado hacer una ruta estable que realmente funcione bien.

Aún así, se ha creado un archivo que genere otros vehículos por toda la ciudad y conduzcan de forma aleatoria y realista llamado *spawn_npc_traffic.py*.

Como conclusión a este apartado, ante el comportamiento excesivamente errático de *traffic manager*, la escasez de rutas creadas y la complejidad del proceso, se decidió finalmente apostar por un control manual del vehículo. Es decir, el mismo usuario debería manejar el vehículo mientras los sensores están en funcionamiento y guardando información.

Figura 3.4: Ruta programada en *Town10HD_Opt*.

3.4. Control manual del vehículo con volante y pedales

La API en *Python* de CARLA Simulator contiene una serie de códigos de utilidades y tutoriales que pueden servir en un momento dado como base para aprender sobre el programa y crear clientes para el simulador. En este caso, dicha interfaz posee varios ejemplos para el control manual de un vehículo: *manual_control.py* y *manual_control_steeringwheel.py* entre otros.

Estos clientes tienen la función de crear un autocar aleatorio dentro del simulador, tomar control sobre él y leer los datos de entrada dados por el usuario para moverlo. En el caso del primer cliente mencionado, el manejo se realiza a través de las flechas del teclado. Por otro lado, el segundo archivo realiza el manejo mediante el volante y pedales *Logitech G29*.

Para el control a través del teclado, hubo que copiar y modificar el código de la interfaz dada por CARLA Simulator. En vez de crear un vehículo aleatorio, se le hizo buscar dentro del simulador el vehículo sensorizado, que debía haber sido previamente creado. El resto de funcionalidades se mantuvieron igual y el nuevo nombre del archivo pasó a ser *manual_control_toyota.py*, que se puede encontrar en la documentación de este TFG.

Figura 3.5: Ruta programada en *Town03_Opt*.

Sin embargo, para hacer simulaciones realistas, la precisión con la que se podía conducir el coche podía llegar a ser insuficiente en algunos momentos. Es por esto que se decidió utilizar un volante y pedales de *Logitech* modelo *G27* disponible en el Departamento de Ingeniería de Sistemas (ver Fig. 3.6). Para ello, se modificó el archivo *manual_control_steeringwheel.py* para que fuese compatible con esta versión del hardware y se verificó que funcionara.

La configuración final del control resultó en la siguiente:

- **Control de la dirección:** Girando el volante hacia la izquierda y derecha.
- **Control de la aceleración:** Pisando el pedal derecho.
- **Control del freno:** Pisando el pedal central.
- **Activación del freno de mano:** Leva metálica derecha justo detrás del volante. El efecto de esta frenada en el simulador es similar a la creada por el pedal de freno, de forma que por comodidad se recomienda detener el vehículo con esta pieza metálica.
- **Cambio de marcha:** El simulador cambia las marchas automáticamente, no es necesario controlarlo.

- **Activación de la marcha atrás:** Leva metálica izquierda justo detrás del volante. Pulsar una vez para alternar entre conducción hacia adelante y hacia atrás.

(a) Pedales. (b) Volante.

Figura 3.6: Equipo Logitech G27.

El nombre definitivo del cliente que se ha utilizado desde entonces para el manejo del vehículo es *manual_toyota_steering_and_traffic.py*. Ahora bien, a continuación era vital comprobar el rendimiento de todo el sistema en los distintos mapas. Es decir, comprobar si el ordenador de laboratorio podría soportar toda la carga a la que lo sometería el servidor de CARLA Simulator, en el que se había creado ya lo más pesado: el modelo digital del vehículo con todos sus sensores y el cliente que realizaba el control manual.

3.5. Pruebas de rendimiento e instalación de mapas adicionales

Tras haber configurado el sistema de control, se estudiaron todos los mapas de CARLA Simulator para comprobar la carga de la CPU y la estabilidad de cada una de las ciudades con todo el sistema básico ejecutado. Antes de nada es preciso aclarar que existen dos tipos de mapas según sus capas: aquellos que posean en el nombre la extensión *_Opt* y aquellos que no posean la extensión *_Opt*.

Este sufijo en el nombre significa que todos los componentes están clasificados por capas. Por ejemplo, las carreteras y señales de tráfico son de la capa 0, los edificios de la capa 1, el cielo y el tiempo atmosférico de la capa 2, etc. Aquellos mapas cuyo nombre no posea dicho sufijo serán unicapa. Experimentalmente se comprobó que el formato multicapa resulta más estable, además de

más útil. Cada mapa puede poseer tanto una versión unicapa como una multicapa, aunque en este análisis sólo se habla de la segunda opción. Es imporante saber que a partir de la décima ciudad, independientemente de su nombre, siempre usarán el sistema multicapa.

3.5.1. Análisis sin sensores semánticos

En primer lugar, se realizó un análisis de rendimiento incluyendo sólamente los sensores normales: es decir, aquellos que el vehículo autónomo real tendría equipados. Para eso, se retiró del modelo digital las cámaras RGB semánticas y los LiDAR semánticos.

Para comenzar con las pruebas de rendimiento, se cargaron los mapas en orden ascendente empezando por *Town01_Opt*. En cada una de estas evaluaciones se creaba el vehículo con sus sensores, se tomaba control sobre él y se conducía el vehículo durante unos diez minutos. Las primeras conclusiones fueron preocupantes, y es que tan sólo los mapas número tres y número diez resultaban ser estables. El mapa número cuatro lograba aguantar pero por únicamente dos minutos aproximadamente y el resto de candidatos fallaban a los pocos segundos.

Con el objetivo de encontrar al menos un mapa más con el que trabajar, se decidió instalar candidatos adicionales desarrollados por CARLA Simulator. Esta opción la ofrecen en la misma página oficial y en su documentación y consiste en un total de cuatro mapas extra: *Town06_Opt*, *Town07_Opt*, *Town11* y *Town12*. Los dos últimos son ciudades masivas con un nuevo formato que no parecían si quiera cargar bien en el simulador desde el primer momento. Por otro lado, las otras dos opciones sí resultaron ser estables (ver Fig. 3.7).

Como conclusión a raíz de este análisis, se obtuvieron los siguientes datos:

- **Mapas estables:** Son las ciudades número tres, seis, siete y diez. No presentaron problemas al ser cargadas y aguantaron bien toda la carga de la configuración básica.
- **Mapas un poco inestables:** Este es el caso de *Town04_Opt*, que tuvo problemas para mantenerse en ejecución sin colgarse por más de un par de minutos.
- **Mapas inutilizables:** El resto de candidatos, incluidos *Town11* y *Town12*. Las simulaciones en estos escenarios apenas pueden durar unos segundos como mucho.

Figura 3.7: Sección de *Town06_Opt*. Fuente [9].

Es necesario destacar que *Town07_Opt* describe un entorno rural, lo cual lo hace un candidato inadecuado para hacer simulaciones en este proyecto (ver Fig. 3.8). Es por eso que las ciudades que se calificaron como utilizables para simulaciones sin sensores semánticos fueron *Town03_Opt*, *Town06_Opt* y *Town10HD_Opt*.

Figura 3.8: Vista en planta de *Town07_Opt*.

3.5.2. Análisis con sensores semánticos

En segundo lugar, se realizó un análisis de rendimiento del modelo digital al completo: tanto sensores normales como sus homólogos semánticos. El procedimiento fue exactamente el mismo al del apartado anterior, aunque las conclusiones fueron distintas:

- **Mapas estables:** Únicamente la ciudad *Town10HD_Opt*. No presentó problemas al ser cargada y aguantó bien toda la carga de la configuración básica.
- **Mapas un poco inestables:** Este es el caso de *Town06_Opt*, que tuvo problemas para mantenerse en ejecución sin colgarse por más de un par de minutos. Al contrario que antes, ahora *Town04_Opt* y *Town03_Opt* pasaron a ser totalmente inestables.
- **Mapas inutilizables:** El resto de candidatos. Las simulaciones en estos escenarios apenas pueden durar unos segundos como mucho.

Esto resultó en la preocupante cantidad de una sola ciudad con la que poder trabajar durante este TFG: *Town10HD_Opt*. Esto es entendible, ya que de los mapas pequeños, a pesar de ser el más detallado, es a su vez el más compacto y el que menos carga para el ordenador supone.

3.5.3. Análisis con sensores semánticos en CARLA Simulator 0.9.15

En la versión de CARLA Simulator 0.9.15 se incluyen nuevos mapas y características (ver Sección 2.3.2). Por esta razón se decidió hacer un análisis completo del rendimiento en los nuevos escenarios como el de la Sección 3.5.2.

Como resultado, los mapas gigantes *Town11*, *Town12*, *Town13* y *Town15* ahora sí parecían cargar bien. Sin embargo, los tres primeros candidatos o eran demasiado lentos a la hora de trabajar con ellos o directamente terminaban siendo inestables cuando se intentaba realizar simulaciones completas.

Por otro lado, a pesar de que fue ligeramente más lento trabajar con ella, *Town15*, supuso una nueva opción estable para poder realizar simulaciones y grabaciones con el modelo digital del vehículo y todos sus sensores.

El análisis en esta versión del simulador llevó a las mismas conclusiones sacadas en la Sección 3.5.2, con la excepción de *Tonw15* que se consideró como

mapa estable. Esta adición supuso poder realizar simulaciones en este TFG con un mapa más además de *Town10HD_Opt*.

3.6. Generación de agentes del tráfico externos

Una vez se comprobó que la idea de manejar el vehículo de forma manual con el volante y pedales disponibles era factible, el siguiente paso fue añadir funcionalidades al código ya existente para completar la calidad de las simulaciones. Por un lado, se hicieron las mismas modificaciones que con el manejo por teclado: tomar control del vehículo ya creado en vez de hacer aparecer un nuevo vehículo aleatorio. Por otro lado, los agentes externos que se planteó añadir fueron tanto otros vehículos como peatones. Para poder hacer esto se investigó en la documentación de CARLA Simulator acerca de cómo generar situaciones de tránsito genéricas con *traffic manager* o el controlador de peatones.

3.6.1. Vehículos NPCs

Como breve contextualización, en CARLA Simulator existen dos formas de sincronizar el serivdor con los clientes: la forma síncrona, en la que uno y sólo uno de los clientes marca el tiempo de simulación y la forma asíncrona, en la que el servidor es el que marca dicho tiempo. Si la estructura es multicliente, todos deberán funcionar de una forma u otra, sin mezclar tipos. En CARLA Simulator, el tiempo de simulación se marca activamente desde un cliente con el comando *world.tick()*. Por otro lado, si se espera que otro cliente o el servidor sea el que lo marque, se debe utilizar *world.wait_for_tick()*.

Ahora bien, *traffic manager* puede funcionar en ambas configuraciones. Sin embargo, el control de los vehículos en modo asíncrono es completamente caótico e irregular, lo cual dejó la alternativa síncrona como la única opción. En este segundo caso, además, *traffic manager* necesita que el cliente en el que es utilizado sea el que marque el tiempo de simulación para poder mover los vehículos.

Por tanto, para añadir la funcionalidad previamente mencionada al código principal, se cambió nuevamene el archivo *manual_toyota_steering_and_traffic.py* para convertirlo en un cliente que funcionara en modo síncrono y marcase el tiempo de simulación. Además, en él se configuraron las condiciones de funcionamiento de *traffic manager*, al cual a continuación se le brindaba el control sobre vehículos NPCs generados por todo el mapa (ver Fig. 3.9). Como detalle, la cantidad de vehículos a generar es especificada por el usuario al ejecutar el código .

Figura 3.9: Ejemplos de vehículos NPCs creados.

El proceso de generar estos vehículos consiste en seleccionar un *blueprint* y un *spawn point* aleatorios para cada uno e intentar crearlos bajo dichas condiciones. Es importante destacar que si el número de NPCs que se quiere generar es lo suficientemente alto (30 por ejemplo), lo más seguro es que la variedad de los agentes sea suficiente. Esto es, contendrá motocicletas, bicicletas, camiones, furgonetas o turismos. Aún así, en la parte de generación de estos NPCs, el código está escrito de forma que asegure la creación de por lo menos algunas bicicletas.

3.6.2. Peatones

La creación y el control de los peatones en CARLA Simulator es distinto y un poco más complicado. Lo primero que salta a la vista es que no existen *spawn points* correspondientes con localizaciones típicas de transeúntes, como aceras. Esto implica que los puntos de creación tienen que ser definidos manualmente con las coordenadas exactas en el mapa y que no existe una lista predefinida en cada ciudad. Para saber qué coordenadas exactas representan ciertas localizaciones de interés en el mapa, basta con situarse en ellas con el espectador dentro del simulador y ejecutar el siguiente script: *spectator_location.py*. Se desarrolló dicho código como utilidad para aligerar los procesos de creación de peatones.

Para el control de los transeúntes debe obtenerse el *blueprint* de un controlador de peatones de CARLA Simulator y adherirlo a cada uno de ellos (como

si de añadir un sensor a un vehículo se tratara). Por último se le especifica una ruta a seguir y se inicia el control. Si no se aclara el trayecto que deben ejecutar, entonces recorrerán un camino aleatorio.

En este proyecto, para crear tráfico de peatones realista en las simulaciones se desarrolló el programa *spawn_pedestrians.py*. En él, se generan aproximadamente 20 transéuntes en distintas localizaciones, según se estén utilizando los mapas *Town03_Opt*, *Town06_Opt* o *Town10HD_Opt*. A dichos actores se les añaden sus respectivos controladores y finalmente se activan para que caminen por rutas aleatorias durante todo el tiempo que dure la simulación en cuestión (ver Fig. 3.10).

Finalmente, es importante destacar que para el control de estos actores, el cliente en el que se contempla esta tarea no tiene por qué marcar el tiempo de simulación. De hecho, en el códidgo descrito en el párrafo anterior, el cliente espera al tiempo de simulación con *world.wait_for_tick()*.

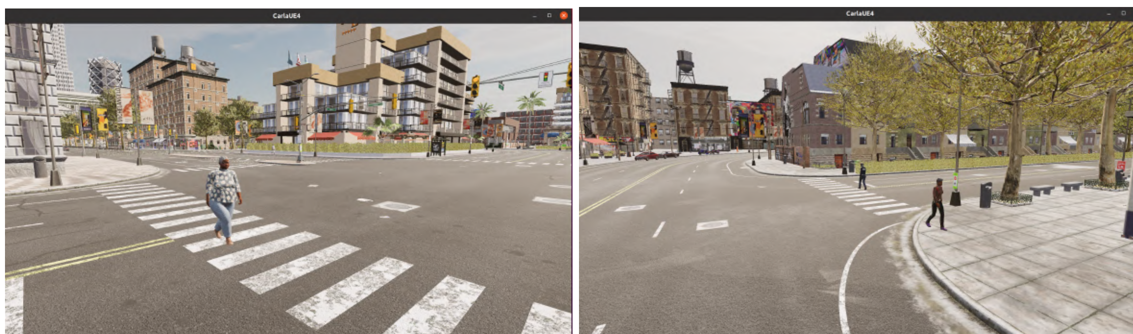

Figura 3.10: Ejemplos de peatones creados.

3.6.3. Agentes externos en CARLA Simulator 0.9.15

Al migrar todo el código escrito para CARLA Simulator 0.9.14 a esta nueva versión, se detectó un nuevo malfuncionamiento a la hora de crear los agentes externos para el tráfico. El problema residía en que *generate_pedestrians.py* dejó de funcionar por un motivo que no se supo diagnosticar.

La solución fue utilizar uno de los códigos de ejemplo en la *PythonAPI* de CARLA Simulator que genera tráfico tanto de vehículos como de transeúntes *generate_traffic.py*. Como este cliente sería el que ahora generaría automóviles, al usar el controlador manual *manual_toyota_steering_and_traffic.py* se debería especificar a cero la cantidad de NPCs que se quiera crear.

Capítulo 4

ROS2 para combinar CARLA Simulator y redes neuronales

Contenido

4.1 Introducción a las redes neuronales para la detección de obstáculos 39
4.1.1 Redes CNN 40
4.1.2 Redes DETR 41
4.2 Redes candidatas para procesar datos 43
4.2.1 Redes para procesamiento de imágenes RGB 43
4.2.2 Redes para procesamiento de nubes de puntos 3D . . . 44
4.3 Incorporación en CARLA Simulator a través de ROS2 46
4.3.1 Nodo para redes YOLO 47
4.3.2 Nodo para redes DETR 47
4.3.3 Nodo para redes DETR en ROS2 Humble 48
4.3.4 Nodo para redes de detección 3D 49

4.1. Introducción a las redes neuronales para la detección de obstáculos

Las redes neuronales son un subconjunto del *machine learning*, compuestas por capas de nodos, entre las cuales destacan una capa de entrada, una o más

ocultas y otra de salida. Cada nodo está conectado como mínimo a otro y dicha conexión tiene asociado un peso sináptico y un umbral de activación.

El acelerado avance tecnológico en el ámbito de los vehículos autónomos ha impulsado la investigación y desarrollo de sistemas de percepción avanzados. Entre las tecnologías clave que posibilitan la autonomía vehicular, resulta vital la detección precisa de obstáculos para garantizar la seguridad y la eficacia de los vehículos autónomos. En este contexto, las redes neuronales se han destacado como herramientas fundamentales, capaces de procesar datos sensoriales complejos y proporcionar información en tiempo real del entorno del automóvil.

En este subapartado de introducción a modelos de aprendizaje profundo, se aborda la evaluación de dos enfoques primordiales para la detección de obstáculos: Redes Neuronales Convolucionales (CNN) y las red *Transformer* de Detección (DETR).

Mientras que las CNN han demostrado su eficacia en la detección precisa de objetos, especialmente con datos de cámaras RGB, la innovadora aproximación de DETR redefine la tarea al abordarla desde una perspectiva de establecimiento de correspondencias, eliminando la necesidad de seleccionar regiones de interés.

4.1.1. Redes CNN

Las redes neuronales de convolución comenzaron a ser implementadas para unidades de procesamiento gráfico (GPUs) ya en 2012. Desde entonces han sido refinadas y mejoradas logrando así resultados cada vez más precisos.

Estos modelos se caracterizan por simular una estructura neuronal simplificada de un cerebro biólogico en el campo de la percepción sensorial. En concreto, se aplican especialmente al procesamiento de imágenes y voz. Su arquitectura se basa en tres claras partes: Capas de convolución, capas de *pooling* y capa totalmente conectada (ver Fig. 4.1).

- **Capas de convolución:** es donde se realiza la extracción de características de los datos de entrada. Suelen ser mútiples capas, comenzando por la búsqueda de patrones más sencillos y desembocando en formas y figuras más grandes y/o complejas. Los datos de entrada, en el caso de imágenes RGB, son el alto y el ancho de la imagen así como el color de cada píxel.
- **Capas de *pooling*:** estas capas pueden ir tras las de convolución aunque suelen aparecer de forma intercalada. Son las encargadas de reducir la dimensión del mapa de características generado por la capa de convolución.

Se da por tanto una pérdida de información sobre el análisis matemático de la entrada, pero solamente lo suficiente como para evitar que la red se sobreespecialice en predecir las imágenes con las que es entrenada. A cambio, se obtiene una mejora del rendimiento, una disminución del coste computacional para capas posteriores y se limita el riesgo de sobreajuste.

- **Capa totalmente conecatada:** es la última capa previa a la salida. Se caracteriza por realizar la clasificación de las características recopiladas en las capas anteriores y dar dicho resultado a la salida de la red.

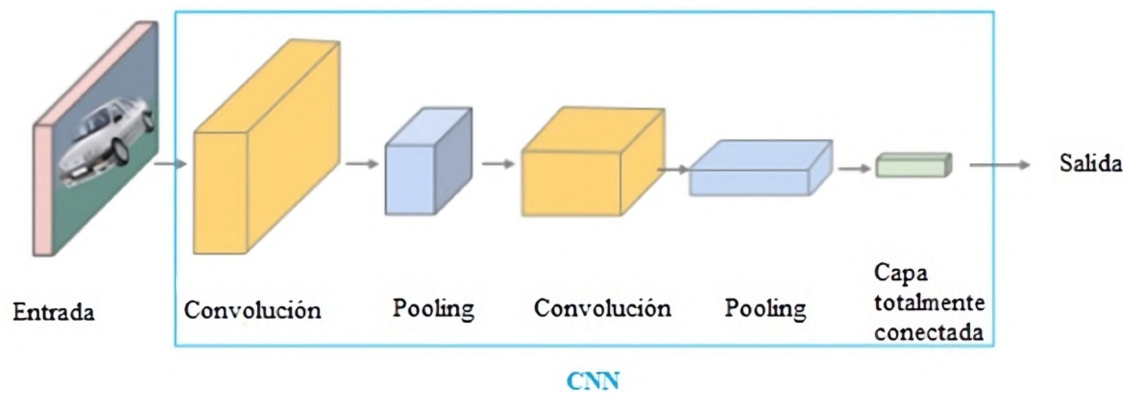

Figura 4.1: Arquitectura CNN. Fuente [10].

Sin embargo, el modelo original de CNN deja sin resolver múltiples problemas complicados como la repetición de cajas de clasificación o *bounding boxes*. Es por eso que muchos algoritmos totalmente basados en ella han sido publicados hasta día de hoy modificando y añadiendo nuevos elementos a la estructura en forma de parches para ir resolviendo todo estas dificultades. El resultado son algoritmos de alta eficacia pero cada vez más complejos, como es el caso de *You Only Look Once* (YOLO) [28].

4.1.2. Redes DETR

El origen de las DETR proviene de una estructura codificador-decodificador utilizada principalmente para el procesamiento del lenguaje natural (NLP). Las redes *Transformer* surgieron oficialmente en 2017 con la propuesta de una mejora de la estructura previa con un mecanismo de atención paralelo multi-cabeza. Además se comenzó a considerar la posibilidad de extender el campo de aplicación de esta tecnología al procesamiento de voz e imagen. Esto terminó de

consolidarse en 2021 con la introducción de las *DEtection TRansformer* (DETR), especializadas en la detección en visión por computador.

Las DETR poseen una arquitectura mucho más simple que la de los algoritmos previos totalmente basados en CNN como YOLO. Su estructura (ver Fig. 4.2) se puede dividir en varias secciones:

- **CNN backbone:** es decir, utiliza un transfondo de las redes convolucionales únicamente para la extracción de cualidades de la imagen de entrada.
- **Codificador:** o en inglés *encoder*, está compuesto por un codificador posicional y un codificador de transformer. Este último, gracias a su propiedad llamada *atención*, crea relaciones entre cada una de las partes de la imagen. Dicho principio es bastante utilizado en NLP para relacionar unas palabras con otras. El formato de los datos de entradas a este codificador debe ser el de un vector plano. Para aplanarlo, se le aplica previamente a las características extraídas de la imagen el codificador posicional.
- **Decodificador:** o en inglés *decoder*, compuesto por un decodificador de transformer que posee como entrada un número fijo de N consultas de objetos. A su salida, cada una de estas consultas contiene ya la información de las predicciones. Llegan a este punto *observando* cada uno de los vectores salientes del *encoder* que se introducen como entradas laterales al decodificador.
- **Cabezales de predicción:** son los clasificadores finales. Se encargan de traducir la información saliente del decodificador en tuplas de dos elementos: la clase del objeto y su *bounding box*.

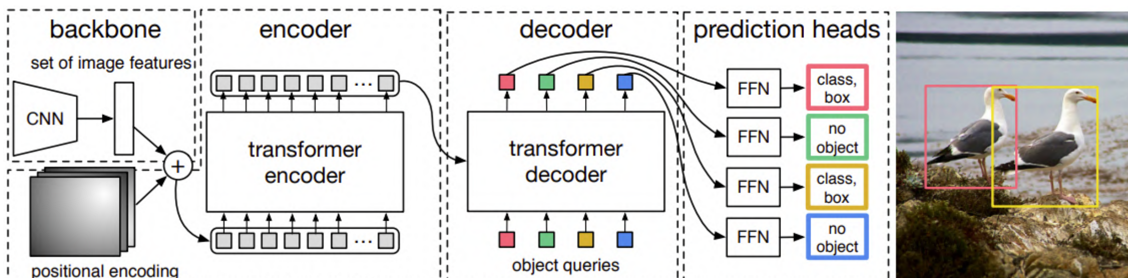

Figura 4.2: Arquitectura DETR. Fuente [11].

La gran ventaja de esta arquitectura es que, sin necesidad de añadir ningún parche, resuelve con un esquema razonablemente sencillo todos los problemas que se daban con la estructura original de las CNN.

Por otro lado, la principal desventaja para este proyecto es que las DETR necesitan de una altísima capacidad computacional para ser entrenadas. Sin embargo, se planteó desde el principio que, en caso de encontrar algún modelo preentrenado que pueda ser ya directamente utilizado para el procesamiento de los datos generados en CARLA Simulator, podría ser una gran opción.

4.2. Redes candidatas para procesar datos

Después de hacer una formación básica sobre aprendizaje profundo y algunos tipos de red, el siguiente paso fue buscar modelos compatibles con la interfaz montada hasta aquel momento en el proyecto. La tarea que se buscó completar desde el inicio fue la de detección de obstáculos. Ya bien fuera por segmentación semántica, por medio de *bounding boxes*, u otro método razonablemente válido.

4.2.1. Redes para procesamiento de imágenes RGB

YOLOv8

La octava versión de *You Only Look Once* (YOLO) [29] desarrollada por *Ultralytics* es la primera solución que se va a considerar por su fácil implementación y uso. Este modelo en estado del arte puede ser utilizado para tareas de detección de obstáculos, clasificación de imágenes y segmentación de instancias. Se puede aplicar en tiempo real y ofrece el mejor rendimiento entre todas las versiones.

El algoritmo YOLO, originalmente publicado en 2015, plantea la detección de obstáculos como un problema de regresión en vez de como una tarea de clasificación. Esto lo hace utilizando una única CNN y los pesos sinápticos de su última versión se pueden obtener a través de la librería *ultralytics* [29] desarrollada en *Python*.

DETR-RESNET

Este es el caso de una red *transformer* para la detección de obstáculos que utiliza modelos *RESNET* como *backbone*.

El modelo exacto que se quiera poner a prueba con sus pesos sinápticos correspondientes se puede extraer del repositorio oficial de *Facebook* en la plataforma *Hugging Face* [30] (ver Tabla 4.1). Esta última representa una comunidad

y librería muy popular para modelos de NLP y con ello redes DETR, que usan el mismo principio.

A partir de esta fuente, se pueden utilizar gran variedad de alternativas. Desde aquellos modelos publicados por el equipo de *Facebook AI* hasta otros personalizados por usuarios más experimentales.

Nombre del modelo	*Backbone*	*Training dataset*
DETR-RESNET-50	RESNET50	COCO
DETR-RESNET-50-DC5	RESNET50 DC5	COCO
DETR-RESNET-101	RESNET101	COCO
DETR-RESNET-101-DC5	RESNET101 DC5	COCO
DETR-RESNET-101-PANOPTIC	RESNET101	COCO *panoptic*

Tabla 4.1: Modelos destacables de *Facebook* en *Hugging Face*

Los modelos preentrenados utilizados para el procesamiento de datos en las simulaciones de este TFG han sido *detr-resnet-50*, *detr-resnet-50-dc5*, *detr-resnet-101* y *detr-resnet-101-dc5*. Estas redes son las más populares aplicables a la detección de agentes externos del tráfico.

4.2.2. Redes para procesamiento de nubes de puntos 3D

PV-RCNN

El modelo de red *PointVoxel Regional based Convolutional Neural Network* (PV-RCNN) [12] es una variación de las CNN adaptada al análisis de nubes de puntos tridimensionales (ver Fig. 4.3). En este caso, se origina en 2019 a través de la fusión de *3D Voxel* CNN y *PointNet*, un algoritmo propuesto inicialmente en 2016 para el procesamiento directo de nubes de puntos usando un método basado en vóxeles.

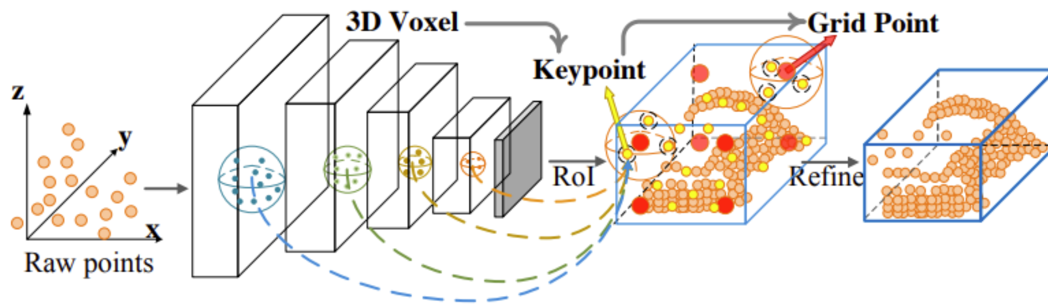

Figura 4.3: Arquitectura de PV-RCNN. Fuente [12].

Dentro de los modelos más utilizados para la detección de obstáculos 3D validados con *KITTI dataset* [22], PV-RCNN presenta la mayor precisión en la detección de coches. Sólo por debajo de *Voxel RCNN (Car)*, que no ha sido probada para el caso de peatones y ciclistas (Tabla 4.2).

Part-A2-Free

Part-A2-Free es una de las posibilidades propuestas como modelo en la publicación del algoritmo *3D Part-Aware and Aggregation Neural Network* (Part-A2-Net) [13]. Este algoritmo base fue publicado en 2019 y divide su estructura en dos partes principales. Por un lado la etapa de consciencia parcial, dedicada a la detección principal de los obstáculos dentro de la nube de puntos y por otro lado la etapa de agregación parcial, dedicada al cálculo de confianza y refinamiento de las *bounding boxes* (ver Fig. 4.4).

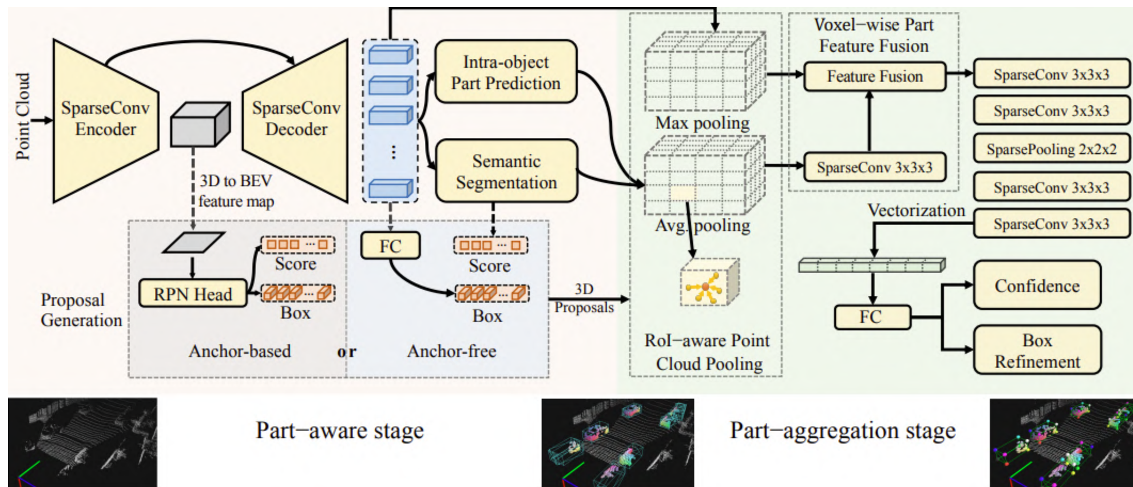

Figura 4.4: Arquitectura de Part-A2-Net. Fuente [13].

En este TFG se hizo uso de la variación libre de anclas o *anchor-free*, llamada Part-A2-Free, ya que ofrece mejores resultados generales en la detección de agentes del tráfico que con la variante Part-A2-Anchor. Además, dentro de los modelos más utilizados para la detección de obstáculos 3D validados con *KITTI dataset*, ofrece la mayor precisión media para la detección de autocares, peatones y ciclistas (ver Tabla 4.2).

Nombre del modelo	Coches (%)	Peatones (%)	Ciclistas (%)
PV-RCNN	83.61	57.90	70.47
Voxel RCNN (Car)	84.54	—	—
Part-A2-Free	78.72	65.99	74.29
Part-A2-Anchor	79.40	60.05	69.90

Tabla 4.2: Precisión de los modelos citados en la detección 3D de agentes

4.3. Incorporación en CARLA Simulator a través de ROS2

La forma que se pensó para proceder fue la de incluir los modelos de red neuronal a través de nodos de ROS2. De esta forma, podría simularse el rendimiento de las redes al procesar imágenes o nubes de puntos en tiempo real. Los nodos están escritos en *Python* y la biblioteca principal que se utilizó escrita en este lenguaje de programación para el manejo de redes neuronales fue *PyTorch*.

El código fuente asociado a estos nodos de ROS2 está accesible y disponible para revisión en el repositorio de *GitHub* correspondiente. Aquellos interesados pueden acceder al código en [27].

4.3.1. Nodo para redes YOLO

En este caso, se puede hacer uso de cualquier modelo YOLO a través de la librería *ultralytics* desarrollada por *Ultralytics Team*. Su implementación es sencilla:

- La carga del modelo se hace en una línea en la función constructora del nodo (ver Listado 4.1).
- Por otro lado se utiliza un subscriptor sencillo de ROS2 para detectar nuevas imágenes llegando por los tópicos especificados.
- Cada vez que llega una imagen, se adapta el formato de ROS2 a *NumPy* [31], aquel que utiliza la red YOLO para sus datos de entrada.
- Finalmente, también en el *callback* del subscriptor, se hace inferencia con cada imagen nueva que se detecta en el tópico en cuestión y se guardan los resultados en un directorio concreto.

En este TFG el código fue desrrollado específicamenta para *YOLOv8*, la versión más actualizada de estas redes y la que se decidió utilizar. Si por algún motivo se quiere hacer uso otra versión de YOLO, se deberán hacer mínimos cambios en la carga e inferencia con el modelo según la nomenclatura que indiquen los mismos desarrolladores de la red.

Listado 4.1: Carga típica del modelo *YOLOv8* con *Python*.

```
from ultralytics import YOLO

#Cargar el modelo de CNN
self.model = YOLO("yolov8n.pt")
```

4.3.2. Nodo para redes DETR

Para las redes DETR, el esquema del nodo de ROS2 es similar al anterior. Como diferencias a marcar, se utilizó para este nodo la librería *transformers* desarrollada por *The Hugging Face Team* para cargar el modelo de DETR deseado.

Los pasos a seguir siguen la misma base, pero son algo más complejos que para la red *YOLOv8*. Por un lado, con *transformer* se cargó el *checkpoint* del modelo deseado. Esto es, sus pesos sinápticos de la red preentrenada. Para utilizar las DETR con estas librerías se tuvo que cargar tanto el modelo como un procesador de imágenes para acomodar el formato de los datos de entrada (ver Listado 4.2).

Además, cada vez que el nodo recibía una imagen a través del subscriptor, se había de convertir de formato ROS2 a *cv2* [32] en este caso. Y finalmente, una vez hecha la inferencia, había que realizar un postprocesamiento de los resultados para plasmarlos en la imagen inicial.

Listado 4.2: Carga de DETR con un *backbone* CNN con *Python*.

```
from transformers import DetrForObjectDetection, DetrImageProcessor

#Carga del procesador de imagenes y del detector de obstaculos
self.image_processor = DetrImageProcessor.from_pretrained(
    CHECKPOINT)
self.model = DetrForObjectDetection.from_pretrained(CHECKPOINT)
```

En el código desarrollado, para modificar la CNN del *backbone* que se quiera utilizar en el modelo de DETR, basta con cambiar el nombre de la variable *CHECKPOINT* al del modelo subido en *The Huggin Face*

4.3.3. Nodo para redes DETR en ROS2 Humble

Durante la actualización a ROS2 Humble, lo único que presentó más dificultad fue el nodo desarrollado previamente para la incorporación de las redes DETR.

El código presentaba errores en dos secciones:

- **Creación de etiquetas para las detecciones:** Tras el cambio de versiones el sistema de creación de etiquetas presentaba diferencias en el formato de salida cuando en la imagen procesada se habían dado detecciones o no. Se solucionó adaptando el formato de etiquetado para cada uno de ambos escenarios con una lógica *try-except* (ver Listado 4.3).
- **Procesamiento de imágenes "vacías":** Con ROS2 Humble, en este nodo saltaban errores cuando se detectaba una imagen vacía o sin datos por errores en la grabación. Antes, esto lo dejaba pasar de largo, pero ahora se tuvo que incorporar una lógica *if-else* para solventarlo manualmente y no ejecutar el algoritmo de procesamiento si la imagen de entrada era inválida.

Listado 4.3: Estructura *try-except* para adaptar el etiquetado de las detecciones.

```
try:
    #Procesamiento anterior de las imagenes (solamente cuando no
       hay detecciones)
except:
    #Procesamiento con adaptacion del formato de etiquetado para
       imagenes con detecciones
```

En el resto de los aspectos, este nodo quedó exactamente igual al descrito en la Sección 4.3.2.

4.3.4. Nodo para redes de detección 3D

En el caso de la detección en nubes de puntos, se optó por utilizar una interfaz ya existente en forma de paquete de ROS2 llamada *pcdet_ros2*. Se trata de un *wrapper* de ROS2 para *OpenPCDet* publicado en marzo de 2023 [26].

El nodo principal que constituye este paquete es *pcdet_node.py*, donde por un lado se crea un subscriptor de ROS2 que obtiene datos de las nubes de puntos a analizar en formato *PointCloud2*. Esta información la adapta al formato típico que se suele utilizar para la entrada de estas redes de detección 3D y la procesa por la red en cuestión. Por último publica las cajas de detección fabricadas en otro tópico de ROS2.

Además existen otros componentes dentro de la estructura del paquete dedicados a la configuración de la red que se vaya a emplear, el formato de los datos, los tópicos con los que se va a trabajar o los pesos sinápticos y arquitectura que se quieren cargar en el modelo.

Los modelos preentrenados que se querían utilizar se descargaron en formato *.pth*. Para poder hacer uso de ellos, se debían situar en el directorio *checkpoints* dentro del mismo paquete *pcdet_ros2* (ver Fig. 4.5).

cfgs	Working Version	9 months ago
checkpoints	Add checkpoints folder.	9 months ago
config	Working Version	9 months ago
docs	Working Version	9 months ago
launch	Working Version	9 months ago
pcdet_ros2	Working Version	9 months ago
resource	Working Version	9 months ago
test	Working Version	9 months ago
.gitattributes	Show Language	9 months ago
.gitignore	Working Version	9 months ago
LICENSE	Working Version	9 months ago
README.md	Dependencies for ros2_numpy	9 months ago
package.xml	Working Version	9 months ago
setup.cfg	Working Version	9 months ago
setup.py	Working Version	9 months ago

Figura 4.5: Estructura de *pcdet_ros2* visto desde el repositorio en *Github*.

Parte III

Análisis final y conclusiones

Capítulo 5

Puesta en marcha y análisis de resultados

Contenido

5.1 Introducción 53
5.2 Realización de simulaciones 54
5.2.1 Condiciones de conducción seleccionadas 54
5.2.2 Grabación de datos con ROS2 *Bag* 55
5.3 Resultados del procesamiento de datos 55
5.3.1 Simulación 1 55
5.3.2 Simulación 2 63
5.3.3 Simulación 3 69

5.1. Introducción

En este capítulo se aborda todo el proceso de puesta en marcha de las interfaces de CARLA Simulator y ROS2 creadas para la realización de simulaciones realistas.

Además, se realiza un análisis de los resultados producidos por la redes neuronales seleccionadas para el procesamiento de imágenes RGB y nubes de puntos de LiDAR 3D generados desde CARLA Simulator.

5.2. Realización de simulaciones

Una vez todas las interfaces fueron preparadas y los nodos para utilizar las redes fueron codificados, se pasó a realizar las simulaciones en CARLA Simulator. La idea desde el principio fue grabarlas con la herramienta ROS2 *Bag* para poder recurrir a los datos generados por los sensores en cualquier momento. Es decir, se decidió construir un pequeño *dataset* de imágenes RGB y nubes de puntos basado en CARLA Simulator.

5.2.1. Condiciones de conducción seleccionadas

Como se ha visto en las Secciones 3.5.2 y 2.3.2 de esta memoria, las ciudades seleccionadas para probar la eficiencia de las redes candidatas fueron *Town10HD_Opt* y *Town15*.

Ahora bien, las condiciones meteorológicas que se quisieron utilizar fueron básicas. Lo suficiente como para poder probar el rendimiento de la red en cuestión en escenarios bastante diversos dentro del entorno urbano. Por ende, se pensó inicialmente en realizar simulaciones ante condiciones soleadas, de lluvia diurna, de niebla diurna y de noche despejada. Sin embargo, por problemas de renderizado, no se simulaban bien los casos de lluvia y niebla. Así que se optó finalmente por sólo realizar simulaciones despejadas diurnas y nocturnas (ver Fig. 5.1).

Para poder establecer estos estados del tiempo, se utilizó en todo momento el código proporcionado en la *PythonAPI* de CARLA Simulator *environment.py*. Por otro lado, cabe destacar que las modificaciones en las condiciones meteorológicas sólo afectan a las cámaras RGB, ya que ni la luz, ni la niebla, ni la lluvia modifican el comportamiento del LiDAR en CARLA Simulator.

Figura 5.1: Condiciones de las simulaciones en *Town10HD_Opt*. Fuente [14].

5.2.2. Grabación de datos con ROS2 *Bag*

Los nodos descritos previamente en la Sección 4.3 fueron preparados para recibir datos directamente de tópicos de ROS2. Esto significa que, conforme dicha información fuera llegando en tiempo real, las redes neuronales elegidas tendrían que procesarlos bajo las mismas condiciones de tiempo que lo haría el ordenador abordo del vehículo real sensorizado.

Ahora bien, existen varias formas de hacer que las imágenes y nubes de puntos circulen por los tópicos de ROS2. Por un lado, se pueden hacer simulaciones que utilicen cámaras y sensores. Por otro lado, se puede grabar todo el flujo de datos de dicha simulación para posteriormente reproducirlo a antojo del usuario. La herramienta que se ha utilizado para ello es la ya descrita ROS2 *Bag* (Sección 2.4.2).

La idea fue realizar una grabación con este medio en ambos mapas descritos. Por un lado se hicieron dos simulaciones en *Town10HD_Opt* y por otra parte se hizo una sola simulación en *Town15*, donde el escenario es más similar al de un entorno universitario pero la densidad de tráfico lograda es menor.

5.3. Resultados del procesamiento de datos

Con un fin explicativo, se ha decidido dividir los resultados en función de la simulación a la que pertenezcan.

5.3.1. Simulación 1

El primer escenario fue *Town10HD_Opt* con tiempo diurno y despejado. Se simularon aproximadamente 30 vehículos NPCs y 25 peatones. La conducción manual se realizó durante 3 minutos respetando señales de tráfico y conduciendo tanto a velocidades lentas como más moderadas. CARLA Simulator se cargó con un nivel de calidad de gráficos bajo para que el equipo fuera capaz de soportarlo.

Detección 2D

La detección de obstáculos por parte de las 5 redes candidatas probadas fue funcional y aplicable a tiempo real. Esto se pudo observar de forma sencilla reproduciendo todas las imágenes procesadas fotograma a fotograma como si fuera un vídeo, obteniendo como resultado una secuencia fluida de la simulación.

Por un lado, YOLOv8 presentó especialmente una alta velocidad durante la inferencia con imágenes. En comparación con otras redes, el procesamiento es aceptable, pero menos preciso y con menor índice de confiabilidad en todas las detecciones (ver Figs 5.2).

Dirigiendo la atención a las redes DETR, se apreció en seguida una mayor potencia de detección de obstáculos. Para hacerlo más obvio, el límite menor de confianza que se aplicó a estas redes fue de un 85 % frente al incluso 30 % de YOLOv8. Aún así, se registraron más detecciones de la mano de las redes *transformer*.

Haciendo más incapié en estos últimos modelos, las 4 redes mostraron resultados similares, siendo *detr-resnet-50* y *detr-resnet-101*, en ese orden, las más rápidas con diferencia con respecto a sus variantes *dc5*. En el caso de *detr-resnet-50*, se pudo observar una ligera menor exactitud con las detecciones realizadas (ver Fig. 5.3). Por otro lado, a pesar de ser la más lenta de las *transformers*, *detr-resnet-101-dc5* sobresalía también ligeramente por presentar mayor potencia de detecciones (ver Fig. 5.6). Finalmente, y como punto medio a las dos anteriores, quedaron con detecciones similares *detr-resnet-50-dc5* y *detr-resnet-101*, donde la segunda fue claramente más rápida (ver Figs. 5.4 y 5.5).

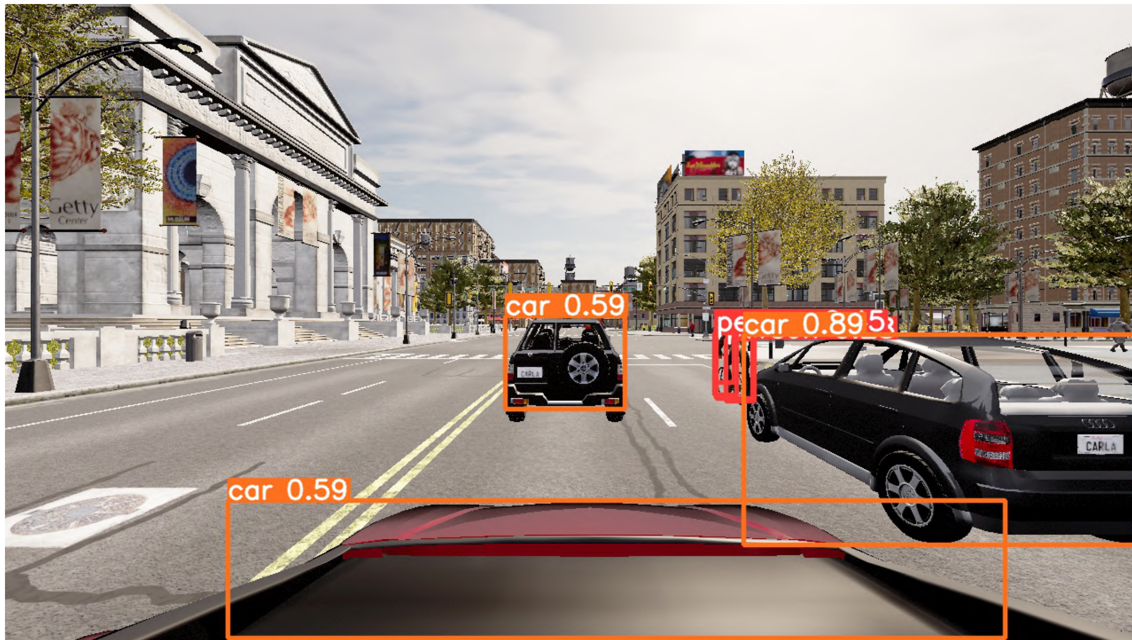

Figura 5.2: Procesamiento de *YOLOv8* en la simulación 1.

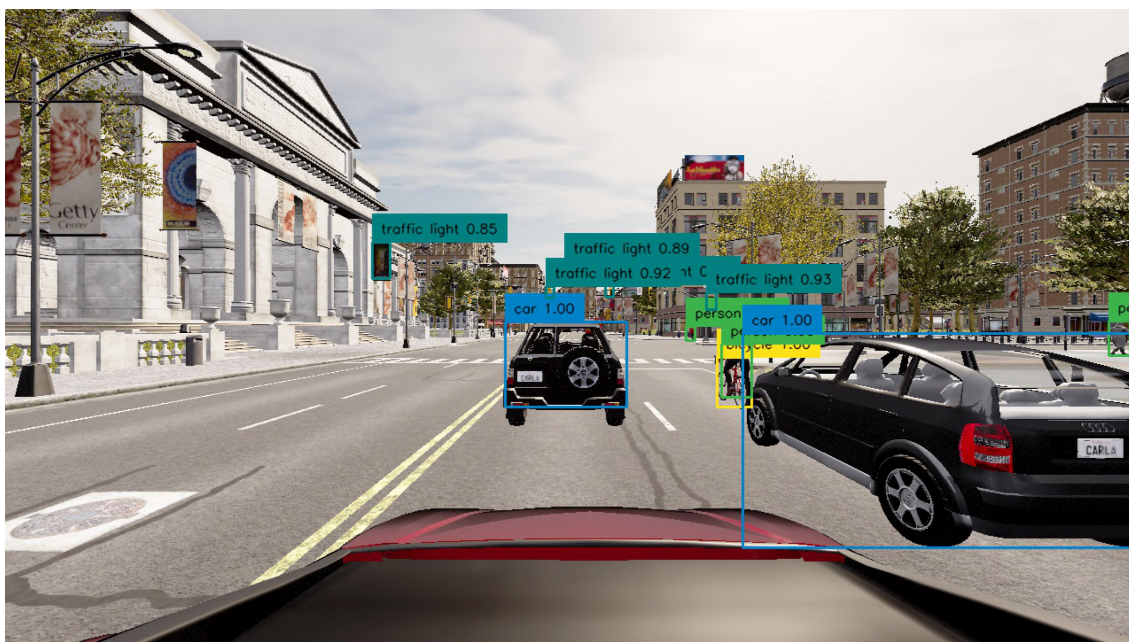

Figura 5.3: Procesamiento de *detr-resnet-50* en la simulación 1.

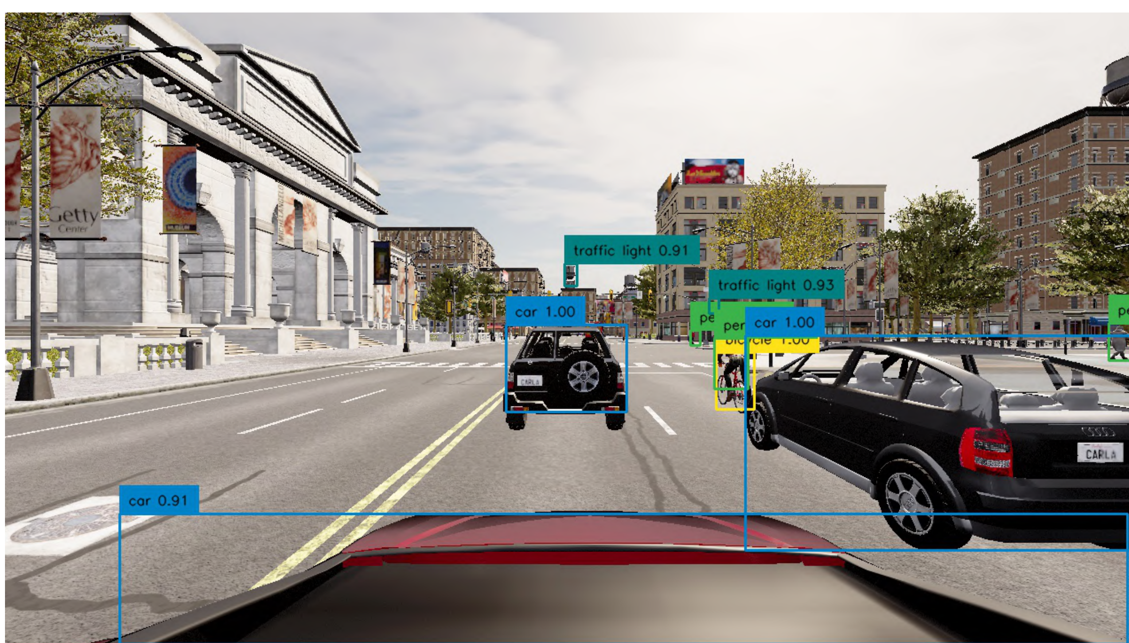

Figura 5.4: Procesamiento de *detr-resnet-50-dc5* en la simulación 1.

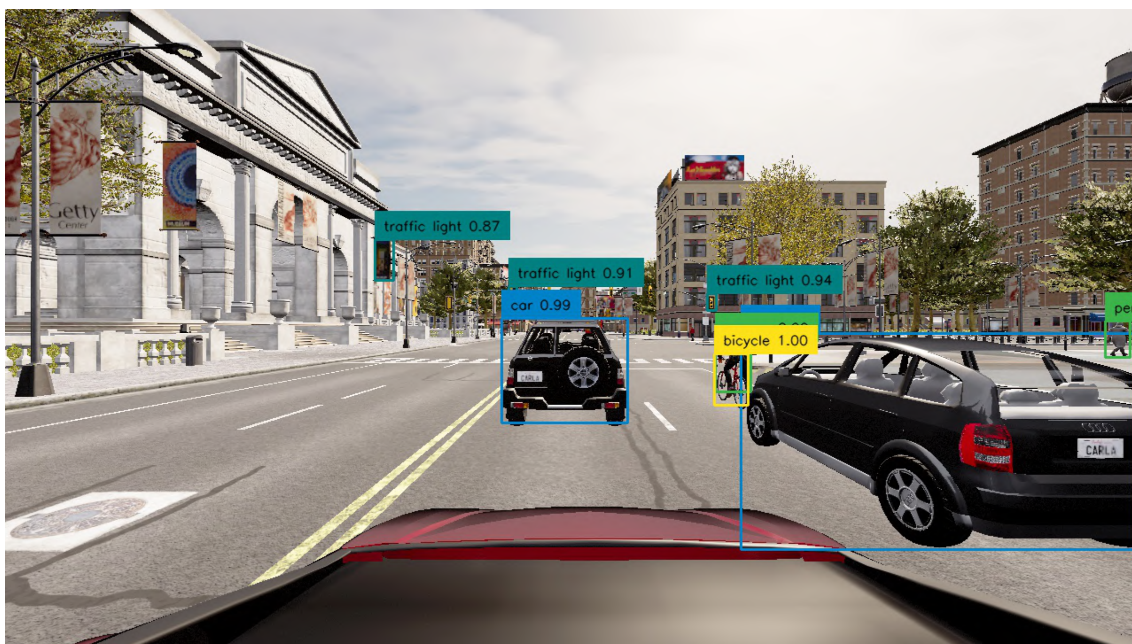

Figura 5.5: Procesamiento de *detr-resnet-101* en la simulación 1.

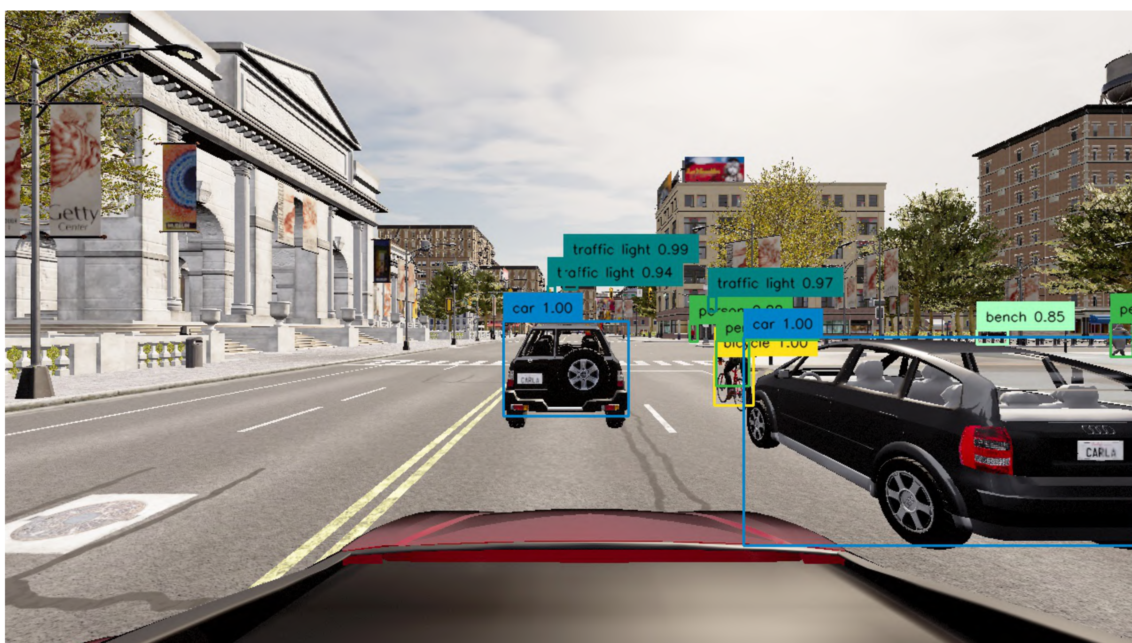

Figura 5.6: Procesamiento de *detr-resnet-101-dc5* en la simulación 1.

Detección 3D

Las redes de procesamiento de nubes de puntos tridimensionales mostraron en la primera simulación su correcta funcionalidad, aunque de menor precisión que en la detección 2D. Por otro lado, se observaron también una menor cantidad de etiquetas en la búsqueda de obstáculos: coches, bicicletas y peatones.

La primera característica que se pudo extraer a raíz de la simulación es la invalidez de ambas redes neuronales para la detección de grandes camiones, como por ejemplo para vehículos de bomberos o ambulancias (ver Figs. 5.7 y 5.8).

En segundo lugar, *Part-A2-Free* mostró una capacidad superior de detección de bicicletas, generando cajas más precisas y más frecuentes ante este tipo de agente de tráfico. Por otro lado, *PV-RCNN*, aunque más exacta para la detección de autocares, se topó con mayor dificultad para la localización de bicicletas. En las Figuras 5.9 y 5.10 se plasma la diferencia de rendimiento entre ambas redes en una situación donde una bicicleta se sitúa a la derecha del vehículo sensorizado.

Finalmente, se comenzó a observar una incapacidad de detección de peatones tanto por parte de *PV-RCNN* así como *Part-A2-Free*, que sostenía tener una mejor capacidad de detección de transeúntes (ver Figs. 5.11, 5.12 y 5.13). Una posible explicación de este comportamiento, es que en la nube de puntos generada por el LiDAR, no se aprecian alteraciones en los lugares en los que debería haber un peatón.

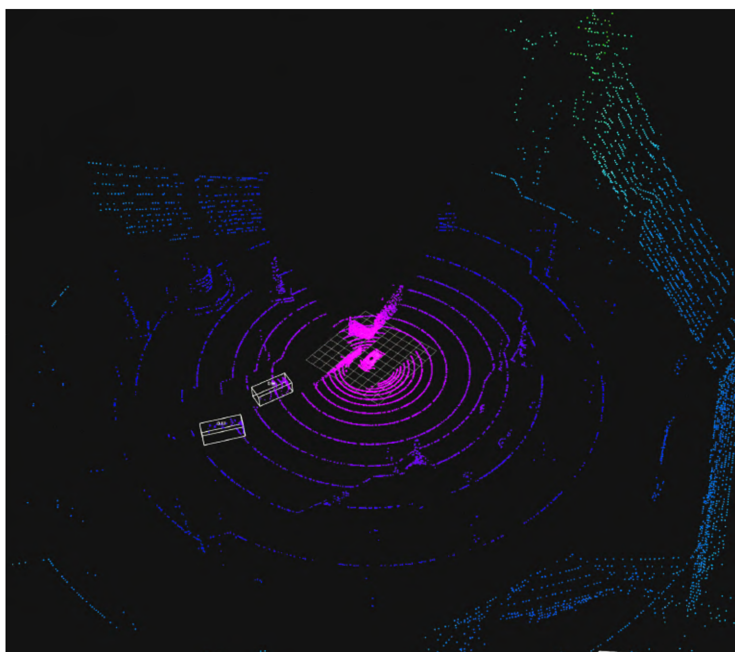

Figura 5.7: Detección de vehículos grandes de PV-RCNN en simulación 1.

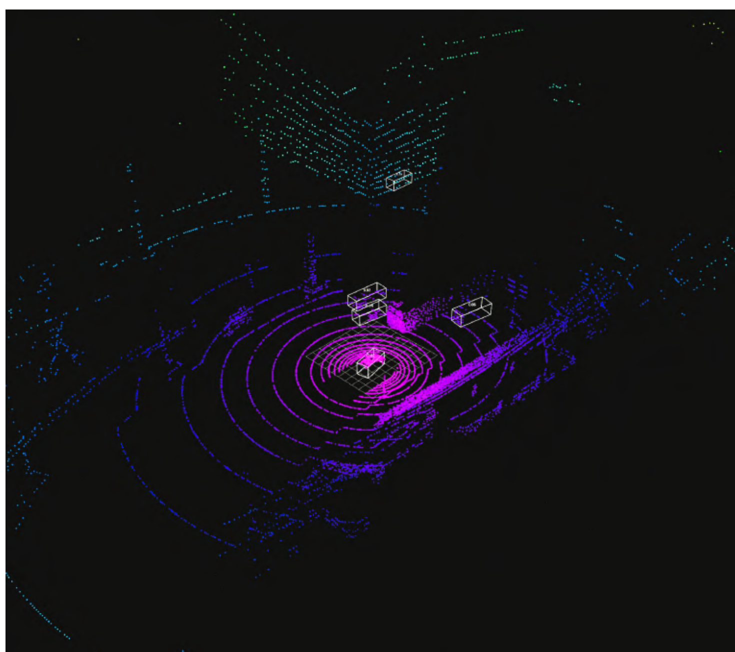

Figura 5.8: Detección de vehículos grandes de Part-A2-Free en simulación 1.

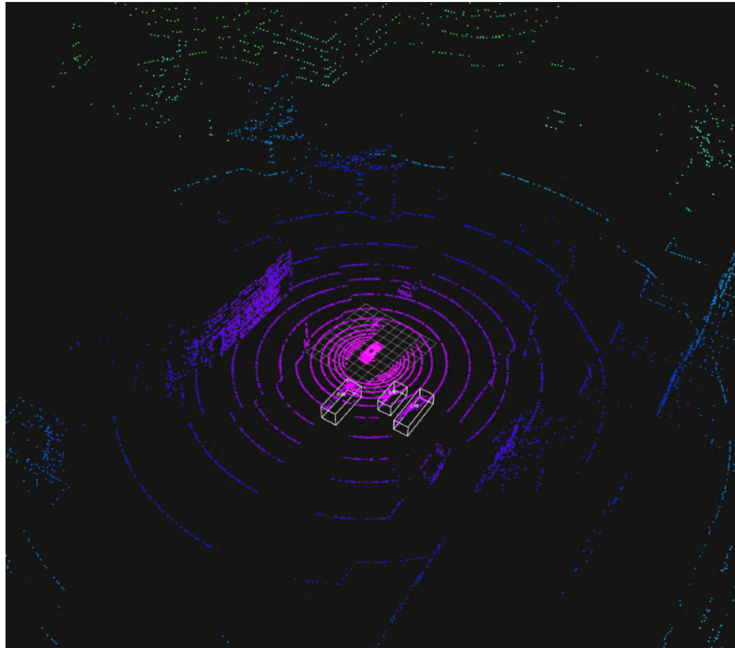

Figura 5.9: Detección de bicicletas de PV-RCNN en la simulación 1.

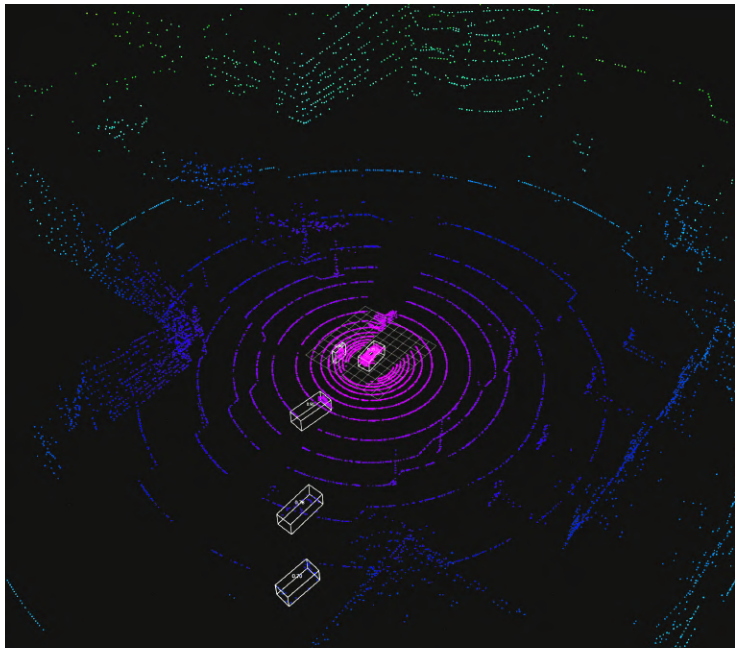

Figura 5.10: Detección de bicicletas de Part-A2-Free en la simulación 1.

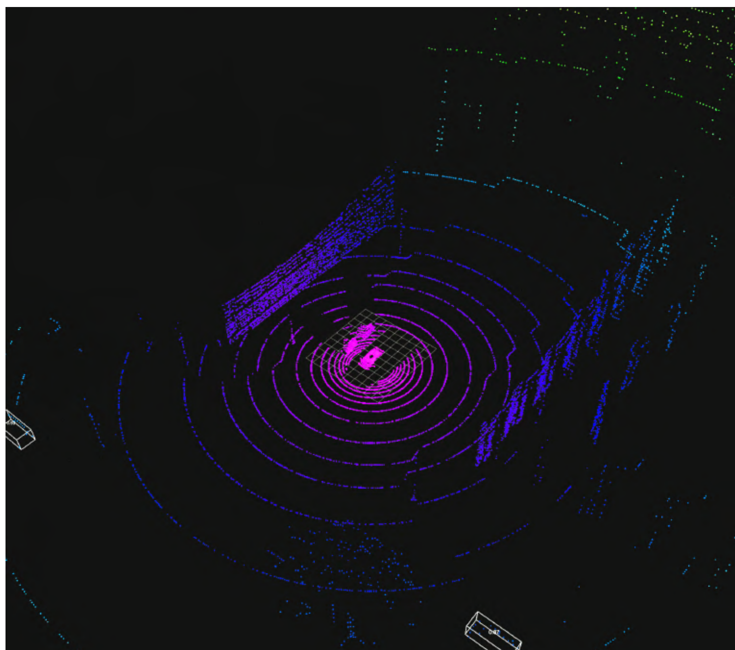

Figura 5.11: Detección de peatones de PV-RCNN en la simulación 1.

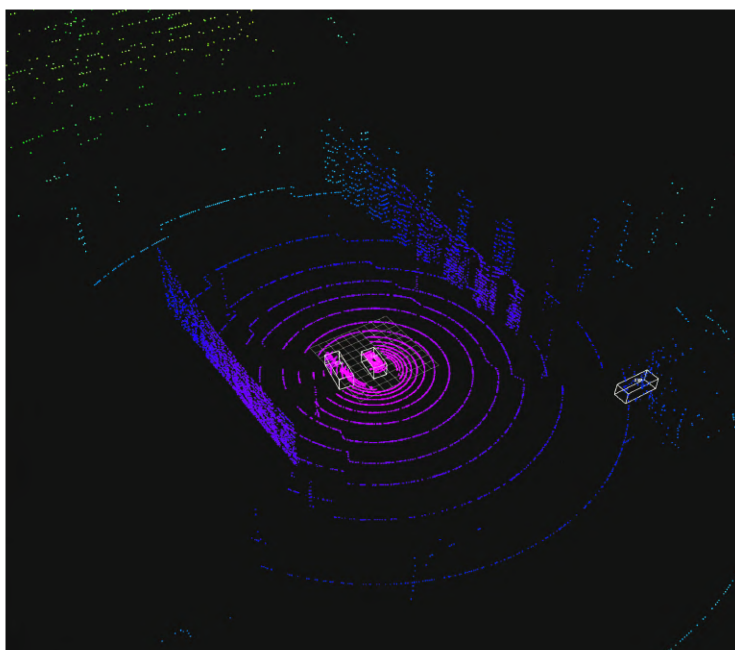

Figura 5.12: Detección de peatones de Part-A2-Free en la simulación 1.

Figura 5.13: Peatón captado por cámara RGB en la simulación 1.

5.3.2. Simulación 2

El segundo escenario fue de nuevo *Town10HD_Opt* pero con tiempo nocturno y despejado esta vez. Se simularon aproximadamente la misma cantidad de agentes externos que en la primera simulación. La conducción manual se realizó durante unos 5 minutos y 5 segundos respetando señales de tráfico y conduciendo tanto a velocidades lentas como más moderadas. CARLA Simulator se volvió a cargar con un nivel de calidad de gráficos bajo para que el equipo fuera capaz de soportarlo.

Detección 2D

Sorprendentemente, las detecciones no se vieron apenas afectadas por la condición nocturna de esta simulación. Los resultados, por ende, fueron bastante similares a los anteriores. Se comenzó a observar un mayor rendimiento por parte de las redes *transformer* ante densidades de tráfico moderadas.

YOLOv8 presentó nuevamente menor potencia de detección en comparación a las demás redes además de un índice de confiabilidad más bajo que en la primera simulación (ver Fig. 5.14). *Detr-resnet-50* y *detr-resnet-50-dc5* mostraron cierta inexactitud a la hora de detectar semáforos y confundiéndolos con carteles

colgantes (ver Figs. 5.15 y 5.16). Por otro lado, *detr-resnet-101* y *detr-resnet-101-dc5* fueron las que procesaron mejores detecciones (ver Figs. 5.17 y 5.18).

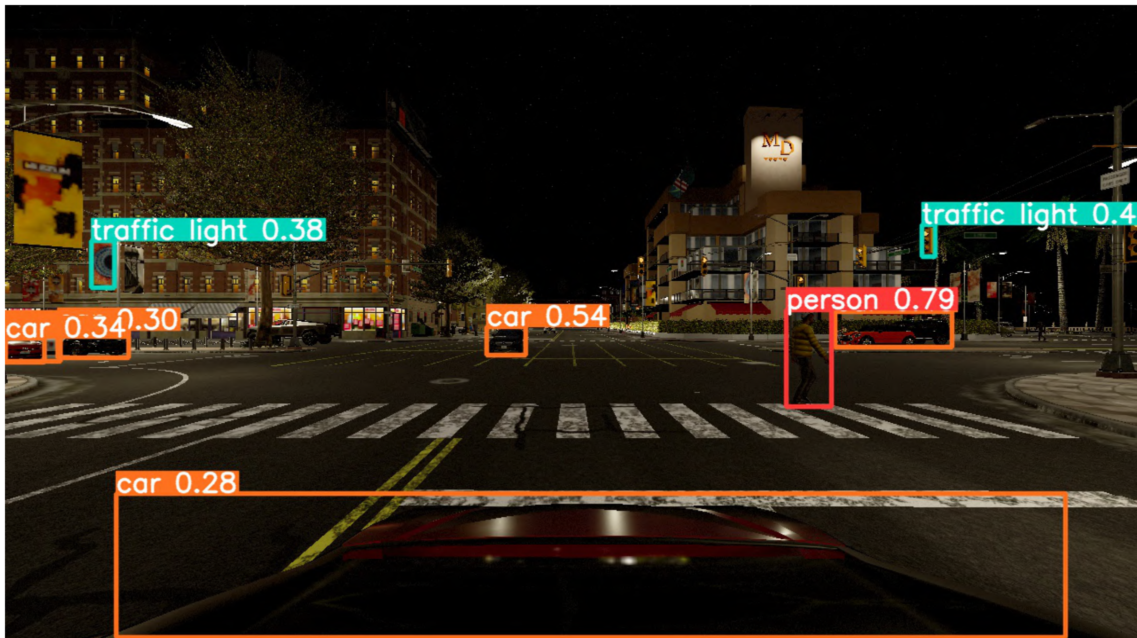

Figura 5.14: Procesamiento de *YOLOv8* en la simulación 2.

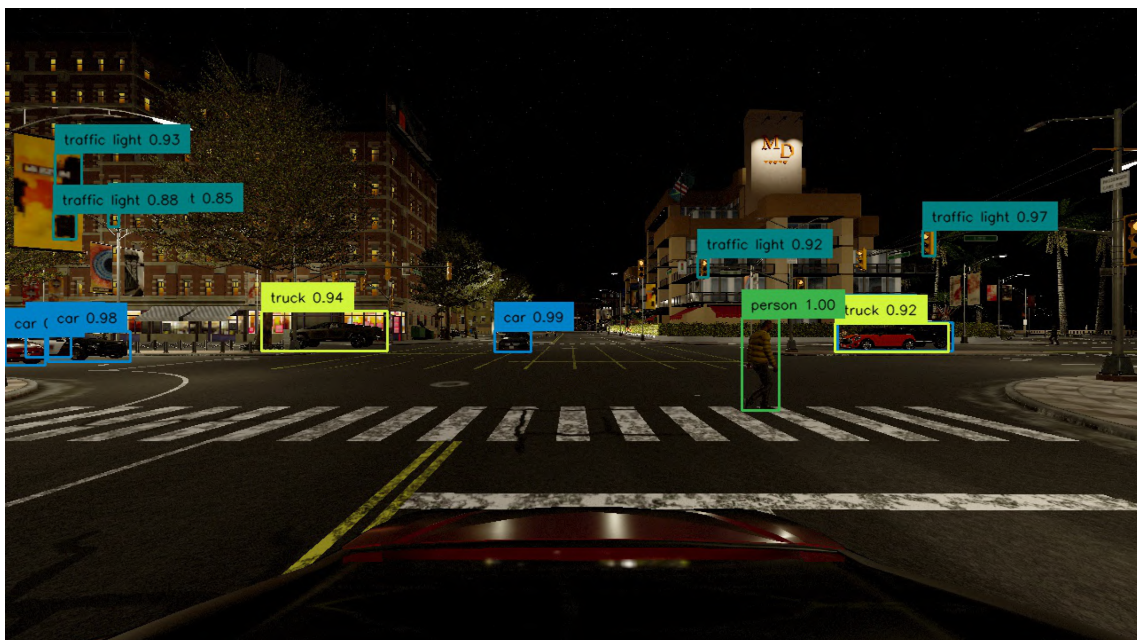

Figura 5.15: Procesamiento de *detr-resnet-50* en la simulación 2.

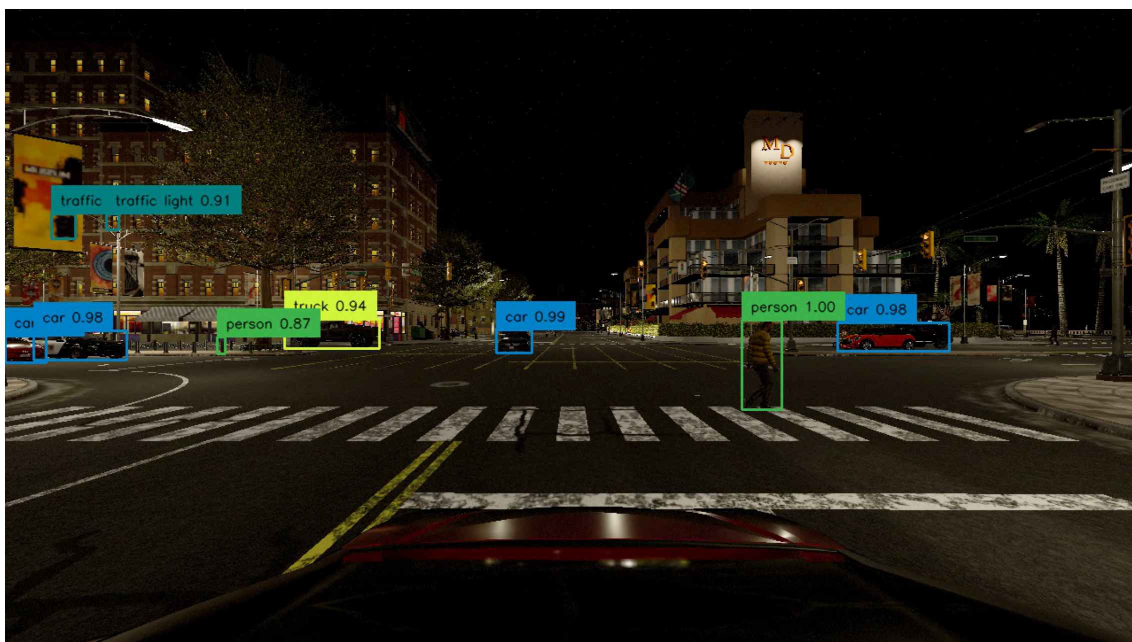

Figura 5.16: Procesamiento de *detr-resnet-50-dc5* en la simulación 2.

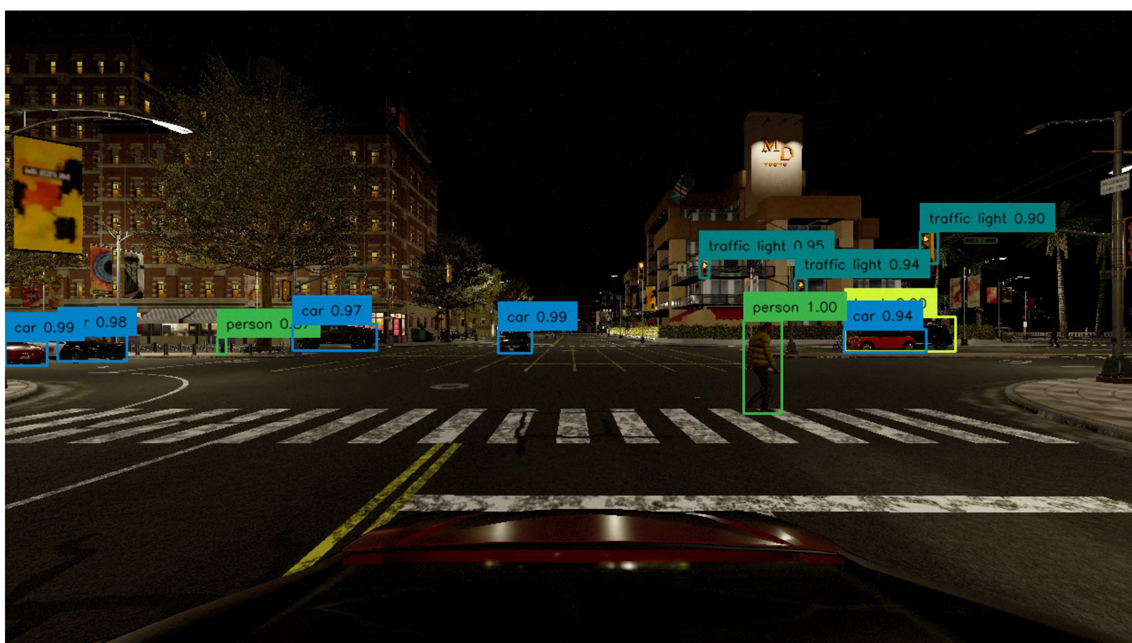

Figura 5.17: Procesamiento de *detr-resnet-101* en la simulación 2.

Figura 5.18: Procesamiento de *detr-resnet-101-dc5* en la simulación 2.

Detección 3D

En la segunda simulación en *Town10HD_Opt*, se observaron resultados idénticos a los de la anterior prueba. Esto sucede ya que, en CARLA Simulator, los cambios meteorológicos no afectan al rendimiento del LiDAR. Dicho fenómeno puede suponer un inconveniente para el estudio del comportamiento de los sensores en situaciones adversas.

A modo de comprobación, se volvieron a apreciar buenas detecciones de vehículos típicos, como turismos o monovolúmenes de tamaño moderado (ver Figs. 5.19 y 5.20). Por otro lado, de nuevo se presentaron dificultades en la localización de vehículos grandes como camiones de bomberos o ambulancias.

Por último, el problema de la detección de peatones se repitió, confirmando así las sospechas que se dieron en la primera simulación (ver Figs. 5.21, 5.22 y 5.23).

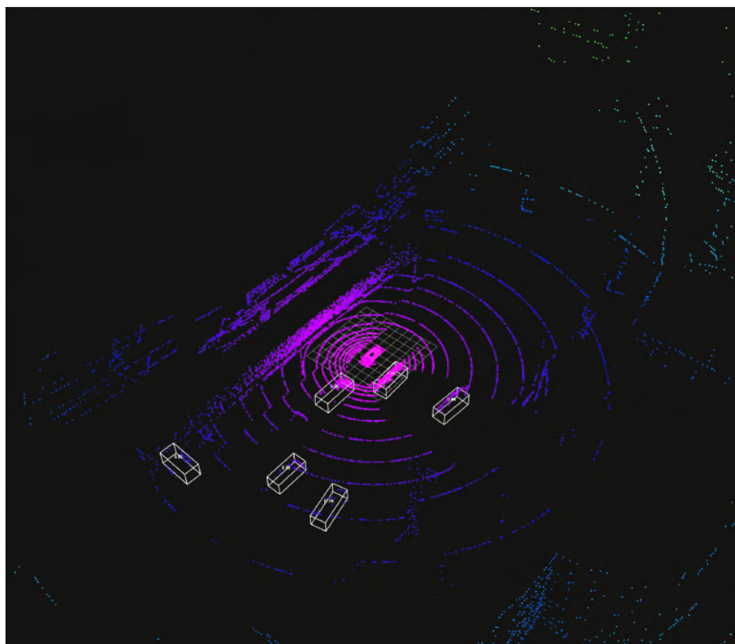

Figura 5.19: Ejemplo de buena detección 3D de PV-RCNN en simulación 2.

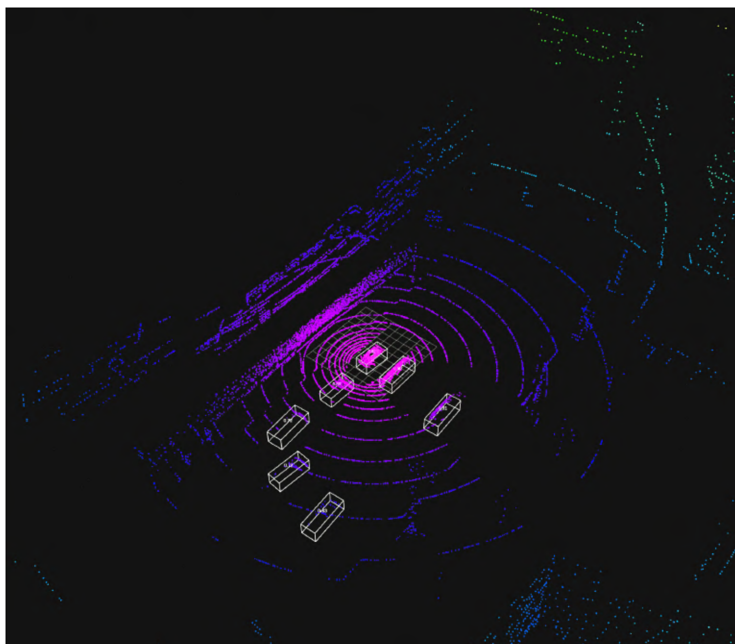

Figura 5.20: Ejemplo de buena detección 3D de Part-A2-Free en simulación 2.

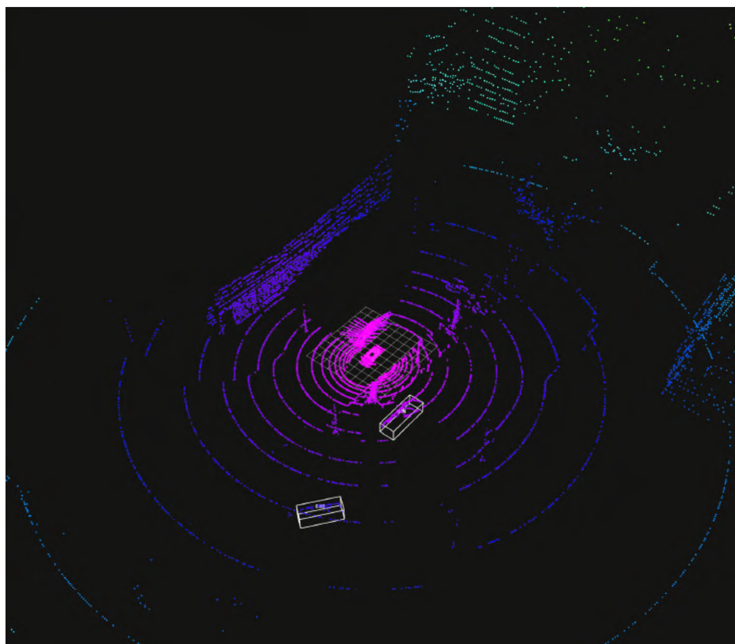

Figura 5.21: Detección de peatones de PV-RCNN en la simulación 2.

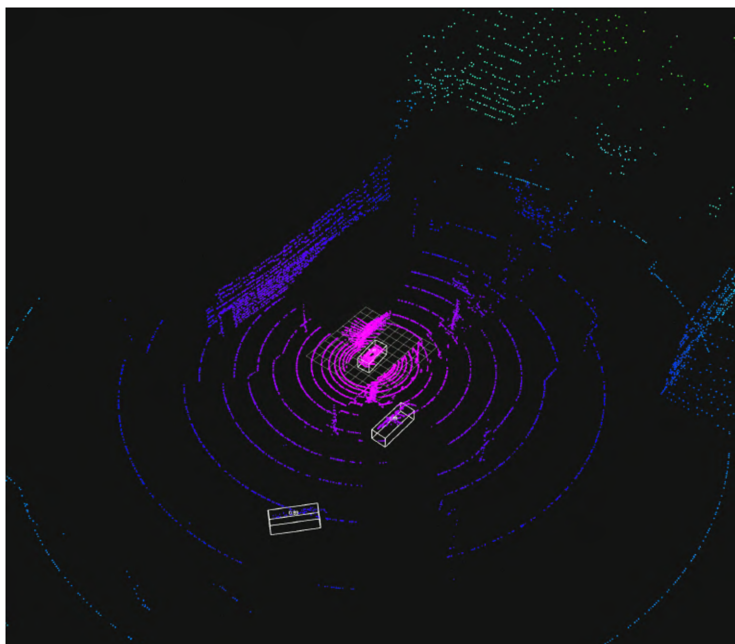

Figura 5.22: Detección de peatones de Part-A2-Free en la simulación 2.

Figura 5.23: Peatón captado por cámara RGB en la simulación 2.

5.3.3. Simulación 3

El tercer escenario fue *Town15* con tiempo diurno y despejado. Se simularon aproximadamente 150 vehículos NPCs y 100 peatones. A pesar de esta elevada cifra de agentes del tráfico externos, al ser el mapa mucho más grande que *Town10HD_Opt*, la densidad de vehículos y peatones móviles fue bastante menor. La conducción manual se realizó durante 6 minutos y 55 segundos respetando señales de tráfico y conduciendo tanto a velocidades lentas como más moderadas. CARLA Simulator se cargó con un nivel de calidad de gráficos alto esta vez, aunque no supuso diferencia aparente alguna en cuanto al análisis de las detecciones.

Detección 2D

Ante una menor densidad de tráfico, la diferencia de potencia y exactitud entre un modelo de red y otro se vio aminorada. Ciertamente se pudo observar en las simulaciones una gran cantidad de coches aparcados, los cuales por lo general no supusieron un problema para ninguna de las candidatas.

YOLOv8 siguió mostrando ser la más rápida de las 5 y la de menor potencia y exactitud con un margen razonable (ver Fig. 5.24). Por otro lado, el análisis con

las 4 redes *Transformer* se volvió más ambiguo y semejante, dando resultados muy similares entre ellas (ver Figs. 5.25, 5.26, 5.27 y 5.28). De nuevo, el orden de rapidez en la inferencia de mayor a menor fue el siguiente:

1. **YOLOv8:** La más rápida.
2. **Detr-resnet-50**
3. **Detr-resnet-101**
4. **Detr-resnet-50-dc5**
5. **Detr-resnet-101-dc5:** La más lenta.

Figura 5.24: Procesamiento de *YOLOv8* en la simulación 3.

Figura 5.25: Procesamiento de *detr-resnet-50* en la simulación 3.

Figura 5.26: Procesamiento de *detr-resnet-50-dc5* en la simulación 3.

Figura 5.27: Procesamiento de *detr-resnet-101* en la simulación 3.

Figura 5.28: Procesamiento de *detr-resnet-101-dc5* en la simulación 3.

Detección 3D

Durante la tercera simulación la densidad de tráfico fue bastante menor en comparación a las dos anteriores. Sin embargo, se dieron casos de circuilación por vías con una alta cantidad de coches estáticos, es decir, aparcados.

Lo descrito permitió poner a prueba la capacidad de ambas redes neuronales ante situaciones con gran número de detecciones. Se pudo observar en seguida que las dos consiguieron generar tantas cajas de clasificación como fueran necesarias de manera exitosa (ver Figs. 5.29 y 5.30). Sin embargo, salió a la luz de forma más evidente que las detecciones ocurren con mucha más frecuencia por delante del vehículo, dándose en la parte trasera pocos casos.

En cuanto al resto de características, no se destacó nada más que no hubiera ocurrido en las pruebas previas, como la detección de vehículos grandes, bicicletas o peatones. En general, la exactitud de ambas redes fue similar durante toda la simulación.

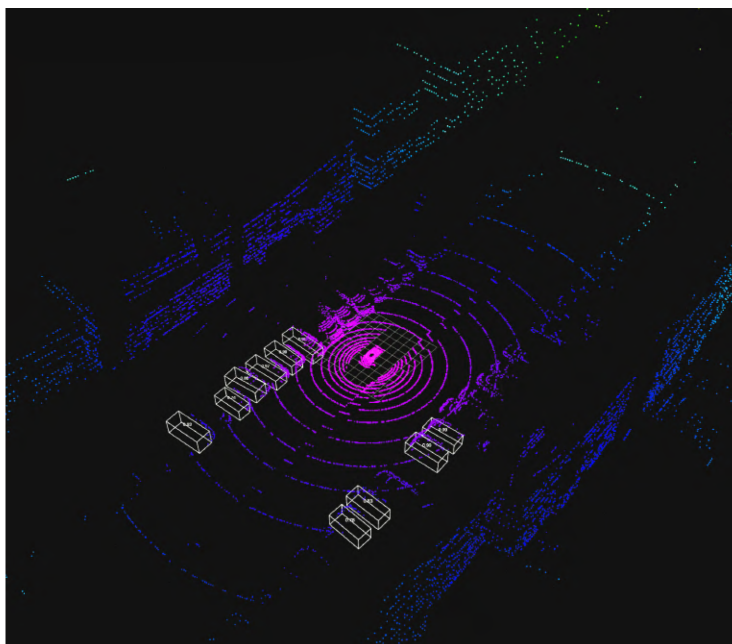

Figura 5.29: Detección de PV-RCNN de múltiples objetos en la simulación 3.

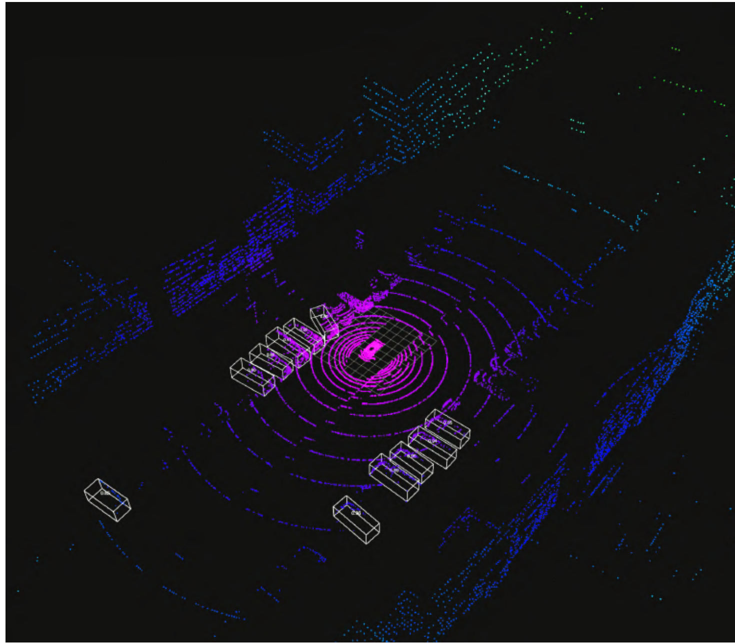

Figura 5.30: Detección de Part-A2-Free de múltiples objetos en la simulación 3.

Capítulo 6

Conclusiones y líneas futuras

Contenido

6.1 Conclusiones 75
6.2 Líneas futuras 76

6.1. Conclusiones

Este TFG se enmarca dentro de las labores de investigación del departamento de Ingeniería de Sistemas y Automática en el proyecto PREMOVE. En particular, el objetivo era el de realizar un primer estudio sobre las estrategias disponibles para la detección de agentes del tráfico en entornos urbanos.

En primer lugar, se ha elaborado una interfaz completa en CARLA Simulator para la realización de simulaciones ante situaciones de tráfico realistas. El control manual del vehículo sensorizado y la generación de agentes externos ha resultado en una mejora directa de las simulaciones generadas en el TFG de Daniel Gamba. A través de esta interfaz y utilizando la herramienta ROS2 *Bag*, se han grabado hasta 3 simulaciones en entornos urbanos que pueden ser accedidas en cualquier momento.

Por otra parte se han hecho depuraciones del nodo fusión codificado en el TFG previo para generar un modelo digital del LiDAR RS-Helios 5515. Además, se han añadido al modelo digital los sensores semánticos RGB y LiDAR homólogos a los ya presentes. Inclusive, se ha desarrollado otro nodo fusión para dicho nuevo LiDAR.

En tercer lugar, se han instalado, modificado y desarrollado los paquetes de

ROS2 necesarios para generar una interfaz capaz de procesar los datos generados por los sensores en el simulador. Múltiples redes neuronales han sido puestas a prueba para la detección 2D y 3D.

Entre las redes para detección de obstáculos en imágenes RGB, han destacado como mejores candidatas las redes DETR. Han ofrecido un rendimiento y precisión mucho mayores a los de la más tradicional *YOLOv8*, a cambio de reducir la velocidad de inferencia en una magnitud lo suficientemente baja. Entre estas innovadoras redes, la menos eminente ha sido *detr-resnet101-dc5*, que ha ofrecido un desempeño muy similar al resto pero ha sido la más lenta sin duda. Por otro lado, *detr-resnet-50-dc5* ha generado resultados muy parecidos a los de *detr-resnet-101* aunque a una menor velocidad. Esto nos ha dejado como conclusión a considerar como mejores candidatas a *detr-resnet-50* y *detr-resnet-101*, que además de ser las más populares, han sido las que han ofrecido una mejor relación precisión-rapidez en las simulaciones.

En cuanto a las nubes de puntos, la red que ha ofrecido un mejor rendimiento ha sido *Part-A2-Free*. El argumento se ha basado en una exactitud similar con respecto a *PV-RCNN* para la detección de automóviles de tamaño moderado. Sin embargo, es en la localización de bicicletas donde destaca claramente *Part-A2-Free* como clara ganadora ante la precisión superior que ha ofrecido. Aunque, por otro lado, ninguna de las dos redes previamente mencionandas ha sido capaz de realizar detecciones de peatones durante las 3 simulaciones presentadas.

En cuarto orden, es importante destacar que el equipo de laboratorio se ha visto en su límite durante la realización de las simulaciones. Esto se debe a que ha habido múltiples procesos activos simultáneamente y bastante demandantes. Se ha podido observar en las grabaciones cómo algunas secciones suceden menos fluidas de lo normal, es decir, con saltos notables entre un fotograma y otro.

El código fuente asociado al TFG está accesible y disponible para revisión en el repositorio de *GitHub* correspondiente. Aquellos interesados pueden acceder al código en [27].

6.2. Líneas futuras

Como posibles líneas futuras a raíz del trabajo realizado se proponen las siguientes:

- Considerar dividir la carga de trabajo en varias computadoras. Por ejemplo, una podría ir dedicada a la realización de simulaciones y otra a la grabación

de estas. Otra opción sería dividir los procesos de CARLA Simulator entre varios dispositivos, algo que resulta posible desde la versión de CARLA Simulator 0.9.15.

- Utilizar los sensores semánticos incorporados al modelo digital para realizar procesos de evaluación cuantitativa y entrenamiento de las redes neuronales que se pongan a prueba. Los datos que generan estos dispositivos digitales están ya etiquetados por el mismo simulador, algo que facilita enormemente el trabajo.

- Incorporar un mapa del campus de Teatinos cercano a la Escuela de Ingenierías Industriales utilizando el motor *Unreal Engine* para usarlo en CARLA Simulator. Se puede afrontar esta tarea modificando, mejorando y adaptando el ya creado en el TFM de Eloy Vergara [33]. Además, sería interesante investigar el uso de gemelos digitales para la generación del mapa, una característica que ha sido introducida en la última versión de CARLA Simulator 0.9.15.

- Sincronizar la nube de puntos y las imágenes RGB procesadas para potenciar la detección de agentes del tráfico. Por otro lado, se podría considerar la investigación de otros modelos preentrenados de red neuronal que puedan estar más especializados en esta aplicación.

- Añadir mayor variedad de agentes del tráfico a la simulación. Por ejemplo, personas con movilidad reducida utilizando muletas, sillas de ruedas, bastón, entre otros. También incorporar otros medios de transporte cada vez más frecuentes en la zona universitaria de Málaga como patinetes o motocicletas eléctricas.

Parte IV

Apéndices

Apéndice A

Manual de instalación de Software

Contenido

A.1 Contextualización . . . 81

A.2 CARLA Simulator distribución *package* . . . 82

A.2.1 Descarga del paquete . . . 82

A.2.2 Instalación de la librería del cliente de CARLA Simulator . 82

A.2.3 Instalación de mapas adicionales . . . 83

A.3 ROS2 Foxy . . . 84

A.3.1 Añadir repositorio . . . 84

A.3.2 Instalación del paquete ROS2 . . . 84

A.3.3 Descarga de paquetes externos para este TFG . . . 85

A.4 *Carla-ros-bridge* . . . 88

A.5 Librerías y código base para uso de redes neuronales . . . 88

A.5.1 CUDA 12.1 . . . 89

A.5.2 *PyTorch* . . . 89

A.5.3 *Ultralytics* . . . 90

A.5.4 *Transformers* . . . 90

A.5.5 *OpenPCDet* . . . 90

A.1. Contextualización

Todas las instalaciones se hicieron en el sistema operativo Linux/Debian, distribución Ubuntu 20.04 LTS. Todo el código desarrollado para este TFG se en-

cuentra en el repositorio de *GitHub* correspondiente [27].

A.2. CARLA Simulator distribución *package*

En este TFG realmente es únicamente necesario usar la distribución *package* de CARLA Simulator, que es la que se explicará a continuación.

Para consultar algún detalle en concreto o instalar la distribución *source*, acudir a la documentación oficial [34].

A.2.1. Descarga del paquete

En primer lugar, se ha de seleccionar la versión que se quiera utilizar y buscarla en el repositorio de CARLA Simulator en *GitHub* [35].

Para la versión 0.9.14, descargar el archivo *CARLA_0.9.14.tar.gz* (ver Fig. A.1)

Release 0.9.14

- `[Ubuntu]` CARLA_0.9.14.tar.gz
- `[Ubuntu]` AdditionalMaps_0.9.14.tar.gz
- `[Ubuntu]` CARLA_0.9.14_RSS.tar.gz
- `[Windows]` CARLA_0.9.14.zip
- `[Windows]` AdditionalMaps_0.9.14.zip

Figura A.1: Archivos para CARLA Simulator *package* 0.9.14.

Después, descomprimir en un directorio de trabajo.

A.2.2. Instalación de la librería del cliente de CARLA Simulator

Existen varios métodos. Se recomienda altamente la instalación mediante los archivos *.whl*. Para ello, acudir al repositorio de CARLA Simulator 0.9.14 en *Pypi* y descargar el archivo necesario (ver Fig. A.2).

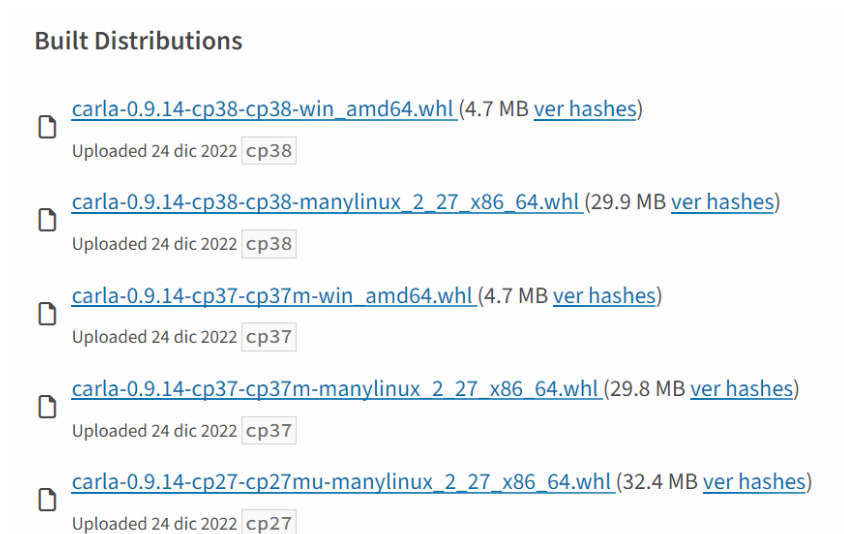

Figura A.2: Archivos *.whl* disponibles para CARLA Simulator 0.9.14.

En este caso, como la versión de *Python* que estamos utilizando es la 3.8 y el sistema operativo es Linux, el archivo que se debe descargar es *carla-0.9.14-cp38-cp38-manylinux_2_27_x86_64.whl*.

Para instalarlo, ir al directorio donde se encuentre el archivo (Listado A.1).

Listado A.1: Comando para instalar archivo *.whl* descargado

```
pip3 install carla-0.9.14-cp38-cp38-manylinux_2_27_x86_64.whl
```

A.2.3. Instalación de mapas adicionales

Para ello, descargar el archivo *AdditionalMaps_0.9.14.tar.gz* que aparece en la Figura A.1.

Mover el comprimido a la carpeta *Import* dentro del paquete de CARLA Simulator descargado en la Sección A.2.1. Ejecutar el archivo *ImportAssets.sh* en el directorio de CARLA Simulator para descomprimir e instalar el paquete adicional (Listado A.2).

Listado A.2: Comando para instalar mapas adicionales

```
cd path/to/carla/root
./ImportAssets.sh
```

A.3. ROS2 Foxy

La instalación de ROS2 Foxy es sencilla. A continuación se explica de forma detallada cómo proceder a su instalación. Si se desea consultar más información, acudir a la documentación oficial [36].

A.3.1. Añadir repositorio

El primer paso es añadir el repositorio apt de ROS2 al sistema. Para ello se ha de comrobar que el repositorio *Ubuntu Universe* está activado (Listado A.3).

Listado A.3: Comando para comprobar la disponibilidad de *Ubuntu Universe*

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Tras esto, añadir la llave GPG de ROS2 con apt (Listado A.4).

Listado A.4: Comando para añadir la llave GPG de ROS2

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro
   /master/ros.key -o /usr/share/keyrings/ros-archive-keyring.
   gpg
```

Finalmente, añadir el repositorio a la lista de fuentes (Listado A.5).

Listado A.5: Comando para añadir el repositorio de ROS2 al sistema

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/
   share/keyrings/ros-archive-keyring.gpg] http://packages.ros
   .org/ros2/ubuntu $(. /etc/os-release && echo
   $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/
   ros2.list > /dev/null
```

A.3.2. Instalación del paquete ROS2

Como preparativo primero se comprueba que la versión de apt está actualizada. En caso contrario, actualizarla (Listado A.6).

Listado A.6: Comandos para comprobar y actualizar versión de apt

```
sudo apt update
sudo apt upgrade
```

Tras esto, se recomienda descargar la versión ROS2 Foxy de escritorio. Es sencilla de instalar y contiene diversas herramientas, tutoriales y demos (List A.7).

Listado A.7: Comando para instalar la versión escritorio de ROS2 Foxy

```
sudo apt install ros-foxy-desktop python3-argcomplete
```

Además, es posible también la instalación de compiladores y otras herramientas para compilar paquetes de ROS2 (Listado A.8).

Listado A.8: Comando para instalar más herramientas para ROS2

```
sudo apt install ros-dev-tools
```

A.3.3. Descarga de paquetes externos para este TFG

Para la instalación de nuevos paquetes externos a ROS2 que no vengan ya instalados por defecto, se ha de proceder en el directorio que represente el entorno de trabajo de ROS2 objetivo.

Ros2_numpy

Es una instalación básica que permite adaptar el formato de los mensajes de los tópicos de ROS2 a un array NumPy. Para obtenerlo es tan sencillo como clonar el repositorio de *Github* [37] en el directorio de trabajo de ros2, compilarlo y aplicarlo (Listado A.9).

Listado A.9: Comando para instalar paquete *ros2_numpy*

```
# Ir al espacio de trabajo de ROS2
cd src/
git clone https://github.com/Box-Robotics/ros2_numpy -b humble
cd ..
colcon build --symlink-install --packages-select ros2_numpy
source install/setup.bash
```

Pcdet_ros2

Este paquete fue la principal razón por la que se decidió migrar el sistema a CARLA Simulator 0.9.15 (Cap. 5). Ha sido desarrollado para trabajar con la siguiente configuración:

- Ubuntu 22.04, ROS2 Humble
- *Python* 3.10
- *PyTorch* >= 2.0

En cuanto a la preparación del entorno, en primer lugar se debe instalar las versiones correspondientes de *PyTorch* y *Tensorflow* (Listado A.10).

Listado A.10: Comando para instalar *PyTorch* y *TensorFlow* para *pcdet_ros2*

```
python3 -m pip install torch torchvision torchaudio
python3 -m pip install tensorflow
```

Si se quiere instalar una versión específica de *PyTorch* (recomendado) acudir a Sección A.5.2. Tras esto, se procede con *OpenPCDet* (Sección A.5.5).

Finalmente, se ha de instalar dos librerías más (Listado A.11).

Listado A.11: Comando para instalar más liberías necesarias para *pcdet_ros2*

```
python3 -m pip install kornia open3d
python3 -m pip install pyquaternion
```

Una vez el entorno con *OpenPCDet* ya está listo, se puede proceder a la instalación del paquete para ROS2 Humble (Listado A.12). Es importante realizarlo en un directorio de trabajo de ROS2 previamente creado.

Listado A.12: Comandos para instalar *pcdet_ros2*

```
# Ir al espacio de trabajo de ROS2
cd src/
python3 -m pip install catkin_pkg
sudo apt install ros-humble-ament-cmake-nose -y
python3 -m pip install nose
python3 -m pip install transform3d
git clone https://github.com/pradhanshrijal/pcdet_ros2
cd ..
rosdep install -i --from-path src --rosdistro humble -y
colcon build --symlink-install --packages-select pcdet_ros2
source intsall/setup.bash
```

Además, se debe primero tener ya instalado el paquete *ros2_numpy* (Sección A.3.3), una dependencia importante de *pcdet_ros2*. Para acceder a más detalles, consultar su repositorio en *Github* [26].

Cv_bridge

Este paquete desarrolla una funcionalidad similar a la de *ros2_numpy*, sólo que en este caso adapta el formato de los mensajes en los tópicos de ROS2 al utilizado por la librería cv2 [38].

El formato cv2 resulta muy útil para poder modificar, dibujar, guardar o realizar todo tipo de tareas con imágenes 2D de forma sencilla.

De nuevo, para instalarlo sólo es necesario bajar el código de *Github* en el directorio de trabajo de ROS2, compilarlo y aplicarlo (Listado A.13).

Listado A.13: Comandos para instalar *cv_bridge*, contenido por *vision_opencv* [38].

```
# Ir al espacio de trabajo de ROS2
cd src/
git clone https://github.com/ros-perception/vision_opencv.git
   -b ros2
cd ..
colcon build --packages-select vision_opencv
source intsall/setup.bash
```

Vision_msgs_rviz_plugins para ROS2 Humble

Este paquete es una extensión para la herramienta *rviz2* de ROS2 Humble que permite visualizar mensajes de clase *vision_msgs* entre otros.

Con esta adición se podrá, por tanto, plasmar en *rviz2* las detecciones realizadas por las redes neuronales en las nubes de puntos de LiDAR 3D.

De nuevo, para instalarlo sólo es necesario bajar el código de *Github* [39] en el directorio de trabajo de ROS2, compilarlo y aplicarlo (Listado A.14).

Listado A.14: Comandos para instalar *vision_msgs_rviz_plugins*.

```
# Ir al espacio de trabajo de ROS2
cd src/
git clone https://github.com/NovoG93/vision_msgs_rviz_plugins
   -b humble
cd ..
rosdep install --from src --ignore-src -r -y
colcon build --packages-select vision_opencv
source intsall/setup.bash
```

A.4. *Carla-ros-bridge*

Primero, se debe crear un espacio de trabajo del puente y clonar el repositorio de ROS bridge (Listado A.15).

Listado A.15: Comando para clonar el repositorio carla-ros-bridge

```
mkdir -p ~/carla-ros-bridge && cd ~/carla-ros-bridge
git clone --recurse-submodules https://github.com/carla-
   simulator/ros-bridge.git src/ros-bridge
```

Después, construir el entorno de ROS2 (Listado A.16).

Listado A.16: Comando para construir el entorno de ROS2

```
source /opt/ros/foxy/setup.bash
```

Los siguientes pasos es importante hacerlos dentro del espacio de trabajo de ROS2. Por un lado instalar las dependencias (Listado A.17).

Listado A.17: Comando para instalar dependencias de carla-ros-bridge

```
rosdep update
rosdep install --from-paths src --ignore-src -r
```

Por otro lado, en el directorio donde veas la carpeta *src*, compilar el espacio de trabajo de ROS2 (Listado A.18).

Listado A.18: Comando para compilar carla-ros-bridge

```
colcon build
```

Para utilizarlo, se debe configurar el entorno de ROS2 para *carla-ros-bridge*. Para ello, el comando de Listado A.19 see debe ejecutar dentro del directorio de trabajo de este paquete.

Listado A.19: Comando para configurar carla-ros-bridge

```
source install/setup.bash
```

A.5. Librerías y código base para uso de redes neuronales

Se recomienda la instalación de estas librerías en entornos virtuales para poder explorar múltiples códigos base, herramientas e interfaces sin problemas con incompatibilidad de versiones.

A.5.1. CUDA 12.1

Para ello navegar a la página oficial [40], seleccionar las condiciones particulares del usuario y reproducir el código resultante en el terminal (ver Fig. A.3).

CUDA Toolkit 12.3 Update 1 Downloads

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the CUDA EULA.

Operating System: Linux, Windows
Architecture: x86_64, ppc64le, arm64-sbsa
Distribution: CentOS, Debian, Fedora, KylinOS, OpenSUSE, RHEL, Rocky, SLES, Ubuntu, WSL-Ubuntu
Version: 20.04, 22.04
Installer Type: deb (local), deb (network), runfile (local)

Figura A.3: Ejemplo de selección de características para CUDA 12.1.

En el caso de este ejemplo, el código que finalmente se ha de ejecutar es el de Listado A.20.

Listado A.20: Comando para instalar CUDA 12.1 según condiciones del ejemplo

```
wget https://developer.download.nvidia.com/compute/cuda
   /12.3.1/local_installers/cuda_12.3.1_545.23.08_linux.run
sudo sh cuda_12.3.1_545.23.08_linux.run
```

Si se quiere tener diferentes versiones disponibles de CUDA y poder alternar entre una y otra, se pueden seguir los pasos del blog explicativo correspondiente referenciado [41].

A.5.2. *PyTorch*

De forma similar al paso anterior, navegar a la página oficial de *PyTorch* [42] y seleccionar las características del usuario (ver Fig. A.4).

PyTorch Build	Stable (2.1.2)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.8	CUDA 12.1	~~ROCm 5.6~~	CPU

Figura A.4: Ejemplo de selección de *PyTorch*.

En este caso, el código resultante a ejecutar es el de Listado A.21.

Listado A.21: Comando para instalar *Pytorch* con CUDA 12.1 en el sistema

```
pip3 install torch torchvision torchaudio --index-url https://
    download.pytorch.org/whl/cu121
```

A.5.3. *Ultralytics*

Para esta librería, basta con ejecutar este único comando que viene documentado en la página oficial [43]. En este caso se utiliza con instalación *pip* (Listado A.22).

Listado A.22: Comando para instalar *Ultralytics*

```
pip install ultralytics
```

A.5.4. *Transformers*

Para *transformers*, basta con escribir el comando recomendado (Listado A.23) en la página oficial de *The Hugging Face* [44].

Listado A.23: Comando para instalar *transformers*

```
pip install transformers
```

A.5.5. *OpenPCDet*

En primer lugar, se debe clonar el repositorio de *OpenPCDet* en un directorio de trabajo (Listado A.24).

Listado A.24: Comando para clonar el repositorio de OpenPCDet

```
git clone https://github.com/open-mmlab/OpenPCDet
```

Después, instalar la librería *spconv* con su respectiva versión y los requerimientos (Listado A.25).

Listado A.25: Comando para instalar dependencias de OpenPCDet

```
python3 -m pip install spconv-cu117
cd OpenPCDet
python3 -m pip install -r requirements.txt
```

Finalmente, ejecutar el código *Python setup.py* junto con todas las librerías que notifique como no instaladas durante el proceso (Listado A.26).

Listado A.26: Comando para ejecutar *setup.py* para usar OpenPCDet

```
python3 setup.py develop
```

Para consultar más detalles, acudir al repositorio oficial [22].

Apéndice B

Manual de Usuario

Contenido

B.1 Introducción y repositorio de *GitHub* 93
B.2 CARLA Simulator 94
B.2.1 Crear modelo digital dentro del simulador 94
B.2.2 Modificar modelo digital 95
B.2.3 Algoritmo para crear y utilizar rutas programadas 96
B.2.4 Algoritmo para utilizar el control manual 97
B.2.5 Generación de agentes del tráfico externos 98
B.2.6 Otros comandos básicos importantes 99
B.3 ROS2 101
B.3.1 Modificar, compilar y aplicar cambios en un paquete . . . 101
B.3.2 Puesta en marcha del LiDAR fusión normal 102
B.3.3 Puesta en marcha del LiDAR fusión semántico 102
B.3.4 ROS2 *Bag* y *rviz2* 102
B.3.5 Uso de los nodos de procesamiento de datos 103
B.4 Ejemplo de algoritmo para realización de simulaciones . . . 104
B.5 Ejemplo de algoritmo para procesar datos recopilados . . . 107

B.1. Introducción y repositorio de *GitHub*

Este apéndice es una guía para el usuario que quiera utilizar la interfaz creada para este proyecto. Aborda tanto algoritmos generales y utilidades como explica-

ciones de cómo realizar tareas específicas del TFG.

Todo el código desarrollado para este TFG se encuentra en el repositorio de *GitHub* correspondiente [27].

B.2. CARLA Simulator

En esta sección se muestran algunos tutoriales para aprender a utilizar la interfaz de CARLA Simulator con objetivo de hacer tareas similares a las realizadas en este TFG. Por defecto, todo está explicado para la versión CARLA Simulator 0.9.14, que por lo general suele ser idéntico para la nueva versión. Si en algún momento se explica algo para CARLA Simulator 0.9.15, se notificará en el documento.

Para iniciar el simulador, lo que equivale a arrancar el servidor, basta con abir un terminal y lanzar el ejecutable contenido en la carpeta de CARLA Simulator (Listado B.1).

Listado B.1: Comandos para iniciar CARLA Simulator

```
cd a/directorio/de/CARLA
# Graficos originales
./CarlaUE4.sh
# Graficos de baja calidad para simulaciones de alto
   rendimiento
./CarlaUE4.sh -quality-level=Low
```

Es importante destacar, que los clientes que se quieran ejecutar, pueden ser utilizados desde cualquier otra pestaña del terminal, o incluso en otra ventana.

B.2.1. Crear modelo digital dentro del simulador

En primer lugar, se ha de abrir un terminal e iniciar el puente de *carla-ros-bridge* en forma pasiva (Listado B.2). Esto último indica que el puente no va a marcar el tiempo de simulación, sino que será el servidor u otro cliente.

Listado B.2: Comando para iniciar carla-ros-bridge

```
ros2 launch carla_ros_bridge carla_ros_bridge.launch.py
   passive:=True
```

Tras esto, abrir otra pestaña en el terminal y ejecutar el nodo de ROS2 del paquete *carla-ros-bridge* encargado de crear objetos en el simulador a partir de archivos *.json*. El archivo seleccionado será aquel con el modelo digital deseado.

Listado B.3: Comando para crear algún modelo digital en CARLA Simulator

```
ros2 launch carla_spawn_objects carla_spawn_objects.launch.py
   objects_definition_file:=a/directorio/de/archivo.json
```

B.2.2. Modificar modelo digital

Existen 4 modelos digitales creados. Uno con el vehículo sin sensores, otro sólo con las cámaras RGB normales y semánticas, un tercero análogo con los LiDARs y el último siendo el modelo digital completo.

En caso de que se quiera crear un modelo personalizado, es necesario entender de qué están compuestos estos archivos *.json*. En primer lugar, se crea el modelo del actor principal, en este caso el vehículo sensorizado, pero puede ser otro vehículo, un peatón, una bicicleta, etc (ver Fig. B.1). El agente posee distintas características, entre ellas el lugar de creación, que deberá ser cuidadosamente ajustado según la prueba a realizar.


```
{
    "objects":
    [
        {
            "type": "sensor.pseudo.traffic_lights",
            "id": "traffic_lights"
        },
        {
            "type": "sensor.pseudo.objects",
            "id": "objects"
        },
        {
            "type": "sensor.pseudo.actor_list",
            "id": "actor_list"
        },
        {
            "type": "sensor.pseudo.markers",
            "id": "markers"
        },
        {
            "type": "sensor.pseudo.opendrive_map",
            "id": "map"
        },
        {
            "type": "vehicle.toyota.prius",
            "id": "ego_vehicle",
            "spawn_point":{"x": -48.84, "y": 43.35, "z": 1.0, "roll": 0.0, "pitch": 0.0, "yaw": -90.00} ,
```


Figura B.1: Ejemplo de actores principales en un archivo *.json*.

Tras esto, se le adhieren distintos sensores con sus respectivas características (ver Fig. B.2). Si se desea realizar pruebas con una distribución concreta de sensores, basta con copiar y pegar aquellos de interés que ya aparezcan en el modelo completo. Otro caso típico es el de querer probar un mapa o una localización distinta, donde es importante ajustar las coordenadas de creación del actor principal.

```
{
    "type": "vehicle.toyota.prius",
    "id": "ego_vehicle",
    "spawn_point":{"x": -48.84, "y": 43.35, "z": 1.0, "roll": 0.0, "pitch": 0.0, "yaw": -90.00} ,
    "sensors":
    [

        {
            "type": "sensor.camera.rgb",
            "id": "Left_RGB_1",
            "spawn_point":{"x": 0.354, "y": 0.425, "z": 1.615, "roll": 0.0, "pitch": 0.0, "yaw": 65},
            "image_size_x": 1280, "image_size_y": 720, "fov": 110, "fstop":1.8,
            "sensor_tick": 0.03333, "shutter_speed":0.03333
        },
        {
            "type": "sensor.camera.semantic_segmentation",
            "id": "Left_Semantic_1",
            "spawn_point":{"x": 0.354, "y": 0.425, "z": 1.615, "roll": 0.0, "pitch": 0.0, "yaw": 65},
            "image_size_x": 1280, "image_size_y": 720, "fov": 110,
            "sensor_tick": 0.03333
        },
        {
            "type": "sensor.lidar.ray_cast",
            "id": "LIDAR_0",
            "spawn_point": {"x": 0.2, "y": 0.0, "z": 1.8635, "roll": 0.0, "pitch": 0.0, "yaw": 0.0},
            "range": 100,
            "channels": 4,
            "points_per_second": 72000,
            "upper_fov": 15.0,
            "lower_fov": 9.0,
            "rotation_frequency": 20,
            "horizontal_fov":360
        },
        {
            "type": "sensor.lidar.ray_cast_semantic",
            "id": "LIDAR_Sem_0",
            "spawn_point": {"x": 0.2, "y": 0.0, "z": 1.8635, "roll": 0.0, "pitch": 0.0, "yaw": 0.0},
            "range": 100,
            "channels": 4,
            "points_per_second": 72000,
            "upper_fov": 15.0,
            "lower_fov": 9.0,
            "rotation_frequency": 20,
            "horizontal_fov":360
        },
```

Figura B.2: Ejemplo de sensores creados para el vehículo sensorizado en un archivo *.json*.

Si se quiere modificar dicha localización a un lugar atractivo en el mapa, basta con ejecutar un cliente útil desarrollado para CARLA Simulator que informa de la posición del espectador en el mapa (Listado B.4).

Listado B.4: Comando para consultar la posición del espectador

```
cd /home/premovesim/tfg_alberto/code/utils
python3 spectator_location.py
```

B.2.3. Algoritmo para crear y utilizar rutas programadas

En primer lugar se debe iniciar el simulador (Listado B.1), seleccionar el mapa a utilizar (Listado B.9), iniciar el puente con ROS2 y crear el modelo digital

(Sección B.2.1).

Finalmente, se ha de abrir una nueva pestaña en el terminal y ejecutar el cliente de CARLA Simulator correspondiente a la ruta que se quiera realizar (Listado B.5). Existen dos rutas programadas para las ciudades *Town03_Opt* y *Town10HD_Opt*. Es recomendable observar el punto de inicio de cada una de las rutas y comprobar que sea coherente con la coordenada de creación del modelo digital del vehículo sensorizado.

Listado B.5: Comandos para iniciar ruta programada

```
cd /home/premovesim/tfg_alberto/code/rutas_pkg
# Para Tonw10HD_Opt
python3 t10_ruta1.py
# Para Town03_Opt
python3 t3_ruta1.py
```

B.2.4. Algoritmo para utilizar el control manual

Al igual que para las rutas programadas, inicialmente se han de realizar los siguientes pasos. En primer lugar se debe iniciar el simulador (Listado B.1), seleccionar el mapa a utilizar (Listado B.9), iniciar el puente con ROS2 y crear el modelo digital (Sección B.2.1).

Tras esto, el siguiente paso es ejecutar el cliente escrito en *Python* para el control manual del vehículo (Listado B.6). Además, tras su inicio, se deberá introducir la cantidad de vehículos NPCs que se quieren en la simulación (ver Fig. B.3).

Listado B.6: Comandos para iniciar control manual

```
cd /home/premovesim/tfg_alberto/code/vehicle_control
python3 manual_toyota_steering_and_traffic.py
```

Figura B.3: Ejemplo de especificación de NPCs a generar.

B.2.5. Generación de agentes del tráfico externos

CARLA Simulator 0.9.14

En esta versión se ha de realizar todos los pasos para utilizar el control manual (Sección B.2.4). Para añadir vehículos NPCs, simplemente se debe incluir el número deseado después de ejecutar *manual_toyota_steering_and_traffic.py* (ver Fig. B.3).

En segundo lugar, para añadir peatones a la simulación, basta con ejecutar el código específico, que creará 20 peatones en localizaciones concretas (Listado B.7). Este código sólo sirve para las ciudades *Town10HD_Opt*, *Tonw06_Opt* y *Town03_Opt*.

Listado B.7: Comandos para crear peatones en CARLA Simulator 0.9.14

```
cd /home/premovesim/tfg_alberto/code
python3 generate_pedestrians.py
```

CARLA Simulator 0.9.15

Para la nueva versión, el algoritmo es algo distinto. Primero se deben seguir todos los pasos para lanzar el control manual (Sección B.2.4). En este caso, se debe especificar como cero la cantidad de vehículos NPC a crear una vez se ejecute *manual_toyota_steering_and_traffic.py* (ver Fig. B.4).

En su lugar, para la creación de tráfico tanto de vehículos como de peatones, se utilizará el cliente *generate_traffic.py*. Para concretar la cantidad de vehículos que se quiere crear, se debe utilizar el argumento *-n*. En el caso de los peatones, el algurmento es *-w* (Listado B.8).

Listado B.8: Comandos para crear tráfico en CARLA Simulator 0.9.15

```
cd /home/premovesim/tfg_alberto/code
python3 generate_traffic.py -n 30 -w 20
```

Figura B.4: Ejemplo de especificar una cantidad nula de NPCs a generar.

B.2.6. Otros comandos básicos importantes

Cambiar de mapa

Para ello se utilizará el argumento *–map* en el código *config.py* dentro de la *PythonAPI* de CARLA Simulator (Listado B.9).

Listado B.9: Ejemplo para cambiar de mapa

```
cd a/directorio/de/CARLA/PythonAPI/util
python3 config.py --map Town03_Opt
```

Modificar condiciones climáticas

Para ello se utilziarán los argumentos de interés en el cliente *environment.py* dentro de la *PythonAPI* de CARLA Simulator (Listado B.10).

Listado B.10: Ejemplo para ajustar la altura del sol en la simulación

```
cd a/directorio/de/CARLA/PythonAPI/util
python3 environment.py -alt 10.00
```

Salir del modo síncrono

Si el cliente que marca el tiempo de simulación muere, el servidor se quedará congelado y no responderá. En ese caso es conveniente ejecutar un cliente que devuelve a CARLA Simulator a modo asíncrono, haciéndolo funcionar de nuevo (Listado B.11).

Listado B.11: Comandos para devolver al simulador al modo asíncrono

```
cd /home/premovesim/tfg_alberto/code/utils
python3 out_syncr.py
```

Mostrar *spawn points*

Si se quiere plasmar en el espectador la localización de cada uno de los *spawn points* del mapa correspondiente, basta con iniciar un cliente encargado de ello.

Listado B.12: Comandos para mostrar los *spawn points* del mapa

```
cd /home/premovesim/tfg_alberto/code/utils
python3 spawn_points.py
```

Es posible modificar el código escrito en *Python* para ajustar el tiempo que aparecen en pantalla dichos puntos de creación.

B.3. ROS2

En esta sección se describirá cómo utilizar la interfaz de ROS2 desarrollada en este TFG. El manual de usuario es de uso idéntico tanto para ROS2 Foxy como para su distribución ROS2 Humble.

Para poder utilizar los comandos de ROS2, se debe inicializar el entorno de la aplicación en cada nuevo terminal que se abra. En el equipo de trabajo en el laboratorio, está incluido el comando al *script* de inicialización del *shell*, lo cual evita tener que realizar este paso cuando se abran nuevas pestañas.

Listado B.13: Comando para inicializar entorno de ROS2

```
# Para ROS2 Foxy
source /opt/ros/foxy/setup.bash
# Para ROS2 Humble
source /opt/ros/humble/setup.bash
```

B.3.1. Modificar, compilar y aplicar cambios en un paquete

A la hora de modificar un paquete de ROS2, los típicos casos suelen ser cambiar el código de un nodo o de un archivo de lanzamiento, también llamado *launch file*. Una vez dicho cambio haya sido efectuado, se debe compilar en primer lugar seleccionando el paquete modificado. Para ello es muy importante hacer la compilación en el espacio de trabajo de ROS2 en el que se encuentre el material modificado. El compilador utilizado en este TFG es *colcon* (Listado B.14).

Listado B.14: Comandos para compilar paquetes en ROS2

```
cd a/espacio/de/trabajo/de/ROS2
colcon build --packages-select paquete_modificado
```

Tras esto, se deben aplicar los cambios para que a la hora de ejecutar los nodos o lanzadores alterados, queden plasamdos las últimas modificaciones realizadas (Listado B.15).

Listado B.15: Comandos para aplicar cambios en ROS2

```
cd a/espacio/de/trabajo/de/ROS2
source install/setup.bash
```

B.3.2. Puesta en marcha del LiDAR fusión normal

Para ello, una vez se hayan seguido todos los pasos para crear el modelo digital del vehículo sensorizado (Sección B.2.1), se debe ejecutar el nodo de ROS2 que pone en marcha el LiDAR fusión normal.

Listado B.16: Comando para poner en marcha el LiDAR normal

```
ros2 run regular_lidar_pkg fusion_lidar_alberto
```

B.3.3. Puesta en marcha del LiDAR fusión semántico

El procedimiento es similar que para el homólogo no semántico. Para ello, una vez se hayan seguido todos los pasos para crear el modelo digital del vehículo sensorizado (Sección B.2.1), se debe ejecutar el nodo de ROS2 que pone en marcha el LiDAR fusión semántico.

Listado B.17: Comando para poner en marcha el LiDAR semántico

```
ros2 run semantic_lidar_pkg fusion_semantic_lidar
```

B.3.4. ROS2 *Bag* y *rviz2*

ROS2 *Bag*

Para grabar flujos de datos por los tópicos de ROS2 se utiliza el comando *record*. En él se escribe el nombre de los tópicos de interés además del argumento *-o*, que permite especificar el nombre del archivo resultante (Listado B.18).

Listado B.18: Ejemplo de uso de ROS2 Bag para grabar tópicos

```
ros2 bag record -o nombre_de_grabacion /topico1 /topico2 /
   topicoN
```

Por defecto, el fichero de la grabación será guardado en el directorio desde el que fue ejecutado el comando.

Para reproducir las grabaciones se debe utilizar el comando contrario *play*. Además, es necesario escribir el nombre del archivo objetivo desde el directorio en el que se encuentre (Listado B.19).

Listado B.19: Comandos para reproducir grabaciones de ROS2

```
cd a/directorio/de/grabaciones/bag
ros2 bag play nombre_de_grabacion
```

Rviz2

Para iniciar *rviz2*, basta con ejecutar el comando de la Listado B.20. Desde esta herramienta, se podrán visualizar los datos de todos los sensores, tanto los normales como sus homólogos semánticos (ver Fig. B.5).

Listado B.20: Comando para iniciar rviz2

```
ros2 run rviz2 rviz2
```

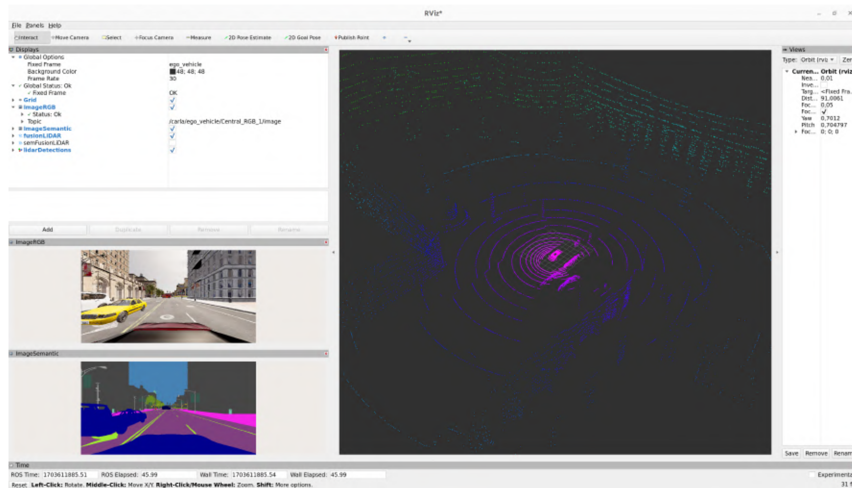

Figura B.5: Interfaz de *rviz2* con todos los sensores activos.

B.3.5. Uso de los nodos de procesamiento de datos

Para utilizarlos se necesita haber creado primero el modelo digital en el simulador (Sección B.2.1). Esto es, para que los sensores se encuentren generando datos que fluyan por los tópicos de ROS2. La otra opción es simplemente reproducir una grabación previamente capturada (Sección B.3.4).

Para seleccionar la red DETR en *rgb_2_detr* del paquete *ros2_object_detection_pkg*, se ha de modificar la variable *CHECKPOINT* en el código del nodo. Tras esto se puede utilizar ejecutando directamente el nodo (Listado B.21).

Listado B.21: Comando para utilizar redes DETR

```
ros2 run ros2_object_detection_pkg rgb_2_detr
```

El nodo *rgb_2_cnn* del paquete *ros2_object_detection_pkg* utiliza la red *YOLOv8*, que es la única que se ha puesto a prueba en este TFG de la librería *ultralytics*. Se puede utilizar ejecutando directamnete el nodo (Listado B.22).

Listado B.22: Comando para utilizar red YOLOv8

```
ros2 run ros2_object_detection_pkg rgb_2_cnn
```

En el caso de las redes de detección 3D del paquete *pcdet_ros2* se han de hacer dos pasos. El primero es incluir los pesos sinápticos en la carpeta *checkpoints* en el paquete de ROS2. El segundo paso es modificar la variable *config_file* en el lanzador *pcdet.launch.py* para que utilice dicho modelo preentrenado. Tras esto se puede utilizar ejecutando el lanzador (Listado B.23).

Listado B.23: Comando para utilizar redes de detección 3D

```
ros2 launch pcdet_ros2 pcdet.launch.py
```

B.4. Ejemplo de algoritmo para realización de simulaciones

Este algoritmo describe todo el proceso para realizar simulaciones personalizadas conduciendo el vehículo de forma manual. Se recomienda reservar una ventana del terminal sólo para esta secuencia. El sistema operativo utilizado es Ubuntu 22.04, que es donde se han hecho todas las pruebas a partir del último mes de 2023.

Preparación del terminal y el entorno virtual

Abir un terminal y dividirlo en 8 pestañas. En cada una de ellas se debe iniciar el entorno virtual de conda *humble_env* que contiene a ROS2 Humble (Listado B.24). En este entorno virtual vienen además todas las librerías instaladas para el uso de los nodos de redes neuronales.

Listado B.24: Comando para activar entorno de conda para ROS2 Humble

```
mamba activate humble_env
```

Iniciación de CARLA Simulator y ajustes del servidor

En una pestaña, iniciar el simulador como en la Listado B.1 con gráficos bajos. Tras esto, en otra pestaña, seleccionar el mapa a utilizar así como las condiciones climatológicas (Listados B.9 y B.10). Reservar esta pestaña sólo para procesos de ajustes del servidor

Puente de *carla-ros-bridge* y modelo digital

En otras dos pestañas, iniciar el puente de *carla-ros-bridge* y crear el modelo digital completo en el simulador tal y como se describe en la Sección B.2.1. Por ejemplo, si se está utilizando *Town10HD_Opt* el archivo estará en el directorio de la ciudad 10 tal y como se muestra en Listado B.25.

Listado B.25: Comando para crear toyota completo en *Town10HD_Opt*

```
ros2 launch carla_spawn_objects carla_spawn_objects.launch.py
   objects_definition_file:=/home/premovesim/tfg_alberto/code/
   objects_files/town10HD/full_toyota.json
```

Otro ejemplo para *Town15* es el de la Listado B.26.

Listado B.26: Comando para crear toyota completo en *Town15*

```
ros2 launch carla_spawn_objects carla_spawn_objects.launch.py
   objects_definition_file:=/home/premovesim/tfg_alberto/code/
   objects_files/town15/full_toyota.json
```

Comando del control manual

En una cuarta ventanilla, iniciar el control manual como en la Sección B.2.4. Como resultado se generará una pequeña pantalla desde la que se podrá manejar el vehículo (ver Fig. B.6).

Figura B.6: Ventana con control manual del vehículo sensorizado.

Generación de tráfico externo

Seguir los pasos tal y como se indican en la Sección B.2.5 según la versión del simulador que se utilice. En este algoritmo se ha optado por la opción para

CARLA Simulator 0.9.15, por lo que se ha indicado una cantidad nula de NPCs a generar desde el cliente de control manual del vehículo.

Por otro lado, se genera el resto de agentes del tráfico con el cliente *generate_traffic.py* en una nueva ventana.

Puesta en marcha de los LiDARs

A continuación, en dos nuevas ventanas, poner en marcha los nodos encargados de fusionar los LiDARs normales y semánticos (Secciones B.3.2 y B.3.3).

Grabación con ROS2 *Bag*

Finalmente, en la última ventana de las 8 abiertas, iniciar la grabación de todos los tópicos con ROS2 *Bag* y comenzar la conducción y con ello la simulación (Listado B.27 y ver Fig. B.7).

Listado B.27: Comando para grabar todos los sensores con ROS2 Bag

```
ros2 bag record -o nombre_de_simulacion /carla/ego_vehicle/
    Central_RGB_1/image /carla/ego_vehicle/Left_RGB_1/image /
    carla/ego_vehicle/Right_RGB_1/image /carla/ego_vehicle/
    LIDAR /carla/ego_vehicle/LIDAR_SEMANTIC /carla/ego_vehicle/
    Central_Semantic_1/image /carla/ego_vehicle/Left_Semantic_1
    /image /carla/ego_vehicle/Right_Semantic_1/image
```

Figura B.7: Vista previa del terminal con todos los comandos.

Para más detalles sobre ROS2 *Bag record*, consultar la Sección B.3.4.

B.5. Ejemplo de algoritmo para procesar datos recopilados

En este apartado se describirá cómo desplegar la interfaz completa para reproducir, procesar y visualizar los datos grabados en las simulaciones. Para ello se recomienda reservar una ventana entera de un nuevo terminal divido en 4 pestañas. En cada una de ellas deberá ser activado el entorno virtual *humble_env* (Listado B.26).

Preparación de *rviz2*

En la primera pestaña, ejecutar el comando de la Listado B.20. Tras esto, abrir la plantilla de *rviz2* [insertar nombre de la plantilla] con la visualización de todos los tópicos de interés. Si se quieren abrir otros tópicos, modificar la plantilla a placer y guardarla como una nueva.

Inicialización de los nodos de las redes neuronales

Iniciar los nodos de las redes neuronales para la detección de obstáculos 2D y 3D (Sección B.3.5). Es posible tener una red de cada tipo activa simultáneamente.

Reproducción de datos con ROS2 *Bag*

Una vez todos los nodos de procesamiento están activos y los subscriptores de cada uno de ellos creados, es el momento de, en otra ventana, iniciar la reproducción de datos con ROS2 *Bag* (Listado B.19).

Visualización de resultados

Para visualizar las detecciones realizadas en la nube de puntos generada por el LiDAR fusión normal, se ha de mirar las cajas generadas desde *rviz2* (ver Fig. B.8). Por otro lado, las detecciones realizadas sobre las imágenes 2D se guardan en su carpeta correspondiente en el directorio */home/premovesim/tfg_alberto/carla_records* como nuevas ilustraciones.

Los resultados son de cajas delimitadoras o *bounding boxes* 2D o 3D correspondientemente como se puede consultar en la Sección 5.3.

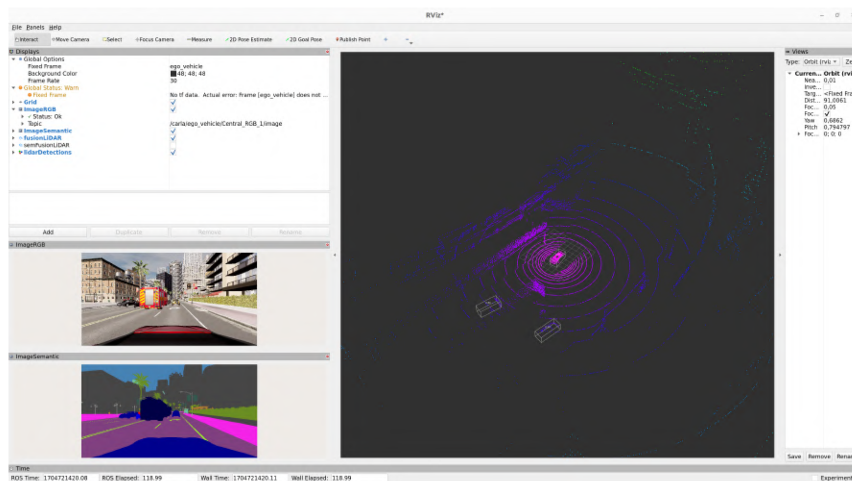

Figura B.8: Resultados de detecciones 3D en *rviz2*.

Bibliografía

[1] J. Morales and J. L. Martínez. Universidad de Málaga: Predicción del Movimiento de los Participantes del Tráfico para la Integración Segura del Vehículo Autónomo en Áreas Urbanas. https://www.uma.es/robotics-and-mechatronics/info/138109/premove/, En curso.

[2] D. Steven Gamba. *Incorporación de sensores realistas en la simulación en CARLA de la navegación de coches autónomos*. Trabajo Fin de Grado en Ingeniería Electrónica, Robótica y Mecatrónica. Universidad de Málaga., 2023.

[3] SAE. Levels of Driving Automation™ Refined for Clarity and International Audience. https://www.sae.org/blog/sae-j3016-update#:~:text=With%20a%20taxonomy%20for%20SAE%E2%80%99s%20six%20levels%20of,of%20motor%20vehicles%20and%20their%20operation%20on%20roadways., 2021.

[4] GIGABYTE TM. GeForce RTX™ 4060 Ti GAMING OC 16G. Image Gallery. https://www.gigabyte.com/es/Graphics-Card/GV-N406TGAMING-OC-16GD/sp#sp, 2024.

[5] UnrealEngine. CARLA democratizes autonomous vehicle R&D with free open-source simulator, 2019.

[6] CARLA Simulator Team. Carla 0.9.14 release. new semantic classes. https://carla.org/2022/12/23/release-0.9.14/, 2022.

[7] CARLA Simulator Team. Carla 0.9.14 release. town12. https://carla.org/2022/12/23/release-0.9.14/, 2022.

[8] CARLA Simulator. Sensors reference. Semantic LiDAR sensor. https://carla.readthedocs.io/en/latest/ref_sensors/#semantic-lidar-sensor, 2023.

[9] CARLA Simulator Team. Maps and Navigation. Non layered maps. Town 6. https://carla.readthedocs.io/en/latest/map_town06/, 2023.

[10] H. Andrade. Modelo para detectar el uso correcto de mascarillas en tiempo real utilizando redes neuronales convolucionales. https://www.researchgate.net/figure/Figura-1-Descripcion-del-funcionamiento-de-una-red-neuronal-convolucional-CNN-10_fig1_348825166, 2021.

[11] N. Carion, F. Massa, and G. Synnaeve et al. End-to-End Object Detection with Transformers. https://doi.org/10.48550/arXiv.2005.12872, 2020. arXiv:2005.12872 [cs.CV].

[12] S. Shi, C. Guo, L. Jiang, and Z Wang et al. PV-RCNN: Point-Voxel Feature Set Abstraction for 3D Object Detection. https://doi.org/10.48550/arXiv.1912.13192, 2019. arXiv:1912.13192 [cs.CV].

[13] S. Shi, Z. Wang, J. Shi, X. Wanga, and H. Li. From Points to Parts: 3D Object Detection from Point Cloud with Part-aware and Part-aggregation Network. https://doi.org/10.48550/arXiv.1907.03670, 2019. arXiv:1907.03670 [cs.CV].

[14] CARLA Simulator Team. Maps and Navigation. Non layered maps. Town 10. https://carla.readthedocs.io/en/latest/map_town10/, 2023.

[15] K. O'Shea and R. Nash. An Introduction to Convolutional Neural Networks. https://doi.org/10.48550/arXiv.1511.08458, 2015. arXiv:1511.08458 [cs.NE].

[16] F. Codevilla et al. A. Dosovitskiy, G. Ros. CARLA: An Open Urban Driving Simulator. https://doi.org/10.48550/arXiv.1711.03938, 2017. arXiv:1711.03938 [cs.LG].

[17] S. Kato y T. Azumi Y. Maruyama. Exploring the performance of ROS2. In *Proceedings of the 13th ACM SIGBED International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

[18] CARLA Simulator. ROS bridge installation for ROS 2. https://carla.readthedocs.io/projects/ros-bridge/en/latest/ros_installation_ros2/, 2022.

[19] A. Moore. rviz. https://github.com/ros2/rviz, 2024. Repositorio en línea.

[20] M. Orlov. rosbag2. https://github.com/ros2/rosbag2, 2024. Repositorio en línea.

[21] A. Paszke, S. Gross, and F. Massa et al. Pytorch: An imperative style, high-performance deep learning library. https://doi.org/10.48550/arXiv.1912.01703, 2019. arXiv:1912.01703 [cs.LG].

[22] OpenPCDet Development Team. OpenPCDet Toolbox for LiDAR-based 3D Object Detection. https://github.com/open-mmlab/OpenPCDet, 2020.

[23] CARLA Simulator Team. CARLA Documentation. https://carla.readthedocs.io/en/latest/, 2023.

[24] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[25] T. Gamage. ROS2 bridge for CARLA simulator v0.9.15, 2023.

[26] S. Pradhan. Ros 2 wrapper for openpcdet. https://github.com/pradhanshrijal/pcdet_ros2, 2023. Repositorio en línea.

[27] A. García. Detection of traffic agents in urban environments in simulation usig CARLA and ROS2. https://github.com/aggRobotics/Detection-of-traffic-agents-usig-CARLA-and-ROS2.git, 2024.

[28] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. https://doi.org/10.48550/arXiv.1506.02640, 2016. arXiv:1506.02640 [cs.CV].

[29] G. Jocher, A. Chaurasia, and J. Qiu. Ultralytics YOLO. https://github.com/ultralytics/ultralytics, 2023.

[30] Julien Chaumond y Thomas Wolf Clément Delangue. Hugging Face, Inc. huggingface.co, 2024.

[31] C. R. Harris, K. Jarrod, S. J. van der Walt, R. Gommers, and P. Virtanen et al. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[32] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[33] E. Vergara Gómez. *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*. Trabajo Fin de Máster en Ingeniería Mecatrónica. Universidad de Málaga., 2020.

[34] CARLA Simulation Team. Quick start package instalation. https://carla.readthedocs.io/en/latest/start_quickstart/, 2023.

[35] CARLA Simulator Team. CARLA Docs Download. https://github.com/carla-simulator/carla/blob/master/Docs/download.md, 2023.

[36] ROS2 Team. ROS2 Documentation: Foxy. https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html, 2024.

[37] T. Panzarella. ros2_numpy. https://github.com/Box-Robotics/ros2_numpy, 2023. Repositorio en línea.

[38] K. Brameld. vision_opencv. https://github.com/ros-perception/vision_opencv, 2023. Repositorio en línea.

[39] G. No. Rviz2 plugins for visualization of vision_msgs. https://github.com/NovoG93/vision_msgs_rviz_plugins, 2023. Repositorio en línea.

[40] NVIDIA Developer. CUDA Toolkit 12.1 Downloads. https://developer.nvidia.com/cuda-12-1-0-download-archive, 2023.

[41] C. Zatout. Managing Multiple CUDA Versions on a Single Machine: A Comprehensive Guide. https://towardsdatascience.com/managing-multiple-cuda-versions-on-a-single-machine-a-comprehensive-guide-97db1b22acdc, 2023. Artículo en Towards Data Science.

[42] Facebook's AI Research lab (FAIR). PyTorch: Install PyTorch. https://pytorch.org/, 2023.

[43] Ultralytics Team. Ultralytics YOLOv8 Docs. Quickstart: Install Ultralytics. https://docs.ultralytics.com/quickstart/, 2023.

[44] The Huggin Face. The Hugging Face Transformers: Installation. https://huggingface.co/docs/transformers/installation, 2020.