Escuela de Ingenierías Industriales

Departamento de Ingeniería de Sistemas y Automática

Área de Ingeniería de Sistemas y Automática

Trabajo de Fin de Máster

Integración de técnicas de detección de participantes del tráfico en entornos urbanos mediante un sensor LiDAR 3D y cámaras RGB con CARLA y ROS2

Máster en

Ingeniería Mecatrónica

Autor: Jorge Montenegro Navarro

Tutor: Jesús Morales Rodríguez

Málaga, Noviembre de 2024

Escuela de Ingenierías Industriales. Universidad de Málaga

Integración de técnicas de detección de participantes del tráfico en entornos urbanos mediante un sensor LiDAR 3D y cámaras RGB con CARLA y ROS2

Autor: Jorge Montenegro Navarro.

Tutor: Jesús Morales Rodríguez.

Departamento: Ingeniería de Sistemas y Automática.

Titulación: Máster en Ingeniería Mecatrónica.

Palabras clave: Conducción autónoma, vehículos autónomos, sistemas de percepcion, detección de obstáculos, simulador CARLA, ROS2 Humble, LiDAR 3D, cámara RGB, aprendizaje profundo, redes neuronales convolucionales, fusión multimodal, áreas urbanas.

Resumen

En las últimas décadas, la automatización ha transformado una gran variedad de sectores industriales, permitiendo que los procesos manuales sean reemplazados por sistemas automáticos, desde la fabricación hasta los servicios. Con ello, se ha conseguido una mejora de la eficiencia, la precisión y la seguridad de estos, reduciendo a su vez los errores humanos. Esta tendencia ha sido especialmente notable en sectores tecnológicos donde la robótica, combinada con el aprendizaje máquina, han revolucionado las capacidades de las máquinas para llevar a cabo tareas complejas.

Durante los últimos años el sector de la automoción ha sufrido una gran revolución con la incorporación de tecnologías automatizadas, que van desde sistemas de asistencia a la conducción, como el mantenimiento de carril o el frenado automático de emergencia, hasta los primeros desarrollos de vehículos completamente autónomos. Esto no solo permite una mejora en la experiencia del conductor, sino también un aumento en la seguridad vial, contribuyendo a la evolución hacia los vehículos que sustituyen, de manera progresiva, la necesidad de intervención humana.

En este contexto, los vehículos autónomos se presentan como una aplicación de la automatización en el sector automovilístico. Estos vehículos se encuentran equipados con sistemas de percepción que les permiten identificar el entorno, detectar

obstáculos, tomar decisiones y navegar sin intervención humana. Estos vehículos hacen uso de una amplia gama de sensores que capturan datos del entorno en tiempo real, entre los que se incluyen sensores LiDAR (*Light Detection and Ranging*), cámaras RGB, radar, GPS (*Global Positioning System*) o IMU (*Inertial Measure Unit*), junto a otros dispositivos de percepción.

Uno de los desafíos que presentan los vehículos autónomos es la integración eficaz de los datos provenientes de estos sensores. La fusión multi-modal, es decir, la combinación de datos de diferentes sensores, como LiDAR y cámaras, es de gran utilidad para poder representar el entorno de forma fiel y precisa. Por un lado, el sensor LiDAR proporciona información acerca de la forma de los objetos y su distancia al sensor, mientras que las cámaras capturan detalles de los objetos.

En el presente Trabajo de Fin de Máster se propone realizar una fusión multi-modal de los datos provenientes de un sensor LiDAR y cámaras RGB, con el fin de aprovechar las ventajas que ofrece cada sensor por separado. Con ello se espera superar las limitaciones que enfrentan de forma individual, mejorando así la robustez del sistema de percepción del vehículo.

Escuela de Ingenierías Industriales. Universidad de Málaga

Integration of traffic participant detection techniques in urban environments using a 3D LiDAR sensor and RGB cameras with CARLA and ROS2

Author: Jorge Montenegro Navarro.

Supervisor: Jesús Morales Rodríguez.

Department: *Systems and Automation Engineering.*

Academic Degree: *Master in Mechatronics Engineering.*

Keywords: *Autonomous driving, autonomous vehicles, perception systems, obstacle detection, CARLA Simulator, ROS2 Humble, 3D LiDAR, RGB camera, deep learning, convolutional neural networks, multi-modal fusion, urban areas.*

Abstract

In the last decades, automation has transformed a wide range of industrial sectors, allowing manual processes to be replaced by automatic systems, from manufacturing to services. This has led to improvements in efficiency, accuracy, and safety, while also reducing human error. This trend has been particularly notable in technological sectors, where robotics, combined with machine learning, have revolutionised the capabilities of machines to carry out complex tasks.

In recent years, the automotive sector has undergone a significant revolution with the introduction of automated technologies, from driver assistance systems, such as lane-keeping and automatic emergency braking, to the early development of fully autonomous vehicles. This not only enhances the driving experience but also increases road safety, contributing to the evolution towards vehicles that progressively reduce the need for human intervention.

In this context, autonomous vehicles stand as an application of automation in the automotive sector. These vehicles are equipped with perception systems that allow them to identify their surroundings, detect obstacles, make decisions, and navigate without human intervention. They make use of a wide range of sensors that capture real-time environmental data, including LiDAR (Light Detection and Ranging) sensors, RGB cameras, radar, GPS (Global Positioning System), and IMU (Inertial Measurement Unit), along with other perception devices. One of the challenges

faced by autonomous vehicles is the effective integration of data from these sensors. Multi-modal fusion, meaning the combination of data from different sensors, such as LiDAR and cameras, is key for the system to build an accurate and reliable representation of the environment. On the one hand, the LiDAR sensor provides information about the shape and distance of objects, while cameras capture detailed features of the objects.

In this End of Master Project, a multi-modal fusion of data from a LiDAR sensor and RGB cameras is proposed, in order to leverage the advantages offered by each sensor individually. This aims to overcome the limitations faced by each sensor on its own, thus improving the robustness of the vehicle's perception system.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que han contribuido de alguna manera a la realización de este trabajo.

En primer lugar, agradezco a mi tutor, Jesús Morales Rodríguez, por su apoyo, orientación y paciencia, a la hora de instalar paquetes y herramientas en Ubuntu. Agradecer también a mis alumnos del centro deportivo Vals Sport Cónsul, tanto a los de Body Pump y Body Combat, por ser más fieles que la sombra, como a mis chavalitos de natación, especialmente mis grupos "los paquetes", "los anguila eléctrica" y "los bichos". Además, agradezco a mis coordinadores Antonio, Fer, Paco y Ale por cuadrarme siempre el horario para poder compatibilizarlo con el de la universidad durante este año.

Por otro lado, agradecer a mi familia por apoyarme, confiar en mi, y ponerme los pies en la tierra para "no venirme demasiado arriba" con los logros que voy consiguiendo. Por último, agradecer al Proyecto de Investigación de Excelencia de la Junta de Andalucía PREMOVE (ProyExcel_00684) por la financiación para la realización del trabajo.

Acrónimos

2D	Bidimensional
3D	Tridimensional
AAE	*Average Attribute Error*
AOE	*Average Orientation Error*
AP	*Average Precision*
API	*Application Programming Interface*
ASE	*Average Scale Error*
ATE	*Average Translation Error*
AWS	*Amazon Web Services*
AVE	*Average Velocity Error*
BEVFusion	*Bird-Eye-View Fusion*
CARLA	*Car Learning to Act*
CUDA	*Compute Unified Device Architecture*
cuDNN	*CUDA Deep Neural Network*
DDS	*Data Distribution Service*
DETR	*Detection Transformers*
FOV	*Field of View*
FPS	*Frames Per Second*
GPU	*Graphics Processing Unit*
JSON	*JavaScript Object Notation*
LiDAR	*Light Detection and Ranging*

LSS	*Lift-Splat-Shoot*
MIT	*Massachusetts Institute of Technology*
PREMOVE	*Predicción del Movimiento de los participantes del tráfico para la integración segura del vehículo autónomo en áreas urbanas*
PV-RCNN	*PointVoxel Regional based Convolutional Neural Network*
QoS	*Quality of Service*
RGB	*Red, Green, Blue*
ROS2	*Robot Operating System*
RNA	*Redes Neuronales Artificiales*
RViz	*ROS Visualization*
SAE	*Society of Automotive Engineers*
TensorRT	*Tensor Real-Time*
TFM	Trabajo de Fin de Máster
YOLO	*You Only Look Once*

Índice

Capítulo 1

Introducción

Contenido

1.1. Antecedentes

A lo largo de las últimas décadas, la automatización ha transformado de manera radical diversos sectores industriales. Desde la década de 1960, con la aparición de los primeros robots industriales, como el Unimate, que se integró en las líneas de producción de General Motors en 1961, la automatización ha desempeñado un papel crucial en la evolución de sectores como la manufactura, la agricultura, la minería y los servicios [1]. Este avance permitió que, tareas que tradicionalmente requerían intervención humana, pudieran ser realizadas de manera automática, lo que provocó una mejora significativa en términos de precisión y eficiencia, reduciendo los errores humanos y aumentando la seguridad [2].

En las décadas de 1980 y 1990, la robótica y el aprendizaje máquina impulsaron una revolución en la capacidad de las máquinas para realizar tareas cada vez más complejas. Posteriormente, en 1997, el superordenador Deep Blue de IBM venció al campeón de ajedrez Garry Kasparov, demostrando el potencial de las máquinas para superar retos intelectuales [3]. Concretamente, dicha combinación ha sido especialmente útil en sectores tecnológicos, como la industria automotriz [4].

Desde principios de los años 2000, se han integrado sistemas automatizados en vehículos, como sistemas de asistencia a la conducción. Estos sistemas incluyen funciones como el mantenimiento de carril o el frenado automático de emergencia, que

han sido diseñados para mejorar tanto la experiencia de conducción como la seguridad vial. Posteriormente, en 2014, la Sociedad de Ingenieros de la Automoción (SAE) presentó una clasificación de los niveles de automatización en vehículos, que se muestra en la Figura 1.1, que van desde el nivel 0, sin automatización, hasta el nivel 5, que se corresponde con la plena autonomía del vehículo.

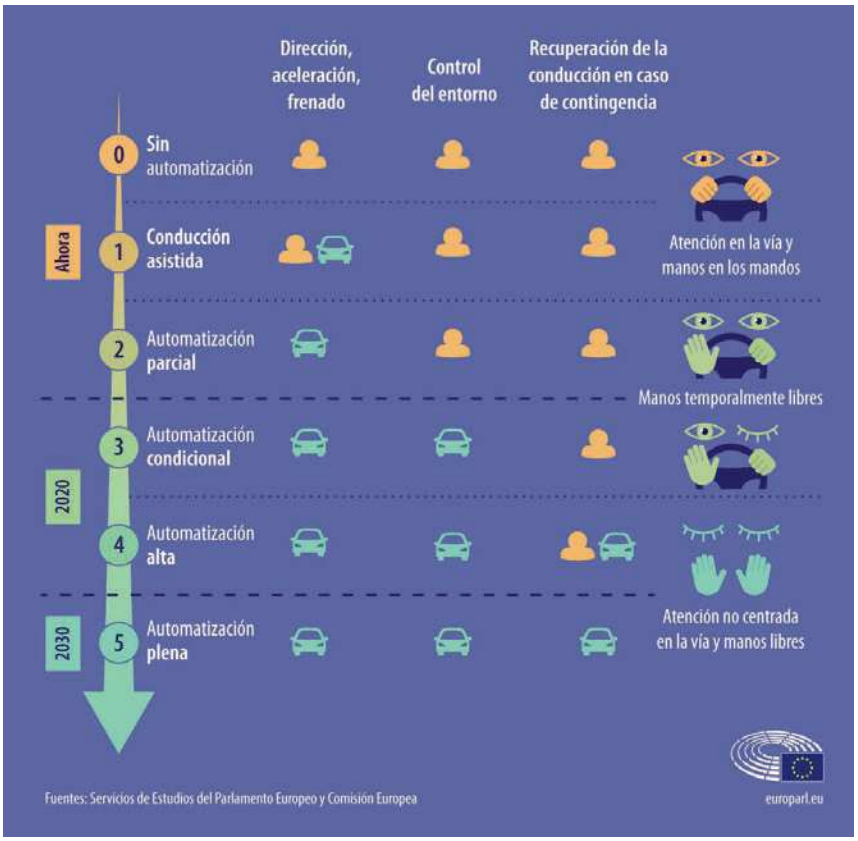


Figura 1.1: Niveles de conducción autónoma según la SAE [5].

En los últimos años, los avances en el campo de la robótica han permitido el desarrollo de los primeros prototipos de vehículos autónomos, capaces de operar sin intervención humana. Uno de los primeros vehículos autónomos surgió en 2009, cuando Google anunció su proyecto de coche autónomo, que posteriormente pasaría a formar parte de Waymo, una empresa independiente de tecnología de conducción autónoma. Esto marcó el inicio de empresas como Tesla, Waymo y Uber, que han apostado por estos vehículos [6], que no solo aportan una experiencia de conducción innovadora, sino que también suponen un avance en la reducción de accidentes y en la mejora de la eficiencia del transporte [7].

Dichos vehículos se encuentran equipados con múltiples sensores, como es el caso de las cámaras RGB [8], que capturan imágenes visuales o del LiDAR 3D (*Light Detection and Ranging*) [9], que representa el entorno mediante nubes de puntos [10]. En este contexto, previamente se han realizado proyectos que tratan sobre detección de participantes del tráfico sobre imágenes RGB y nubes de puntos LiDAR

3D por separado. Sin embargo, esto tiene la desventaja de que con las cámaras se pierde información espacial y con el LiDAR 3D no se detectan adecuadamente los peatones [10]. Es por ello que se presenta la fusión multi-modal de sensores, que es una técnica empleada para mejorar la fiabilidad de las detecciones de los sistemas de percepción. Se basa en combinar datos de diferentes fuentes para obtener una visión más completa del entorno superando las limitaciones que tienen de forma individual [11, 12]. Finalmente, en la Figura 1.2 se muestra un esquema de la arquitectura planteada para la fusión multi-modal de este trabajo.

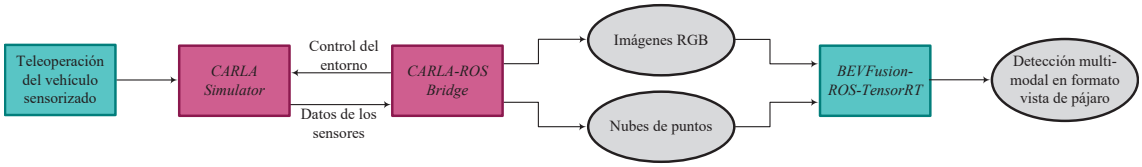


Figura 1.2: Esquema de la arquitectura del TFM. Los nodos de ROS2 se muestran con un rectángulo azul, mientras que los tópicos con elipses.

En la Figura 1.2 se muestra cómo, en primer lugar, el entorno se comunica, mediante ROS2, con el simulador CARLA modificando las condiciones de la simulación y permitiendo la teleoperación del vehículo sensorizado. Por otro lado, dicho simulador envía los datos de las cámaras RGB y del sensor LiDAR 3D, en forma de tópicos, a la red neuronal BEVFusion mediante ROS2. Finalmente la red neuronal realiza la inferencia y devuelve, en un tópico, las detecciones multi-modales en formato vista de pájaro.

1.2. Marco de realización

El presente Trabajo de Fin de Máster se enmarca dentro del proyecto PREMOVE (Predicción del Movimiento de los participantes del tráfico para la integración segura del vehículo autónomo en áreas urbanas), un proyecto de excelencia financiado por la Junta de Andalucía bajo la referencia *ProyExcel_00684* [13]. Este proyecto se lleva a cabo en el Departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga, con el objetivo de investigar y desarrollar técnicas para detectar y predecir el movimiento de los participantes del tráfico con el fin de evitar colisiones en áreas urbanas.

1.3. Objetivo

El presente Trabajo de Fin de Máster tiene dos principales objetivos. Por un lado, la realización de una fusión multi-modal de los datos de los sensores, integrando las detecciones realizadas a partir de un sensor LiDAR 3D y cámaras RGB,

empleando para ello el simulador realista CARLA (*Car Learning to Act*) junto al *middleware* ROS2 (*Robot Operating System*) y la red neuronal BEVFusion (*Bird-Eye-View Fusion*) [14]. Con ello se espera superar las limitaciones que enfrentan de forma individual. Por otro lado, como segundo objetivo se plantea el entrenamiento de la red neuronal mencionada para detectar participantes del tráfico en CARLA.

Además del objetivo principal, en este Trabajo de Fin de Máster se proponen una serie de objetivos adicionales que se espera lograr a lo largo de su desarrollo:

- Actualización del estado del arte sobre vehículos autónomos.
- Estudio de las características y últimos avances de los simuladores realistas.
- Adquisición de competencias en visión por computador.
- Adquisición de competencias en redes neuronales.
- Adquisición de competencias en fusión multi-modal.

1.4. Planificación del trabajo

La planificación del presente Trabajo de Fin de Máster se ha desarrollado como se muestra a continuación. En primer lugar, se investigará sobre el simulador CARLA y el *middleware* ROS2. Posteriormente, se realizará una revisión del estado del arte sobre aprendizaje profundo y detección de objetos empleando redes neuronales artificiales. Tras esto, se investigará sobre el funcionamiento de los sensores LiDAR 3D y las cámaras RGB y, tras esto, se integrarán técnicas de deteccción de objetos con dichos sensores. Finalmente se realizarán pruebas en simulación y se redactará la memoria del Trabajo de Fin de Máster.

1.5. Estructura del documento

La estructura del proyecto es la siguiente:

- En el Capítulo 1 de introducción se contextualiza el proyecto y se describe de forma general el trabajo a realizar.
- Posteriormente, en el Capítulo 2 se describen las herramientas utilizadas, tanto *hardware* como *software*.
- En el Capítulo 3 se explica la configuración y simulación de los sensores LiDAR y cámaras en CARLA, así como el escenario escogido.
- Tras esto, en el Capítulo 4 se describe la red neuronal empleada para detectar participantes del tráfico.

- En el Capítulo 5 se presentan los resultados obtenidos, evaluando el rendimiento del sistema de fusión multi-modal y su precisión en la detección.
- Finalmente, en el Capítulo 6 se presentan las conclusiones del trabajo y se proponen mejoras y futuras líneas de investigación.
- Como apéndices, se incluye el Anexo A, donde se encuentra una guía de instalación y el Anexo B, donde se encuentra una guía de usuario para la configuración y uso del proyecto.

Capítulo 2

Herramientas empleadas

Contenido

2.1. Introducción

En el presente capítulo se describen con detenimiento las herramientas empleadas, tanto a nivel *hardware*, que es la estación de trabajo, como a nivel *software*, que son CARLA [15], ROS2 [16], CARLA-ROS-Bridge [17], Pytorch [18], CUDA [19], cuDNN [20], TensorRT [21], NVIDIA-BEVFusion [14, 22] y BEVFusion-ROS-TensorRT [23].

2.2. Estación de trabajo

La estación de trabajo empleada para entrenar la red neuronal y ejecutar el simulador junto a la red neuronal cuenta con las características que se detallan a continuación.

2.2.1. Equipo

- **Procesador:** Intel Core™ i7-13700F a 5,2 GHz con 16 núcleos, 24 hilos y 16 MB de caché L2.
- **Tarjeta gráfica:** 4xGIGABYTE RTX 4070 TI GAMING OC con 12 GB de memoria.
- **Memoria RAM:** DDR5 2x16 GB a 5600 MHz.
- **Almacenamiento:** Samsung SSD 980 PRO 1 TB + HDD Seagate BarraCuda 3.5"4 TB SATA3.
- **Sistema operativo:** Ubuntu 22.04.
- **Periféricos:** Juego de volante y pedales Logitech G27.

2.3. CARLA Simulator

Es un simulador utilizado en el ámbito de la conducción autónoma que facilita el entrenamiento, validación y prueba de sistemas de conducción de vehículos autónomos en entornos urbanos dinámicos. Dicho simulador fue desarrollado en 2017 como un proyecto del equipo del Centro de Visión por Computador de la Universidad Autónoma de Barcelona [24] en colaboración con la industria: *Intel Labs* [25] y *Toyota Reseach Institute* [26]. En sus orígenes, CARLA se basó sobre Unreal Engine 4, un motor gráfico que permite una simulación realista del entorno tanto de forma visual como física. Además, dicho motor es el que facilita la personalización del entorno por capas, denominado *blueprints*, y los actores de la simulación, que es como se denomina todo aquello que interactua con el entorno, como los vehículos y los peatones [27]. Las principales características del simulador CARLA se muestran a continuación:

- **Simulación de Sensores:** CARLA permite la integración de una gran variedad de sensores, tales como cámaras RGB, cámaras de profundidad, sensores LiDAR, GPS y RADAR. Además, dichos sensores son fácilmente parametrizables, permitiendo modificar sus características mediante un archivo de configuración de extensión *yaml*, lo que permite replicar fielmente el comportamiento de los sensores del mundo real.

- **Configuración del tráfico y las condiciones meteorológicas:** CARLA permite modificar tanto el tráfico, creando vehículos y peatones, denominados actores, como las condiciones climáticas, como lluvia, niebla, viento e iluminación, permitiendo evaluar el comportamiento de los vehículos autónomos bajo diferentes escenarios. Además, en las últimas versiones se han añadido mejoras que permiten controlar la opacidad de la niebla y la intensidad del viento, lo que añade mayor realismo a las simulaciones.

- **Modo de simulación rápida:** Este modo permite realizar simulaciones sin tener que renderizar gráficos, lo que es de utilidad para validar algoritmos de planificación y control para los que no sea necesaria la visualización, mejorando así la eficiencia de las simulaciones.

- **Uso de mapas reales:** CARLA permite el uso de un gran abanico de mapas, los cuales fueron recientemente ampliados en la versión 0.9.15. Además, empleando herramientas como *OpenStreetMap* [28] o *RoadRunner* [29] es posible crear un mapa desde cero o recrear un entorno real.

- **Integración de CARLA con ROS2:** Mediante la utilidad CARLA-ROS-Bridge, es posible comunicar de forma bidireccional el simulador con el entorno ROS2, lo que permite modificar parámetros del simulador y obtener datos del mismo.

- **Uso de APIs (*Application Programming Interface*):** CARLA proporciona una API en Python y C++ que permite controlar todos los aspectos de la simulacion de forma sencilla, como es la creación de sensores, el control del tráfico o las condiciones meteorológicas.

Por otro lado, el simulador CARLA cuenta con dos versiones. Por un lado, existe la versión denominada *Package*, usada en este proyecto, que está precompilada y lista para usar, diseñada para ejecutar simulaciones sin necesidad de modificar el código, como se puede observar en la Figura 2.1. Por otro lado, existe la versión denominada *Source*, que permite acceso total al código y la capacidad de personalizar el simulador desde el motor gráfico *Unreal Engine 4*, como se puede observar en la Figura 2.2.

Figura 2.1: Visualización de la versión *Package* del simulador CARLA [30].

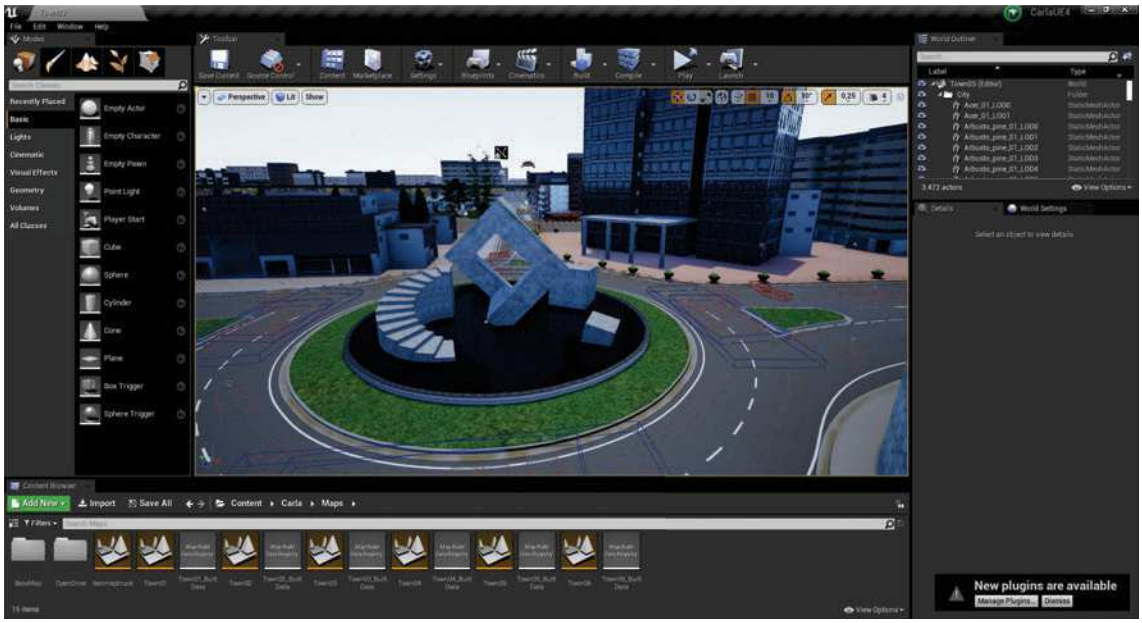


Figura 2.2: Visualización de la versión *Source* del simulador CARLA en Unreal Engine 4 [31].

Finalmente, para desarrollar el presente trabajo se ha hecho uso de la versión 0.9.15 del simulador CARLA en su distribución Package, la cual añade, como novedad, la capacidad de distribuir el trabajo requerido por el simulador en varias GPUs (Unidades de Procesamiento Gráfico) y el uso de gemelos digitales para generar mapas de CARLA de forma más sencilla [32].

2.4. ROS2

ROS2 es un *middleware* utilizado en aplicaciones robóticas que actúa como nexo entre diferentes componentes *hardware* y *software*, simplificando las comunicaciones y el flujo de información entre estos, denominados nodos [33]. Dicho entorno surge como la evolución de su predecesor ROS, superando sus limitaciones, como la dependencia de un único nodo central o la imposibilidad de trabajar con sistemas en tiempo real.

En cuanto a su arquitectura, que se muestra en la Figura 2.3, ROS2 se compone de nodos, que corresponden con programas o procesos que realizan una tarea específica, como procesar datos, actuar sobre el entorno o comunicarse entre sí [33]. Para ello, existen nodos del tipo publicador, que se encargan de transmitir datos por un canal, denominado tópico, mientras que existen otros nodos, denominados suscriptores, que se encargan de recibir esos datos.

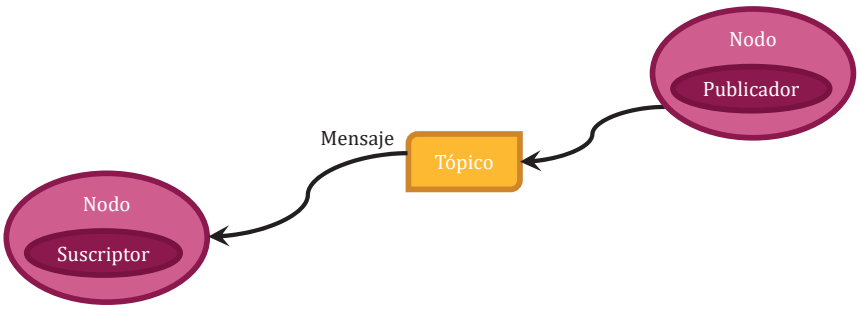


Figura 2.3: Esquema de la arquitectura de ROS2.

Las principales características de ROS2 [33] se muestran a continuación:

- **Arquitectura distribuida:** A diferencia de ROS1, ROS2 en lugar de depender de un nodo centralizado (*roscore*) para coordinar la comunicación, permite que los nodos se comuniquen directamente entre sí, mejorando la robustez y escalabilidad en aplicaciones complejas.
- **Capa de abstracción *middleware*:** En el centro de la arquitectura de ROS2 se encuentra la capa *middleware*, llamada RMW (*ROS Middleware*), que permite la gestión de manera eficiente de la comunicación entre nodos y sistemas distribuidos. Además, permite que ROS2 sea compatible con diferentes tecnologías de comunicación sin cambiar el código de los nodos.
- **Soporte multiplataforma:** ROS2 está diseñado para funcionar tanto en Linux, Windows como macOS, lo que proporciona flexibilidad en términos de entornos de desarrollo. Esto es posible gracias a la modularidad y a la implementación basada en DDS (*Data Distributed Service*).

- **Soporte para tiempo real:** Gracias a la implementación basada en DDS y en la calidad de servicio QoS (*Quality of Service*), ROS2 garantiza el envío y recepción de mensajes entre nodos en tiempo real cumpliendo cumpliendo ciertos tiempos. Es por ello, que son especialmente adecuados en aplicaciones que requieran seguridad y tiempo de respuesta específicos, como los vehículos autónomos.

- **Compatibilidad con ROS1:** Mediante el paquete ROS1_Bridge [34], es posible comunicarse bidireccionalmente entre ROS1 y ROS2, lo que permite aprovechar las ventajas en ROS2 empleando paquetes de ROS1.

- **Integración en la nube:** La arquitectura de ROS 2 facilita la integración con servicios en la nube, como IoT RoboRunner de AWS (*Amazon Web Services*).

Finalmente, existen varias herramientas integradas en ROS2, como son rviz2 o ROS2 Bag. Por un lado, RViz (*ROS Visualization*) es una herramienta que permite visualizar los datos de los tópicos procedentes de sensores, como pueden ser cámaras RGB, cámaras térmicas, LiDAR o RADAR. Además, con esta herramienta también es posible depurar y desarrollar aplicaciones, ofreciendo una representación gráfica intuitiva de mapas, trayectorias, y transformaciones entre los distintos sistemas de referencia. Por otro lado, mediante la herramienta ROS2 Bag es posible capturar datos en tiempo real de tópicos, como información de sensores para, posteriormente, procesarlos o visualizarlos mediante RViz. En la Figura 2.4 se muestra un ejemplo de visualización mediante RViz de datos grabados mediante ROS2 Bag.

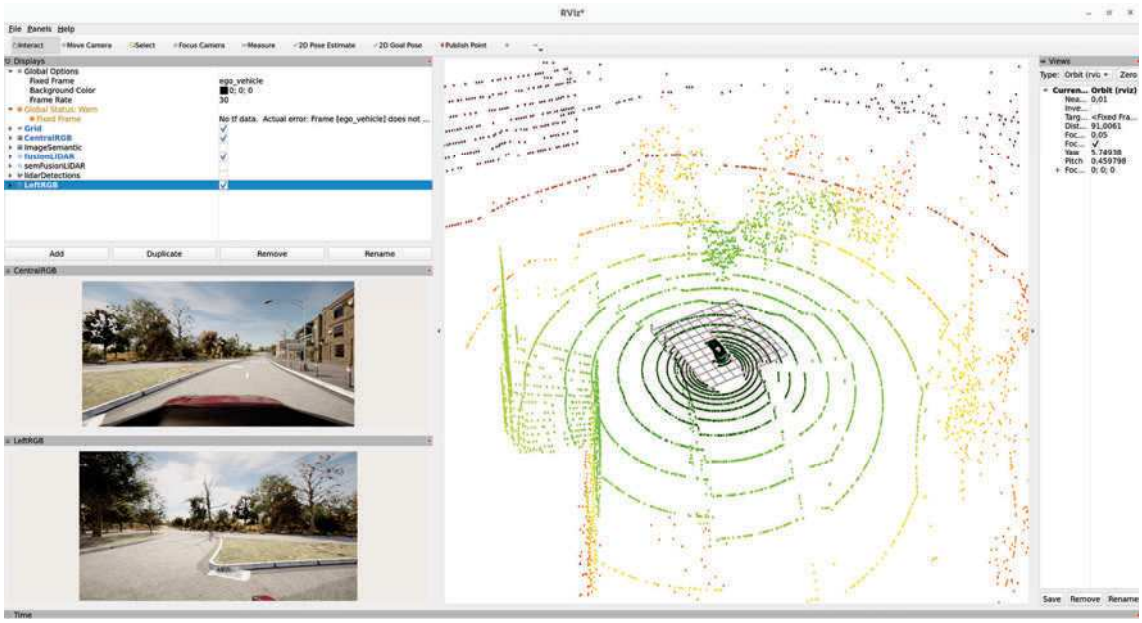


Figura 2.4: Visualización de imágenes RGB y nubes de puntos 3D grabadas con ROS2 Bag mediante RViz.

2.5. CARLA-ROS-Bridge

CARLA-ROS-Bridge es un paquete que permite la comunicación bidireccional entre el simulador CARLA y el *middleware* ROS2. Para ello, dicho paquete se encarga de publicar y suscribirse a tópicos dentro de ROS2, permitiendo la transferencia de datos de los sensores simulados, como cámaras RGB o LiDAR 3D hacia nodos de ROS2, los cuales pueden procesar esta información como si de sensores reales proviniesen. Además, dado el carácter bidireccional del paquete, es posible, mediante ROS2, controlar aspectos propios del simulador, como modificar parámetros del simulador, la creación de un vehículo sensorizado o su teleoperación.

2.6. *PyTorch*

PyTorch es una librería de código abierto desarrollada por el Laboratorio de Investigación de Inteligencia Artificial de Facebook para el desarrollo y entrenamiento de modelos de redes neuronales profundas. Dicha librería facilita el trabajo con tensores y admite el procesamiento por GPU (*Graphics Processing Unit*), lo que la convierte en una herramienta ampliamente utilizada en el entrenamiento y validación de modelos de redes neuronales. De entre las múltiples aplicaciones de *PyTorch* destacan la detección de objetos en imágenes, el procesamiento de lenguaje natural o el desarrollo de modelos generativos [35].

2.7. CUDA

CUDA (*Compute Unified Device Architecture*) es un entorno desarrollado por NVIDIA que permite el procesamiento paralelo, aprovechando la potencia de las GPU para acelerar cálculos costosos computacionalmente y optimizar el manejo de grandes cantidades de datos [19]. De entre sus áreas de aplicación, destaca el aprendizaje profundo, donde se emplea para el entrenamiento de redes neuronales que demanda una alta carga computacional. Las principales características de CUDA [36] se muestran a continuación:

- **Capacidad de procesamiento paralelo:** CUDA permite que varios núcleos de procesamiento en una GPU realicen cálculos simultáneamente, proporcionando un mejor rendimiento frente a las CPU en tareas que pueden ser paralelizadas.
- **Versatilidad:** CUDA es compatible con varios lenguajes de programación, como son C, C++ y Python, lo que lo hace aplicable a un gran abanico de proyectos.
- **Gestión de memoria:** CUDA ofrece un control eficiente y detallado sobre el uso de memoria en las GPU, lo que permite optimizar el rendimiento.

2.8. NVIDIA cuDNN

cuDNN (*CUDA Deep Neural Network*) es una librería optimizada para redes neuronales profundas, diseñada para ofrecer implementaciones eficientes de primitivas utilizadas en modelos de dichas redes. Concretamente, cuDNN se encarga de facilitar la optimización de los núcleos necesarios en tareas relacionadas con redes neuronales profundas [37, 38]. De entre las tareas que cuDNN se encarga de optimizar, se encuentran operaciones comunes como la convolución, el *pooling* y las funciones de activación de las neuronas, tanto en las fases de propagación hacia adelante como hacia atrás, necesarias para entrenar redes neuronales profundas.

2.9. NVIDIA TensorRT

NVIDIA TensorRT es una plataforma de optimización y ejecución de redes neuronales profundas que permite mejorar el rendimiento de la inferencia en GPUs. En el campo del aprendizaje profundo, TensorRT se caracteriza por reducir la latencia optimizar el uso de memoria en modelos de redes neuronales, facilitando una inferencia en tiempo real en aplicaciones críticas como la conducción autónoma y la visión por computador. A través de técnicas como la cuantización y la fusión de capas, TensorRT adapta los modelos para mejorar la eficiencia en el *hardware* de NVIDIA, siendo compatible con librerías como PyTorch y TensorFlow [21, 39].

2.10. BEVFusion

BEVFusion (*Bird-Eye-View Fusion*) es un proyecto desarrollado para la fusión multi-modal de las detecciones de un sensor tipo LiDAR 3D y seis cámaras RGB aplicado a vehículos autónomos. Este se caracteriza por proyectar las detecciones de las cámaras sobre las nubes de puntos del LiDAR a vista de pájaro, lo que permite conservar tanto la información geométrica como semántica, permitiendo una detección precisa de los participantes del tráfico [14]. Respecto a otros métodos de fusión multi-modal como Transfusion [40], BEVFusion destaca por una mayor velocidad de inferencia y por su toleracia al fallo ya que, al estar separadas las modalidades de cámara y LiDAR en flujos independientes, el sistema puede seguir funcionando en caso de que el sensor LiDAR falle o este se vea afectado por condiciones meteorológicas, como la lluvia [14, 40].

2.11. BEVFusion-ROS2-TensorRT

BEVFusion-ROS2-TensorRT es un paquete creado para ROS2 Galactic que sirve como *wrapper* de BEVFusion para ROS2 [23]. Dicho paquete no emplea la versión

original de BEVFusion desarrollada por Mit-Han-Lab, sino que utiliza la solución de NVIDIA para BEVFusion, que optimiza la inferencia en tiempo real con TensorRT [41]. En cuanto a su estructura, este paquete de ROS2 cuenta con un archivo tipo *launch* con el cual se cargan los parámetros de la red a utilizar y el nodo principal, denominado *bevfusion_ node*, donde se define ocho tópicos: siete suscriptores y uno publicador. Los tópicos suscriptores se encargan de obtener las imágenes de las seis cámaras RGB y la nube de puntos del LiDAR 3D, mientras que el tópico publicador se encarga de mostrar el resultado de las detecciones, como se muestra en la Figura 2.5. Además, dicho nodo es el que se encarga de preprocesar los datos del LiDAR y mandarlos, junto con las imágenes de las cámaras RGB a la red BEVFusion para realizar la inferencia.

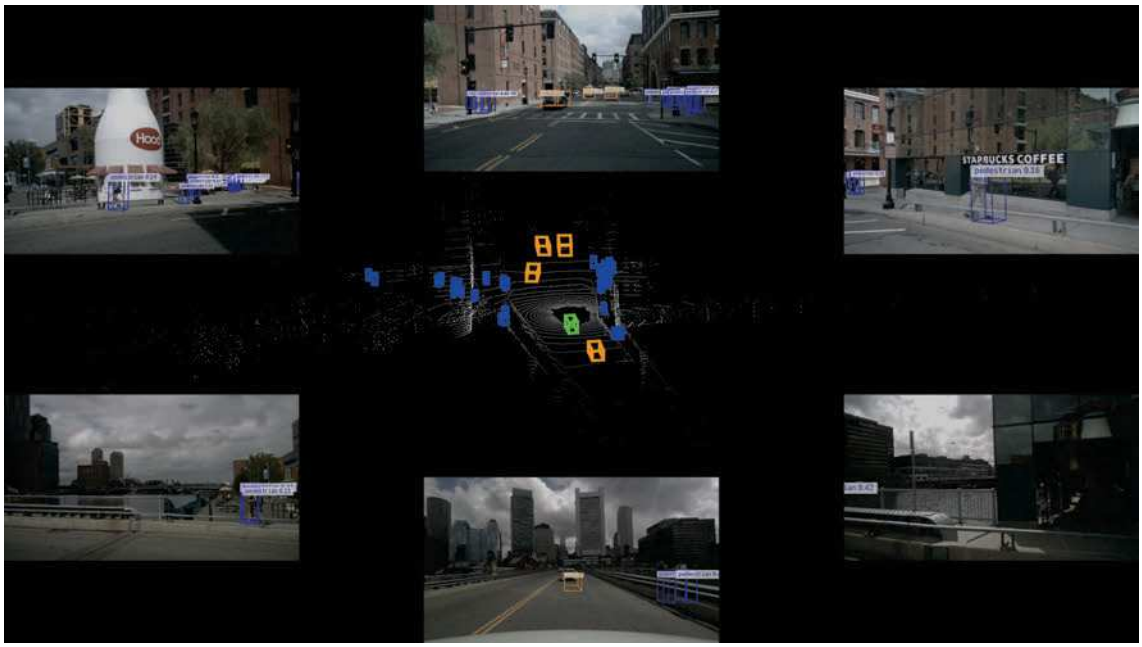


Figura 2.5: Resultado de las detecciones en BEVFusion-ROS-TensorRT [23].

2.12. Conjunto de datos nuScenes

nuScenes es un conjunto de datos de código abierto utilizado en la investigación, desarrollo y validación de sistemas de conducción autónoma. Este conjunto de datos contiene datos anotados de múltiples sensores, en concreto, seis cámaras RGB, un sensor LiDAR 3D, radar y sistemas de posicionamiento GPS, obtenido mediante un vehículo sensorizado, que es un Renault Zoe [42]. La versión completa, denominada `v1.0-trainval` ofrece más de 1000 escenas obtenidas bajo distintas condiciones climáticas, mientras que versiones más reducidas, como la `v1.0-mini`, contienen un menor número de escenas, lo que permite realizar pruebas rápidas.

2.13. Conjunto de datos AdverseOp3D

AdverseOp3D es un conjunto de datos desarrollado en el proyecto ContextualFusion [43] y se basa en el formato y configuración sensorial de nuScenes para extender el conjunto de datos de nuScenes con escenas en condiciones adversas del simulador CARLA. Esto permite una mayor generalización del conjunto de datos a la hora de validar modelos de detección de objetos y la posibilidad de probarlos en simulación.

Capítulo 3

Simulación del sistema sensorial multi-modal en CARLA

Contenido

3.1. Introducción

En el presente capítulo se presenta en detalle el simulador CARLA y su puesta en marcha particularizada para este trabajo, incluyendo el modelo del vehículo empleado para las simulaciones, su configuación sensorial y cómo se controla. Además, se detallan las condiciones de la simulación, como el clima, el escenario o los participantes del tráfico.

3.2. Modelo del vehículo

En el presente proyecto se ha tomado como referencia el vehículo empleado por nuScenes para crear su conjunto de datos, que es un Renault Zoe [44]. Sin embargo,

dado que en el simulador CARLA la selección de vehículos disponibles es limitada y dicho vehículo no se encuentra entre ellos, se plantean dos escenarios posibles. Por un lado se podría modelar desde cero, pero sería un proceso bastante extenso que requeriría conocimientos de Unreal Engine 4, lo cual sale del alcance del presente trabajo. La otra solución consiste en buscar físicamente el vehículo más similar al Renault Zoe de entre los que ofrece CARLA, que es por la que se ha optado. Finalmente el vehículo escogido para realizar las simulaciones ha sido el Lincoln MKZ 2020, perteneciente al fabricante Lincoln, una submarca de automoción de gama alta de Ford. El vehículo se muestra en la Figura 3.1.

Figura 3.1: Modelo 3D del vehículo usado en las simulaciones [15].

3.2.1. Sensores utilizados

Una vez elegido el vehículo a emplear en las simulaciones, el siguiente paso es decidir qué sensores emplear y su modelo. Como se mencionó en el Capítulo 1, el vehículo contará con seis cámaras RGB y un sensor LiDAR 3D, para lo cuál se han escogido los mismos modelos que utiliza nuScenes [42], cuyas características se detallan a continuación.

LiDAR 3D

El sensor LiDAR 3D utilizado cuenta con las características mostradas en la Tabla 3.1. Además, en la Figura 3.2 se muestra una imagen genérica de un LiDAR 3D.

Característica	**Valor**
Alcance máximo	$\leq$ 100 m
Puntos por revolución	35,000 puntos a 20 Hz
Campo de visión vertical (FOV)	$\approx$ 41,33 (+10.7º a -30.7º)
Campo de visión horizontal (FOV)	180º
Velocidad de rotación	5-20 Hz
Resolución vertical	V: 1.33º
Resolución horizontal	H: 0.1º-0.4º
Número de haces	32 haces

Tabla 3.1: Especificaciones del sensor LiDAR 3D.

Figura 3.2: Imagen de un LiDAR 3D [45].

Cámaras RGB

Las cámaras RGB elegidas para el vehículo cuentan con las características mostradas en la Tabla 3.2, mientras que en la Figura 3.3 se muestra una imagen genérica de un cámara RGB.

Característica	**Valor**
Tipo de cámara	RGB
Frecuencia de captura	60 Hz
Tecnología del sensor	CMOS
Tamaño del sensor	1/1.8"
Resolución	1600x1200 px
FOV	50º

Tabla 3.2: Especificaciones de las cámaras RGB.

Figura 3.3: Imagen de una cámara RGB [46].

3.2.2. Disposición espacial de los sensores

Siguiendo con la reconstrucción del vehículo de pruebas de nuScenes en el simulador CARLA, conocidos los sensores que se pretenden montar, el siguiente paso es conocer su disposición espacial, es decir, dónde irán montados sobre el vehículo Lincoln MKZ 2020 de CARLA. Para ello, según [44, 42] los sensores deben instalarse como se muestra en la Figura 3.4

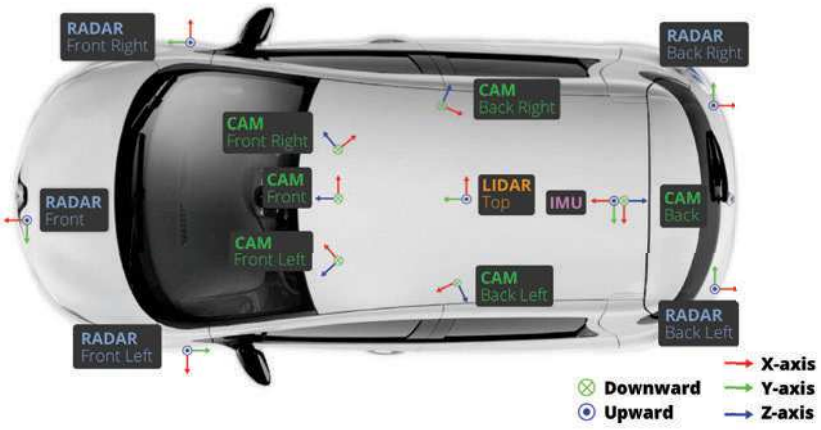
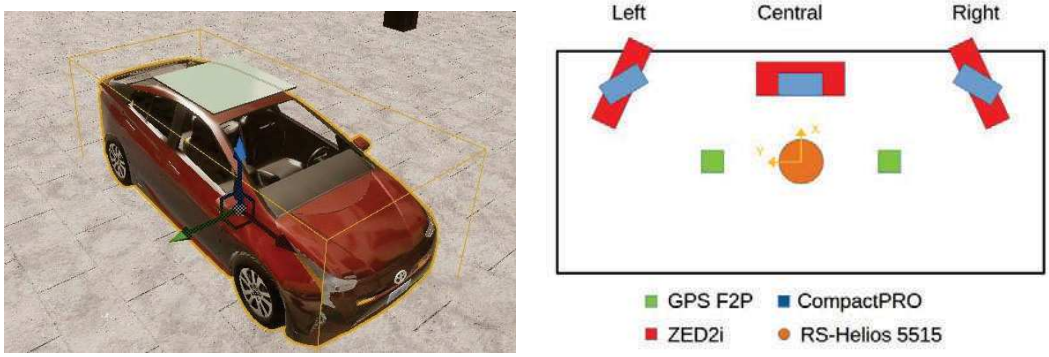


Figura 3.4: Configuración sensorial del vehículo de nuScenes [42].

Para obtener la posición de cada uno de los sensores, se ha recurrido a las matrices de calibración extrínseca proporcionadas por el conjunto de datos nuScenes, concretamente en el fichero `calibrated_sensor.json`. Además, cabe destacar que el origen de coordenadas se encuentra centrado en la base del vehículo al igual que en CARLA, con la diferencia de que, en este último, el vector de desplazamiento horizontal, y, se obtiene con el sistema de la mano izquierda, en vez del usado en nuScenes y en los archivos *json* de CARLA-ROS-Bridge, que emplea el de la mano derecha [47], como se puede observar en la Figura 3.5. Sin embargo, este discrepancia en los sistemas de referencia ya está cubierta por el paquete CARLA-ROS-Bridge, por lo que sobre el archivo *json* de configuración de los sensores del vehículo habrá que seguir el sistema de la mano derecha.



(a) CARLA (mano izquierda). (b) ROS2 (mano derecha).

Figura 3.5: Comparación de los sistemas de referencia [47].

LiDAR 3D

Para el caso del LiDAR 3D, la matriz de calibración extrínseca del LiDAR respecto al sistema de referencia del vehículo se muestra en la expresión 3.1, donde V denota sistema de referencia del vehículo y T denota transformación.

$$ {}^{V}T_{LiDAR} = \begin{bmatrix} 1{,}0000 & 0{,}0000 & 0{,}0000 & 0{,}9437 \\ 0{,}0000 & 1{,}0000 & 0{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 1{,}8402 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.1} $$

En la expresión 3.1, las tres primeras filas de la última columna corresponden a la posición $[x, y, z]$ del LiDAR respecto al vehículo en metros.

Cámaras RGB

Para el caso de las cámaras RGB, las matrices de calibración extrínseca de cada una de las cámaras respecto al LiDAR se muestran en las expresiones 3.2 - 3.7, donde F denota Front, FR denota *Front Right*, FL denota *Front Left*, B denota *Back*, BL denota *Back Left* y BR denota *Back Right*.

$$ {}^{LiDAR}T_{F} = \begin{bmatrix} 0{,}9999 & 0{,}0067 & 0{,}0036 & -0{,}0126 \\ 0{,}0035 & 0{,}0186 & 0{,}9998 & 0{,}7649 \\ 0{,}0068 & -0{,}9998 & 0{,}0186 & -0{,}3109 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2} $$

$$ {}^{LiDAR}T_{FR} = \begin{bmatrix} 0{,}5517 & -0{,}0105 & 0{,}8340 & 0{,}4965 \\ -0{,}8337 & 0{,}0242 & 0{,}5517 & 0{,}6142 \\ -0{,}0260 & -0{,}9997 & 0{,}0046 & -0{,}3268 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3} $$

$$
{}^{LiDAR}T_{FL} = \begin{bmatrix} 0{,}5729 & 0{,}0027 & -0{,}8196 & -0{,}4917 \\ 0{,}8193 & 0{,}0239 & 0{,}5728 & 0{,}5891 \\ 0{,}0211 & -0{,}9997 & 0{,}0115 & -0{,}3196 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}
$$

$$
{}^{LiDAR}T_{B} = \begin{bmatrix} -0{,}9999 & 0{,}0098 & -0{,}0046 & -0{,}0038 \\ 0{,}0047 & 0{,}0075 & -0{,}9999 & -0{,}9088 \\ -0{,}0098 & -0{,}9999 & -0{,}0075 & -0{,}2833 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}
$$

$$
{}^{LiDAR}T_{BL} = \begin{bmatrix} -0{,}3171 & 0{,}0199 & -0{,}9482 & -0{,}4831 \\ 0{,}9481 & 0{,}0329 & -0{,}3163 & 0{,}0991 \\ 0{,}0249 & -0{,}9993 & -0{,}0293 & -0{,}2498 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}
$$

$$
{}^{LiDAR}T_{BR} = \begin{bmatrix} -0{,}3569 & -0{,}0055 & 0{,}9341 & 0{,}4823 \\ -0{,}9335 & 0{,}0401 & -0{,}3564 & 0{,}0768 \\ -0{,}0355 & -0{,}9992 & -0{,}0194 & -0{,}2732 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.7}
$$

Finalmente, para expresar un punto del sistema de referencia de nuScenes (ver Figura 3.4) en el sistema de referencia de ROS2 (ver Figura 3.5b), es necesario realizar una rotación de -90º en torno al eje z. Con lo cual, la expresión genérica de un punto de la cámara en el sistema de referencia de ROS2 se muestra en 3.8.

$$
\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}_{\text{Cámara}} = {}^{V}T_{\text{LiDAR}}\, R_z(-90^\circ)\, {}^{\text{LiDAR}}T_{\text{Cámara}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{3.8}
$$

Siendo R_z la matriz de rotación sobre el eje z, que se muestra en la expresión 3.9.

$$
R_z(\varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}
$$

Con ello, las posiciones de los sensores respecto al sistema de referencia del vehículo se muestran en la Tabla 3.3.

Cámara	$x(m)$	$y(m)$	$z(m)$
Frontal Central	0.3868	-0.0021	1.8110
Frontal Derecha	0.2099	-0.5126	1.8093
Frontal Izquierda	0.2368	0.4754	1.7957
Trasera Central	-1.2857	-0.0215	1.8791
Trasera Izquierda	-0.2991	0.6000	1.8624
Trasera Derecha	-0.2783	-0.5028	1.8910

Tabla 3.3: Posiciones de las seis cámaras RGB en el sistema de referencia de CARLA.

A modo de comprobación se puede observar lo siguiente.

- Todas las cámaras se encuentran prácticamente a la misma altura (componente z).
- La cámara derecha se encuentra a una separación similar (x) respecto a su homóloga izquierda, tanto en el caso de las cámaras frontales como en las traseras.
- Las coordenadas y de las cámaras siempre se encuentran ordenadas según el sistema de referencia de la Figura 3.5b, es decir, $y_{FL} > y_F > y_{FR}$.

Finalmente, del archivo `calibrated_sensor.json` también es posible obtener la matriz de calibración intrínseca de cada una de las cámaras, las cuales se muestran en las expresiones 3.10 - 3.15, donde F denota Front, FR denota *Front Right*, FL denota *Front Left*, B denota *Back*, BL denota *Back Left* y BR denota *Back Right*.

$$K_F = \begin{bmatrix} 1266{,}4172 & 0{,}0000 & 816{,}2670 & 0{,}0000 \\ 0{,}0000 & 1266{,}4172 & 491{,}5071 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.10}$$

$$K_{FR} = \begin{bmatrix} 1260{,}8474 & 0{,}0000 & 807{,}9683 & 0{,}0000 \\ 0{,}0000 & 1260{,}8474 & 495{,}3344 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.11}$$

$$K_{FL} = \begin{bmatrix} 1272{,}5979 & 0{,}0000 & 826{,}6155 & 0{,}0000 \\ 0{,}0000 & 1272{,}5979 & 479{,}7516 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.12}$$

$$K_B = \begin{bmatrix} 809{,}2210 & 0{,}0000 & 829{,}2196 & 0{,}0000 \\ 0{,}0000 & 809{,}2210 & 481{,}7784 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.13}$$

$$K_{BL} = \begin{bmatrix} 1256{,}7415 & 0{,}0000 & 792{,}1126 & 0{,}0000 \\ 0{,}0000 & 1256{,}7415 & 492{,}7758 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.14}$$

$$K_{BR} = \begin{bmatrix} 1259{,}5138 & 0{,}0000 & 807{,}2529 & 0{,}0000 \\ 0{,}0000 & 1259{,}5138 & 501{,}1958 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 1{,}0000 & 0{,}0000 \\ 0{,}0000 & 0{,}0000 & 0{,}0000 & 1{,}0000 \end{bmatrix} \tag{3.15}$$

3.2.3. Implementación en CARLA Simulator

Una vez conocido el modelo de vehículo sobre el cual realizar las simulaciones y los sensores a instalar, junto a sus características y ubicación, ya es posible simularlo en CARLA. Para ello, es necesario modificar un archivo *json* donde establecer todas estas características. En el Listado 3.1 se muestra una versión simplificada de dicho archivo con las modificaciones necesarias.

Listado 3.1: Archivo *json* con los sensores del vehículo

```
{
    "objects":
    [
        {
            "type": "vehicle.lincoln.mkz_2020",
            "id": "ego_vehicle",
            "spawn_point":{"x": 400.00, "y": -167.34, "z": 120.1, "
               roll": 0.0, "pitch": 0.6, "yaw": -40.00} ,
            "sensors":
            [

                {
                    "type": "sensor.camera.rgb",
                    "id": "Front_Left_RGB_1",
                    "spawn_point":{"x": 0.2368, "y": 0.4754, "z":
                       1.7957, "roll": 0.0, "pitch": 0.0, "yaw":
                       55.0},
                    "image_size_x": 1600, "image_size_y": 900, "fov
                       ": 50, "fstop":1.8,
                    "sensor_tick": 0.03333, "shutter_speed":0.03333
                },
                {
                    "type": "sensor.camera.rgb",
```

```
            "id": "Front_Central_RGB_1",
            "spawn_point":{"x": 0.3868, "y": -0.0021, "z":
               1.8110, "roll": 0.0, "pitch": 0.0, "yaw":
               0.0},
            "image_size_x": 1600, "image_size_y": 900, "fov
               ": 50, "fstop":1.8,
            "sensor_tick": 0.03333, "shutter_speed":0.03333
        },
        {
            "type": "sensor.camera.rgb",
            "id": "Front_Right_RGB_1",
            "spawn_point":{"x": 0.2099, "y": -0.5126, "z":
               1.8093, "roll": 0.0, "pitch": 0.0, "yaw":
               -55.0},
            "image_size_x": 1600, "image_size_y": 900, "fov
               ": 50, "fstop":1.8,
            "sensor_tick": 0.03333, "shutter_speed":0.03333
        },


        {
            "type": "sensor.camera.rgb",
            "id": "Back_Left_RGB_1",
            "spawn_point":{"x": -0.2991, "y": 0.6, "z":
               1.8624, "roll": 0.0, "pitch": 0.0, "yaw":
               110.0},
            "image_size_x": 1600, "image_size_y": 900, "fov
               ": 50, "fstop":1.8,
            "sensor_tick": 0.03333, "shutter_speed":0.03333
        },
        {
            "type": "sensor.camera.rgb",
            "id": "Back_Central_RGB_1",
            "spawn_point":{"x": -1.2857, "y": -0.0215, "z":
                1.8791, "roll": 0.0, "pitch": 0.0, "yaw":
               180.0},
            "image_size_x": 1600, "image_size_y": 900, "fov
               ": 110, "fstop":1.8,
            "sensor_tick": 0.03333, "shutter_speed":0.03333
        },
        {
            "type": "sensor.camera.rgb",
            "id": "Back_Right_RGB_1",
            "spawn_point":{"x": -0.2783, "y": -0.5028, "z":
                1.8910, "roll": 0.0, "pitch": 0.0, "yaw":
               -110.0},
            "image_size_x": 1600, "image_size_y": 900, "fov
               ": 50, "fstop":1.8,
            "sensor_tick": 0.03333, "shutter_speed":0.03333
        },


        {
            "type": "sensor.lidar.ray_cast",
```

```
                    "id": "LIDAR",
                    "spawn_point": {"x": 0.9437, "y": 0.0, "z":
                       1.8402, "roll": 0.0, "pitch": 0.0, "yaw":
                       -90.0},
                    "range": 100,
                    "channels": 32,
                    "points_per_second": 1400000,
                    "upper_fov": 10.0,
                    "lower_fov": -30.0,
                    "rotation_frequency": 20,
                    "horizontal_fov":360
                }
            ]
        }
    ]
}
```

En el Listado 3.1 se puede observar cómo en primer lugar se debe crear un objeto, que será un vehículo, para lo cual se establece su identificador como `ego_vehicle`. Tras esto, se ha escogido `vehicle.lincoln.mkz_2020` como modelo de vehículo y un punto de creación, que se ha escogido teniendo en cuenta que caiga dentro del mapa usando el *script* `spectator_location.py` proporcionado por la API de CARLA, el cual devuelve la localización exacta de un punto en el mapa.

Posteriormente se ha creado un diccionario de sensores, donde se han creado las seis cámaras, estableciendo `sensor.camera.rgb` en el campo `type` y el sensor LiDAR 3D estableciendo `sensor.lidar.ray_cast` en su campo correspondiente. Para el caso de las cámaras RGB, se ha modificado el punto de aparición de estas, que coincide con lo expuesto anteriormente en la Tabla 3.3. Además se ha modificado el tamaño de las imágenes a $[\text{image\_size\_x}, \text{image\_size\_y}] = [1600, 900]$ para que concuerde con los datos de entrada esperados por la red BEVFusion [14] y se ha establecido el FOV (*Field Of View*) a 50^{o} para todas las cámaras menos para la trasera, cuyo valor es 110^{o} [42].

Por otro lado, para el caso del LiDAR 3D se ha establecido como punto de aparición el mostrado en la última columna de la expresión 3.1 y se ha rotado -90^{o} en el eje z para que el LiDAR de nuScenes concuerde con el sistema de referencia de ROS2 (ver Figuras 3.4 y 3.5b). Además, el resto de características como el número de haces, los canales, el FOV se han establecido según lo mostrado en la Tabla 3.1.

3.3. Control del vehículo

Una vez creado el modelo digital del vehículo junto a sus sensores en CARLA, el siguiente paso es controlar dicho vehículo. Para ello se presentan dos opciones, emplear rutas programadas o usar un control manual [10].

Rutas programadas

Este método consiste en definir una secuencia de puntos predefinidos (ver Figura 3.6), usando la herramienta `spawn_point.py` formando una ruta de conducción [10]. Posteriormente se utiliza `traffic_manager.py`, de la API de CARLA, se carga el conjunto de puntos, para cargar y seguirlas rutas, como se muestra en la Figura 3.7. Sin embargo, este método tiene el inconveniente de que CARLA aún no cuenta con una versión estable y la herramienta `traffic_manager` cuenta con múltiples fallos, lo que impide completar rutas sin interrupciones inesperadas. Por este motivo, se optó por el control manual.

Figura 3.6: Ejemplo de puntos de una ruta de conducción [10].

Figura 3.7: Ejemplo de una ruta de conducción en el mapa `Town10HD_Opt` [10].

Control manual

Como método alternativo de teleoperación del vehículo, se recurrió un periférico externo actuando sobre el simulador mediante CARLA-ROS-Bridge. Para esto, la API de CARLA nuevamente proporciona dos herramientas que permiten controlar el vehículo: `manual_control.py` y `manual_control_steeringwheel.py`. Ambos utilizan la librería PyGame de Python para abrir una interfaz con la que teleoperar el vehículo, con la diferencia de que en la primera de las herramientas el vehículo se controla mediante el teclado, mientras que, en la segunda de ellas, el vehículo se controla con un juego de volante y pedales Logitech G29. De las dos formas de control, se usó el volante dado que el control es mucho más sencillo que con teclado. Sin embargo, la herramienta `manual_control_steeringwheel.py` viene originalmente programada para generar un vehículo aleatorio en CARLA y mostrar la interfaz de control manual. Como no ese no es el objetivo, se modificó dicha herramienta con el nombre `manual_lincoln_steering_and_traffic.py`, que busca el vehículo previamente creado en el simulador y se le asigna la interfaz de control. De este modo, los controles para la teleoperación del vehículo se recocogen en la Tabla 3.4, mientras que en la Figura 3.8 se muestra la interfaz de teleoperación del vehículo.

Control	**Acción**
Dirección	Girar el volante
Aceleración	Accionar el pedal derecho
Freno	Accionar el pedal central
Freno de mano	Accionar la maneta trasera derecha del volante
Cambio de marcha	Automático
Marcha atrás	Accionar la maneta trasera izquierda del volante

Tabla 3.4: Controles para la teleoperación del vehículo.

Figura 3.8: Interfaz de teleoperación del vehículo.

3.4. Modificación de las condiciones de la simulación

Como se comentó en el Capítulo 2, una de las principales ventajas que ofrece el simulador CARLA es el alto grado de personalización de las condiciones de la simulación. De entre todas las posibles condiciones que se pueden variar, en este trabajo se ha modificado el mapa, las condiciones atmosféricas y los participantes del tráfico.

3.4.1. Mapas

La versión 0.9.15 de CARLA incluye una gran variedad de mapas para realizar simulaciones, que van desde `Town_01` hasta `Town_15`. Además, desde el mapa `Town_01` hasta `Town_10` existen dos posibles versiones, una con el sufijo `_Opt` en su nombre y otras sin dicho sufijo. Los mapas este sufijo tienen sus elementos estructurados en capas específicas según su función (carreteras y señales, edificios, efectos climáticos, etc), lo que los hace más estables. Cabe destacar que, a partir del mapa `Town_11` en adelante, todos cuentan con estructura multicapa [10].

De entre los mapas disponibles, se decidió elegir uno de los más estables según [10], que es el `Town_15`. Este además destaca por su amplitud y la presencia de numerosos vehículos estáticos, lo cual asegura un mayor abanico de vehículos diferentes para entrenar redes neuronales o comprobar su desempeño. En la Figura 3.9 se muestra una imagen del mapa `Town_15`.

Figura 3.9: Imagen del mapa `Town_15` [15].

3.4.2. Condiciones climáticas

El simulador CARLA permite modificar las condiciones climáticas con el objetivo de probar el desempeño de los sensores bajo diferentes escenarios. Para ello, únicamente es necesario modificar el archivo `CarlaSettings.ini` y modificar la llave `WeatherId` por la correspondiente a cada una de los escenarios [15], que se adjuntan en la Tabla 3.5. Además, en la Figura 3.10 se muestran algunos ejemplos de diferentes condiciones climáticas en el simulador.

Clima	Etapa del día	WeatherId
Despejado	Día	1
	Atardecer	2
Nublado sin lluvia	Día	3
	Atardecer	4
Húmedo sin lluvia	Día	5
	Atardecer	6
Lluvia ligera	Día	7
	Atardecer	8
Lluvia moderada	Día	9
	Atardecer	10
Lluvia intensa	Día	11
	Atardecer	12

Tabla 3.5: Tipos de clima y sus identificadores en distintas etapas del día.

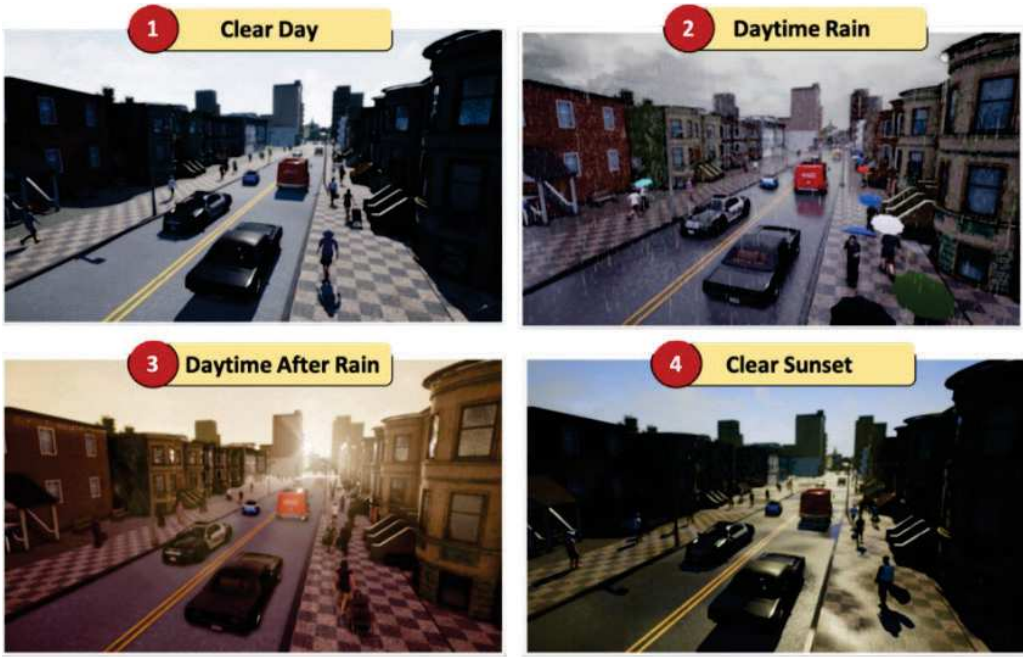


Figura 3.10: Muestra de diferentes condiciones climáticas en CARLA [48].

3.4.3. Participantes del tráfico

Una vez ya se ha creado el vehículo, se ha elegido el mapa y las condiciones meteorológicas, el único paso restante es generar participantes del tráfico. Para ello, nuevamente, se hace uso de la herramienta `traffic_manager.py` de la API de CARLA. Dicha herramienta es una librería que permite crear una cantidad a elegir de peatones y de vehículos, entre los cuales se incluyen coches, motos, bicicletas, camiones, autobuses y trailers. Además, dicha librería controla el comportamiento de los participantes del tráfico en modo piloto automático, creando condiciones de tráfico realista durante la simulación.

En cuanto a la personalización de la herramienta, no solo permite modificar la cantidad de participantes del tráfico, si no que además permite personalizar el comportamiento de los vehículos a nivel de velocidades, ignorar señales o cambiar de carril. Finalmente, la librería `traffic_manager.py` tiene ciertas funciones a destacar, que se exponen a continuación [15].

Modo determinista

Este modo permite guardar las posiciones iniciales y las rutas de cada uno de los participantes del tráfico para romper la aleatoriedad de sus rutas en el caso de que se quieran reproducir los resultados con posterioridad.

Modo híbrido de físicas

Este modo permite desactivar la mayoría de los cálculos de física para vehículos en piloto automático cuando están fuera de un radio específico del vehículo principal sensorizado. Esto permite eliminar el cuello de botella de cálculos de física en la simulación, ya que los vehículos se mueven por cambios en su posición en lugar de física real. Sin embargo, se conservan cálculos básicos de aceleración lineal para asegurar que las velocidades se mantengan de forma realista.

Capítulo 4

Detección multi-modal de participantes del tráfico con RNA.

Contenido

4.1. Introducción

Los sistemas de conducción autónoma incorporan una amplia gama de sensores para capturar datos del entorno, de entre los cuales en este proyecto se utilizan cámaras RGB y LiDAR 3D, cada uno aportando información específica. Las cámaras, por un lado, capturan información semántica, mientras que el LiDAR 3D proporciona información espacial [14]. Para estos sensores existen actualmente diversos métodos de detección de objetos, como las redes YOLO (*You Only Look Once*) [49] o las DETR (*Detection Transformers*) [50] aplicadas a imágenes RGB, mientras que para el caso del LiDAR 3D destacan las redes PV-RCNN (*PointVoxel Regional based Convolutional Neural Network*) [51] y su evolución Part-A^2-Net [52]. Sin embargo, recientes investigaciones [10] muestran que las detecciones por separado en cada uno de estos sensores muestran limitaciones. Por un lado, las detecciones en imágenes RGB carecen de información espacial, es decir, de distancias, mientras que las detecciones en nubes de puntos cuentan con esta característica pero no detectan correctamente los peatones debido a que estos, al ocupar menos espacio que un vehículo, tienen asociados menos puntos del LiDAR 3D. Por tanto, en este con-

texto, la combinación de la información proporcionada por ambas fuentes es crucial para lograr detecciones fiables y detalladas superando las limitaciones individuales de cada uno de estos sensores.

En la actualidad, la mayoría de los métodos de fusión multi-modal entre cámaras-LiDAR se basan en la proyección de características de las cámaras al LiDAR o viceversa. Sin embargo, estos métodos presentan varias limitaciones. Por un lado, en el caso de la proyección del LiDAR 3D a las cámaras, la información se distorsiona geométricamente, lo que la hace poco efectiva para tareas de detección 3D. Por otro lado, en el caso opuesto, al proyectar de las cámaras al LiDAR 3D existe una pérdida de información semántica ya que, solo una pequeña proporción de los datos de la cámara (en torno al 5 % para un LiDAR de 32 haces) coincide con los puntos de LiDAR, mientras que el resto de datos se omiten. Además, cabe destacar que este problema de densidad aumenta conforme menor es la resolución del LiDAR [14].

En este contexto, se presenta el modelo BEVFusion, que permite unificar características multi-modales procedentes de las cámaras RGB y el LiDAR en el espacio BEV (*Bird's Eye View*), una representación del entorno que muestra el plano desde una perspectiva aérea. El uso de este espacio permite conservar tanto la estructura geométrica como la densidad semántica, lo que lo hace especialmente adecuado para tareas de detección 3D.

4.2. Arquitectura del modelo BEVFusion

El modelo BEVFusion cuenta con una arquitectura que se basa en extraer características por separado, post-procesarlas y extraer nuevamente las características de los datos post-procesados. En la Figura 4.1 se muestra la arquitectura del modelo BEVFusion.

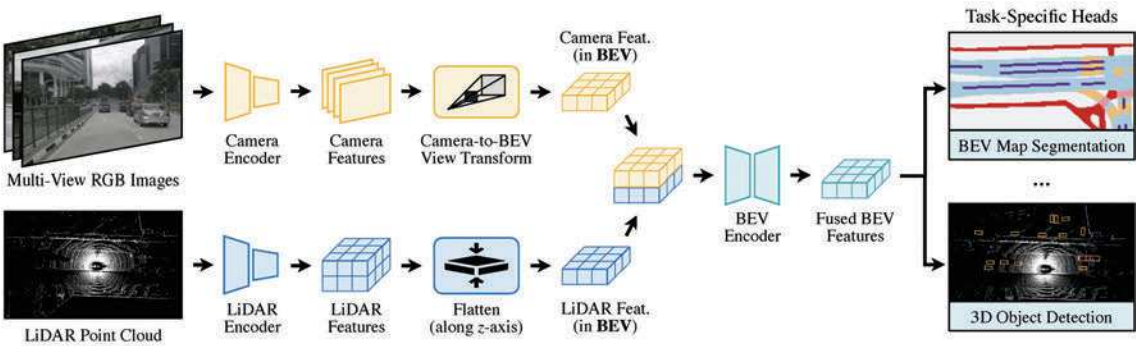


Figura 4.1: Arquitectura del modelo BEVFusion [14].

4.2.1. Extracción de las características

El primer paso que se sigue en el modelo BEVFusion es la obtención de las características de los datos de entrada. Por un lado, para extraer las características

de las nubes de puntos del LiDAR se emplea la arquitectura VoxelNet [53], con lo cual se obtiene información espacial sobre los objetos. Por otro lado, para el caso de las imágenes RGB se presentan dos opciones, con las cuales se obtiene información semántica de los objetos. Se puede usar o bien Swin Transformer [54] o ResNet-50 [55], siendo esta última la elegida debido a su mayor optimización para trabajar con TensorRT.

4.2.2. Proyección al espacio de vista de pájaro BEV

Una vez se han obtenido las características de las nubes de puntos y las imágenes RGB, es necesario proyectarlo al espacio BEV. En primer lugar, como en el caso de las nubes de puntos la información que se obtiene es tridimensional, para realizar la transformación se proyectan todas sus características sobre un plano perpendicular al eje z de referencia, lo cual evita la distorsión geométrica.

En cuanto a las imágenes RGB, su transformación es más complicada debido a que se desconoce de forma directa la profundidad de cada píxel de la imagen. Para ello, se recurre a la técnica LSS (*Lift-Splat-Shoot*) [56], que se basa en calcular una distribución de probabilidad de profundidad para cada píxel de la imagen. Con esto se consigue que, en lugar de asignar una única profundidad, se estima un conjunto de profundidades posibles para cada píxel. Tras calcular esta distribución, cada píxel se proyecta a múltiples puntos a lo largo del rayo de la cámara en distintas profundidades y sus características se ponderan según la probabilidad de cada valor de profundidad. Una vez se ha realizado este proceso, se cuantiza la nube de puntos sobre los ejes x e y, se agrupan todas las características en una cuadrícula y se proyectan sobre el plano perpendicular al eje z con la operación *BEV pooling* [57, 14]. Sin embargo, este proceso es muy costoso computacionalmente dado que la nube de puntos contiene una cantidad de información dos órdenes de magnitud mayor que las generadas con un LiDAR, con lo que es necesario optimizarla con precomputación.

4.2.3. Fusión de las características

Una vez todas las características de los sensores se encuentran transformadas al espacio compartido BEV, pueden fusionarse fácilmente mediante operadores como la concatenación. Sin embargo, aunque están en el mismo espacio, las características en el espacio BEV de ambos sensores pueden desalinearse ligeramente debido a imprecisiones en la estimación de la profundidad. Para solucionar esto, se emplea un codificador basado en convoluciones que compensa este efecto [14]. Finalmente, para cada una de las tareas que se realizan en la percepción se asocia un proceso específico, denominado *head*. Entre estos procesos se incluye el cálculo de la ubicación central de los objetos o la estimación del tamaño, rotación y velocidad de los objetos.

4.3. BEVFusion-ROS-TensorRT

Para aplicar el modelo BEVFusion a los datos sensoriales publicados con ROS2, se hizo uso del repositorio BEVFusion-ROS-TensorRT [23] que ofrece dicho modelo con aceleración empleando TensorRT.

A nivel interno, cuenta con un archivo `bevfusion.launch.py` de ROS2 en el cual se define el modelo de red a utilizar entre ResNet-50 y Swin Transformer. Tras esto, en el nodo principal de procesamiento, denominado `bevfusion_node.py`, se carga el modelo optimizado, se reciben los datos de los sensores de entrada y se procesan en línea con TensorRT para mejorar la velocidad. Los tópicos empleados se muestran en la Tabla 4.1.

Tipo	**Tópico**	**Variable**
Suscriptor	/carla/ego_vehicle/Front_Central_RGB_1/image	`sub_img_f`
	/carla/ego_vehicle/Front_Left_RGB_1/image	`sub_img_fl`
	/carla/ego_vehicle/Front_Right_RGB_1/image	`sub_img_fr`
	/carla/ego_vehicle/Back_Central_RGB_1/image	`sub_img_b`
	/carla/ego_vehicle/Back_Left_RGB_1/image	`sub_img_bl`
	/carla/ego_vehicle/Back_Right_RGB_1/image	`sub_img_br`
	/carla/ego_vehicle/LIDAR	`sub_cloud`
Publicador	/result_image	`pub_img_`

Tabla 4.1: Tópicos empleados en el repositorio BEVFusion-ROS-TensorRT.

Por un lado, los datos se cargan mediante los tópicos suscriptores asociados a las variables `sub_image_f_`, `sub_image_fl_`, `sub_image_fr_`, `sub_image_b_`, `sub_image_bl_`, `sub_image_br_`, `sub_cloud_`, que corresponden respectivamente con la cámara frontal central, cámara frontal izquierda, cámara frontal derecha, cámara trasera central, cámara trasera izquierda, cámara trasera derecha y nube de puntos. Por otro lado, los resultados se publican en forma de imagen en el tópico `/result_image`. Adicionalmente, este repositorio cuenta con el directorio `scripts`, que incluye herramientas para preprocesar y compilar el modelo en TensorRT, optimizando su rendimiento. Además, cuenta con otro directorio, denominado `bevfusion`, en el que está presente el repositorio original de BEVFusion, el cual debe usarse para entrenar la red neuronal.

Capítulo 5

Resultados

Contenido

5.1. Introducción

En el presente capítulo se presenta, por un lado, los resultados del procesamiento de las imágenes y las nubes de puntos en CARLA usando el modelo BEVFusion original, entrenado con el conjunto de datos de nuScenes. Por otro lado, posteriormente se presentan las métricas y resultados del entrenamiento de la red neuronal con el conjunto de datos `AdverseOp3D Dataset`, con datos del simulador CARLA.

5.2. Resultados del modelo BEVFusion entrenado con el *dataset* nuScenes

Para este primer caso, se probó directamente a utilizar el modelo BEVFusion entrenado por Mit-Han-Lab con el conjunto de datos nuScenes, empleando el archivo

de pesos proporcionado en [22]. Cabe destacar que dicho archivo de pesos ya entrenado se obtuvo entrenando con la red `Swin-Transformer` con una configuración de cuatro muestras por GPU, un máximo de cuatro procesos en paralelo para cargar los datos en la GPU, una tasa de aprendizaje de $2 * 10^{-4}$ y un total de 20 épocas.

5.2.1. Métricas

Las métricas que se van a analizar han sido obtenidas con el archivo de pesos original de BEVFusion y analizado sobre la versión `v1.0-mini` de nuScenes, se muestran en la Tabla 5.1. En la Tabla 5.1 se pueden observar los resultados del modelo BEVFusion en diferentes clases de objetos, evaluados con métricas como la precisión promedio (AP), el error de traslación (ATE), el error de escala (ASE), el error de orientación (AOE), el error de velocidad (AVE) y el error de atributos (AAE).

Objeto	**AP**	**ATE**	**ASE**	**AOE**	**AVE**	**AAE**
Coches	0.854	0.167	0.163	0.115	0.108	0.073
Camiones	0.823	0.138	0.133	0.056	0.062	0.005
Autobuses	0.997	0.180	0.096	0.022	0.567	0.284
Trailers	0.000	1.000	1.000	1.000	1.000	1.000
Vehículos de construcción	0.000	1.000	1.000	1.000	1.000	1.000
Peatones	0.926	0.120	0.243	0.300	0.211	0.117
Motocicletas	0.629	0.195	0.268	0.385	0.052	0.000
Bicicletas	0.556	0.167	0.190	0.239	0.429	0.000
Señales de tráfico	0.529	0.078	0.350	–	–	–
Barreras	0.000	1.000	1.000	1.000	–	–
Valores medios	0.759	0.149	0.206	0.186	0.238	0.079

Tabla 5.1: Métricas del modelo BEVFusion con la versión `v1.0-mini` del conjunto de datos nuScenes. Subconjunto de validación (`val`).

En términos generales, se puede observar una detección muy precisa en las clases de coches, camiones, autobuses y peatones, con altos valores de AP, lo que indica que el modelo ha aprendido a identificar estos objetos en el espacio. Sin embargo, las detecciones son mucho más imprecisas en clases como las motocicletas, las bicicletas y las señales de tráfico, con un valor de AP en torno a 0.5. Sin embargo, en clases como trailers, vehículos de construcción y barreras la precisión es nula, con un AP de 0 debido a que originalmente el modelo BEVFusion entrenado con la versión `v1.0-trainval` (conjunto de datos completo) de nuScenes cuenta con estas clases. Sin embargo, en este trabajo, como dicha versión del conjunto de datos ocupa cerca de 300 GB, se utilizó una versión más ligera, denominada `v1.0-mini`, en la cual no hay muestras de trailer, vehículos de construcción y barreras.

Continuando con el análisis de los resultados, el error de traslación (ATE) mide la capacidad del modelo para localizar espacialmente cada objeto y muestra que las

clases señales de tráficos y peatón se localizan con una alta precisión, mostrando valores de ATE bajos. Las clases coche, camión y bicicleta, en cambio, muestran un ATE que, aunque no es tan preciso como en las clases anteriores, sigue siendo aceptable para tareas de detección de objetos. Finalmente, las clases que no están presentes en la versión `v1.0-mini` del conjunto de datos, presentan un error de traslación máximo, lo cuál tiene sentido al no lograrse identificar estos objetos en el espacio.

Respecto al error de escala (ASE), el modelo demuestra una buena capacidad para estimar correctamente el tamaño de algunos objetos como autobuses, camiones y coches. Sin embargo, se observan mayores dificultades para estimar correctamente el tamaño de objetos como señales de tráfico, motocicleta o peatones, donde el error de escala es más elevado. Esto podría deberse a que estos objetos son más pequeños, lo cual hace más complicado el proceso de etiquetado con precisión.

La orientación de los objetos, medida a través del error de orientación (AOE), también refleja, como en el caso anterior, mejores resultados en objetos grandes, mientras que en objetos más pequeños este error aumenta. Esto puede estar debido a que estos objetos están orientados en múltiples direcciones y no siempre presentan una trayectoria claramente definida (por ejemplo los peatones pueden cambiar de dirección drásticamente o quedarse quietos). Por último, este valor no aparece en las señales de tráfico ya que el AOE no se define en modelos estáticos.

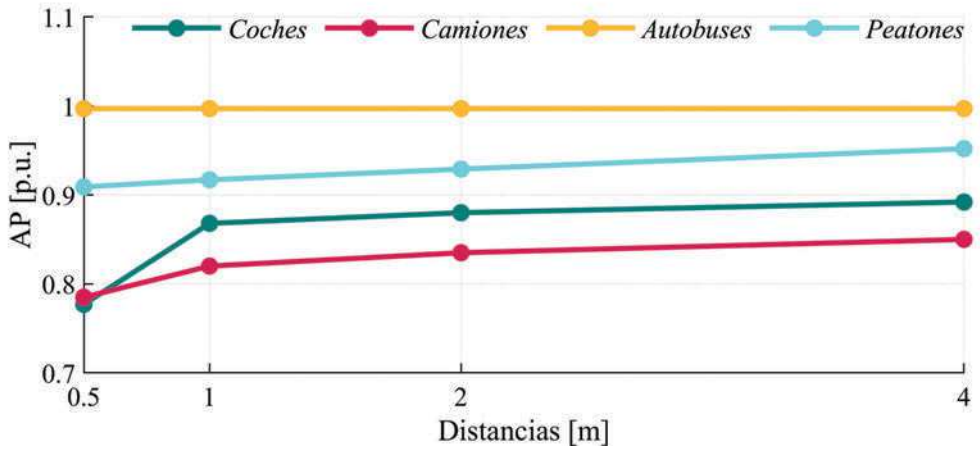
En cuanto a la evaluación de la velocidad, medida con el error de velocidad promedio (AVE), muestra en general buenos resultados, destacando la estimación de la velocidad en vehículos más rápidos como los coches, camiones y motocicletas. Mientras que este error aumenta conforme menor es la velocidad del objeto, como autobuses, peatones o bicicletas. Al igual que en el caso anterior, para las señales de tráfico y barreras este parámetro no se define porque son objetos estáticos. Finalmente, analizando el error de atributos (AAE), se puede apreciar cómo la precisión en la identificación de características adicionales del objeto (un vehículo se mueve, una bibicleta lleva un ocupante, etc) es mayor en motocicletas, bicicletas y camiones, mientras que en mayor en coches, peatones y autobuses. Esto puede deberse a una mayor variación en las características de estos objetos. Nuevamente, este parámetro no se define para las señales de tráfico y barreras por ser estáticos.

5.2.2. Influencia de la distancia en la precisión media

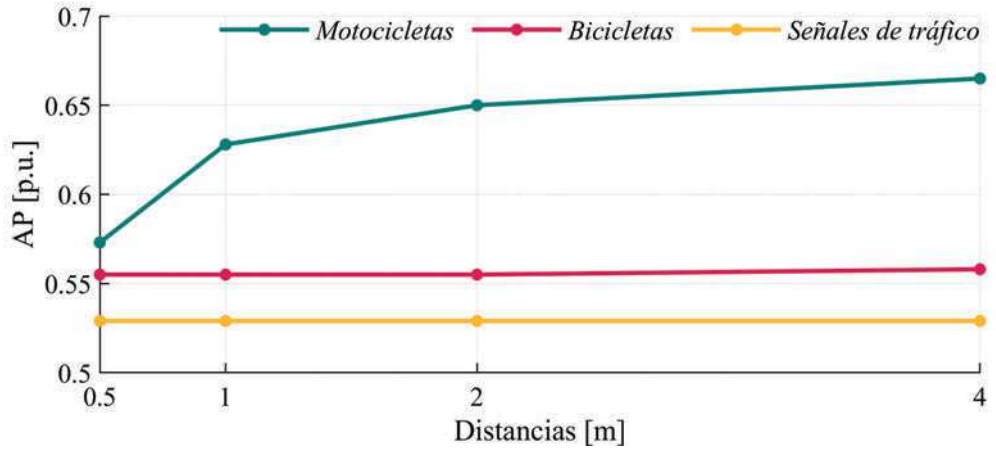
Posteriormente, tras obtener las métricas del modelo BEVFusion con el archivo de pesos entrenado por Mit-Han-Lab, se procedió a realizar un análisis de la influencia de la distancia de los objetos en la precisión media. Los resultados del análisis se muestran el Tabla 5.2, para distancias d menores a 0.5 m, 1.0 m, 2.0 m, 4.0 m. La información de la Tabla 5.2 se pueden representar de forma gráfica como se muestra en la Figura 5.1 que, para mejor visualización, se compone a su vez de dos figuras con las clases repartidas en función del AP.

Objeto	**AP**			
	***d* <0.5 m**	***d* <1.0 m**	***d* <2.0 m**	***d* <4.0 m**
Coches	0.777	0.868	0.880	0.892
Camiones	0.785	0.820	0.835	0.850
Autobuses	0.997	0.997	0.997	0.997
Trailers	0.000	0.000	0.000	0.000
Vehículos de construcción	0.000	0.000	0.000	0.000
Peatones	0.909	0.917	0.929	0.952
Motocicletas	0.573	0.628	0.650	0.665
Bicicletas	0.555	0.555	0.555	0.558
Señales de tráfico	0.529	0.529	0.529	0.529
Barreras	0.000	0.000	0.000	0.000

Tabla 5.2: Métricas de AP a diferentes distancias en el modelo BEVFusion para varias clases de objetos.



(a) Clases: Coches, camiones, autobuses, peatones.



(b) Clases: Motocicletas, bicicletas, señales de tráfico.

Figura 5.1: Evolución del AP frente a la distancia.

Analizando la Figura 5.1, se puede observar cómo la tendencia es que la precisión en las detecciones aumente conforme mayor es la distancia al vehículo, lo cual puede deberse a que conforme más cerca está un objeto, mayor es la pérdida de información espacial sobre el mismo. Además, en algunas clases como los autobuses, las bicicletas o las señales de tráfico no varía el AP con la distancia, lo cual se puede comprobar numéricamente en la Tabla 5.2. La razón de esto es que, en las muestras del conjunto de datos, estos objetos únicamente han aparecido a distancias menores de 0.5 m, con lo que también pertenecen al resto de casos, como distancias menores a 1 m y en adelante.

5.2.3. Resultados en el simulador CARLA

Una vez obtenida las métricas del modelo BEVFusion, se utilizó dicho modelo entrenado en formato `onnx`, resultante de aplicar aceleración mediante TensorRT al modelo original. Para obtener dicho fichero se recurrió directamente al ofrecido en el repositorio del paquete `BEVFusion-ROS-TensorRT` [23]. Algunos de los resultados más representativos se muestran a continuación, donde el fondo muestra la representación de la nube de puntos del LiDAR 3D y las imágenes RGB se muestran alrededor de esta con su correspondiente etiqueta de identificación. Además, el vehículo se representa con una caja de color verde, los coches con una caja de color naranja, los peatones con una caja de color azul y los camiones, autobuses y bicicletas con una caja de color rojo. Sobre estas cajas se muestra una etiqueta con la clase detectada y la precisión.

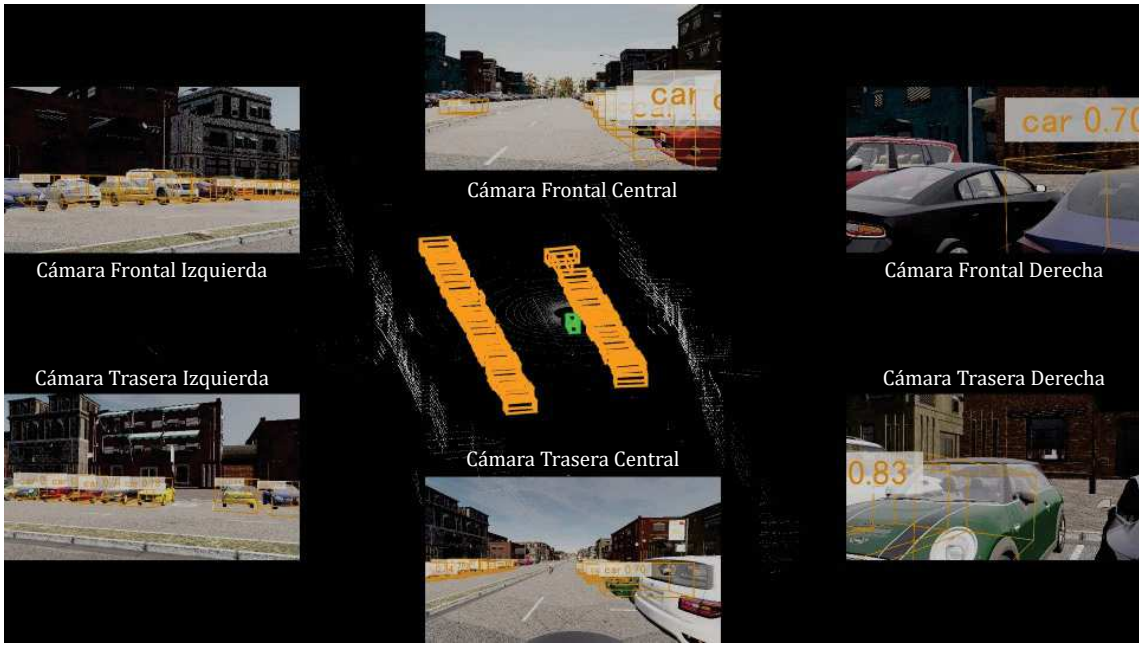


Figura 5.2: Detección de coches.

En la Figura 5.2 se puede observar cómo los coches son detectados correctamente

por la red neuronal, con un AP que varía entre 0.7 y 0.8, lo cual indica la gran precisión de las detecciones. Sin embargo, se puede observar que existe un cierto desajuste entre los coches y las cajas. Esto se debe a que habría que rehacer la calibración intrínseca de los sensores al haber usado las matrices de calibración de las cámaras reales en lugar de las simuladas que, aunque tienen la misma disposición y características, en el simulador estas pueden variar ligeramente.

En la Figura 5.3 se puede observar cómo, en la cámara frontal central aparece un autobús, que es correctamente detectado por la red neuronal con una precisión de 0.49. Sin embargo, la caja que la rodea además de encontrarse desplazada, no corresponde con el tamaño del autobús. La razón de esto es que al no estar correctamente calibrada la cámara, la estimación de la profundidad no es correcta y la altura de la misma varía.

Figura 5.3: Detección de autobuses.

En la Figura 5.4 se puede observar cómo el camión, que se muestra en la cámara frontal central, es detectado de forma incorrecta como un autobús en la Figura 5.4a y como un coche en la Figura 5.4b. La razón de esto puede ser una discrepancia entre el modelo en CARLA del camión y la realidad.

(a) Camión detectado como autobús.

(b) Camión detectado como coche.

Figura 5.4: Detección incorrecta de un camión.

En la Figura 5.5 se puede observar cómo las motos también son detectadas correctamente por la red neuronal (cámara trasera central), con un AP de 0.19, lo cual indica una precisión baja en la detección. Al igual que en el caso anterior, se puede observar cómo hay un desajuste entre la moto y la caja debido a la calibración intrínseca. Por último, se puede observar cómo en la cámara frontal izquierda hay un peatón que no ha sido identificado.

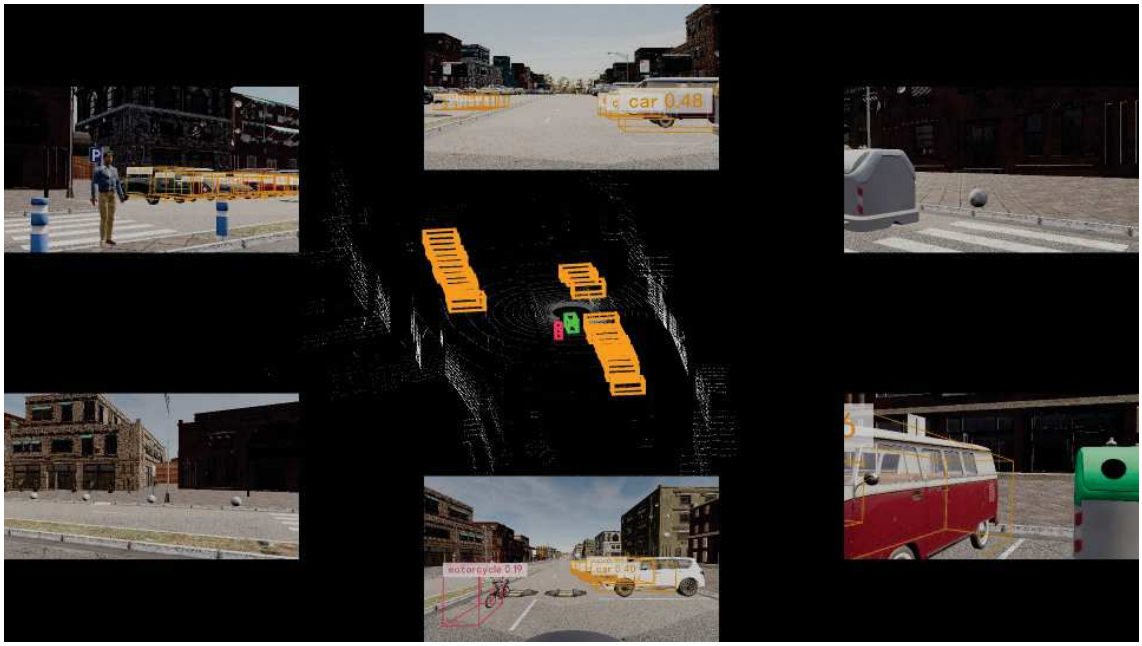


Figura 5.5: Detección de motos.

En la Figura 5.6 se puede observar cómo, nuevamente, aparece un peatón, en este caso en la cámara frontal central, y la red no es capaz de detectarlo. Esto puede deberse a que los peatones tienen un modelo que no es del todo fiel a la realidad (ver Figura 5.7).

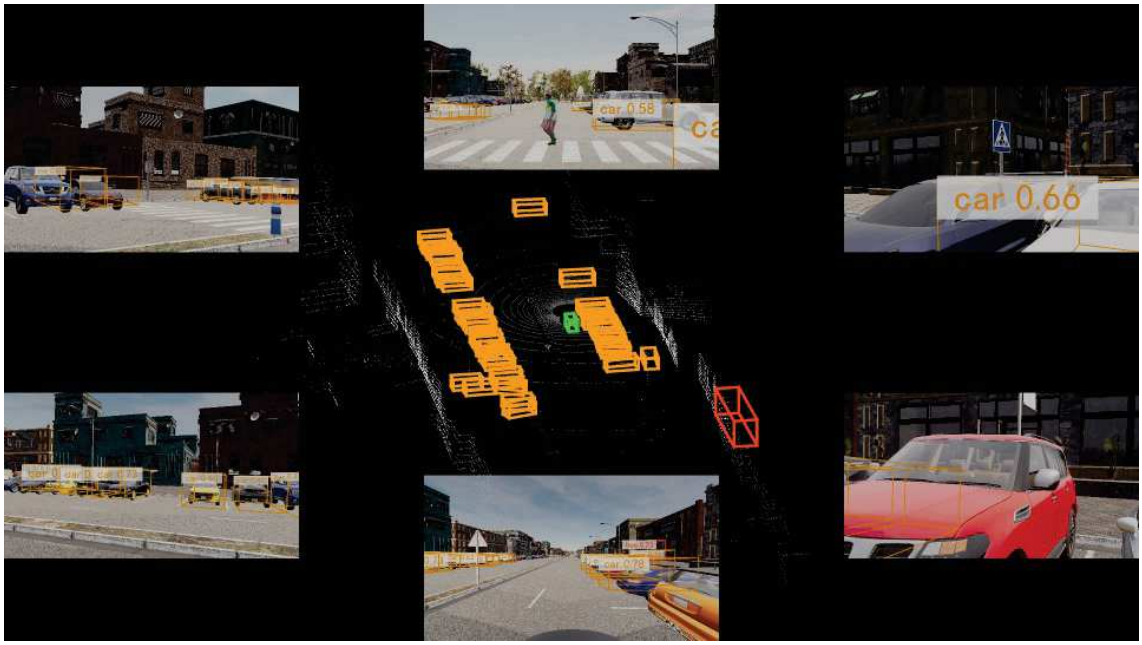


Figura 5.6: Detección incorrecta de peatones.

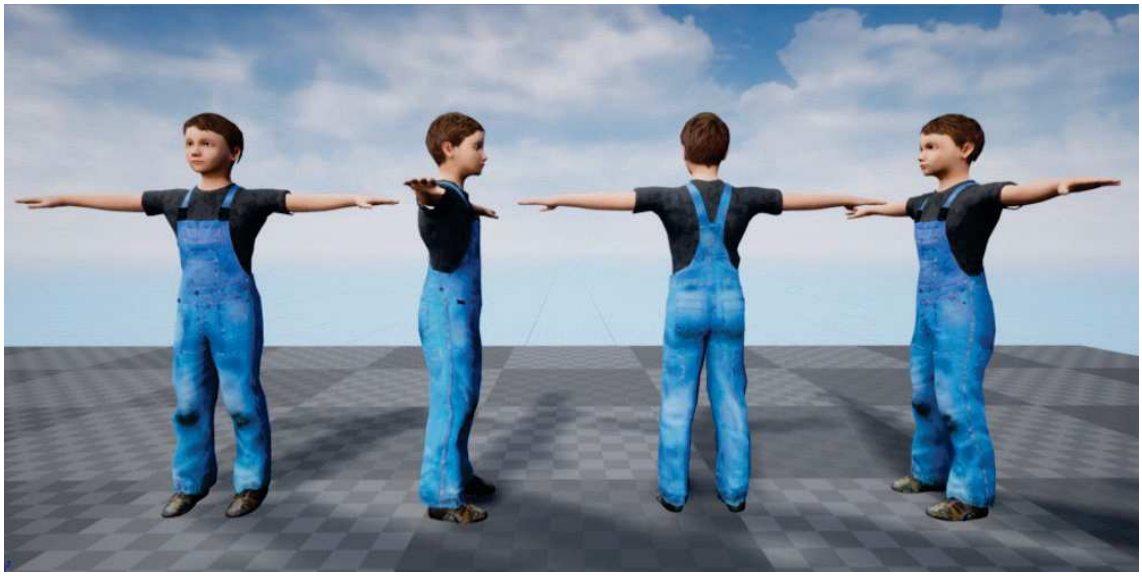


Figura 5.7: Modelo 3D de un peatón en CARLA [15].

A la vista de los resultados, se puede apreciar que el modelo BEVFusion detecta correctamente los coches, las motos y los autobuses con una precisión elevada, especialmente en los coches. Sin embargo, aparece un pequeño desplazamiento de las cajas de detección, lo cual podría ser solucionado con una mejor calirbación de las cámaras. Por otro lado, en cuanto a los camiones, estos han sido confundidos con coches y autobuses, mientras que los peatones no han sido detectados, lo cual puede deberse a la discrepancia del modelo de estos objetos en CARLA respecto a los del mundo real.

Para solventar esto, se decidió reentrenar la red neuronal utilizando un conjunto de datos del simulador CARLA con el formato de nuScenes. Para ello, existe un proyecto denominado ContextualFusion [43] en el cual se reentrenó el modelo BEVFusion en condiciones adversas con un conjunto de datos del simulador CARLA. Por tanto, para reentrenar la red se usó dicho conjunto de datos, denominado `AdverseOp3D Dataset`. Concretamente, dicho conjunto de datos está compuesto por más de 4000 muestras obtenidas tanto de escenas de CARLA como de nuScenes. Sin embargo, de cara a reducir la carga computacional, se redujo el número de escenas del conjunto de datos. Se asignaron para entrenamiento las escenas `scene-0001` (condiciones diurnas) y `scene-0002` (condiciones nocturnas), para validación las escenas `scene-0015` (condiciones diurnas) y `scene-0016` (condiciones nocturnas) y para pruebas (*test*) la escena `scene-0061`. Con ello, el número total de muestras para entrenamiento son 1654, para validación son 416 y para pruebas son 39. Por tanto, de forma proporcional, un 78.43 % de las muestras son de entrenamiento, un 19.70 % son de validación y un 1.87 % son de prueba. Cabe destacar que no fue necesario realizar cambios en el modelo del vehículo de CARLA ya que, el conjunto de datos `AdverseOp3D Dataset`, al igual que este proyecto, utiliza una configuración sensorial idéntica a la original de nuScenes.

5.3. Resultados del modelo BEVFusion entrenado con el *dataset* AdverseOp3D

En esta sección se decidió reentrenar el modelo BEVFusion con el conjunto de datos `AdverseOp3D` para mejorar los resultados mostrados anteriormente. Este *dataset* contiene muestras tomadas del simulador CARLA, con lo que se puede resolver el problema de la discrepancia entre los modelos de peatones y camiones en CARLA frente a los reales. Para este conjunto de datos, el número de clases a entrenar se redujo a coches, camiones, bicicletas y peatones dado que al incluir alguna más, aparecía un error de CUDA relativo a falta de memoria en la GPU a pesar de haber establecido el número de muestras por GPU al mínimo (`samples_per_gpu = 1`).

5.3.1. Métricas

Tras entrenar el modelo BEVFusion con el conjunto de datos `AdverseOp3D`, en la Tabla 5.3 se muestran los resultados con las mismas métricas analizadas anteriormente.

Objeto	**AP**	**ATE**	**ASE**	**AOE**	**AVE**	**AAE**
Coches	0.937	0.085	0.121	0.170	1.000	1.000
Camiones	0.901	0.194	0.172	0.929	1.000	1.000
Autobuses	0.000	1.000	1.000	1.000	1.000	1.000
Trailers	0.000	1.000	1.000	1.000	1.000	1.000
Vehículos de construcción	0.000	1.000	1.000	1.000	1.000	1.000
Peatones	0.490	0.124	0.277	0.946	1.000	1.000
Motocicletas	0.000	1.000	1.000	1.000	1.000	1.000
Bicicletas	0.885	0.093	0.168	0.360	1.000	1.000
Señales de tráfico	0.000	1.000	1.000	–	–	–
Barreras	0.000	1.000	1.000	1.000	–	–
Valores medios	0.803	0.124	0.185	0.601	1.000	1.000

Tabla 5.3: Métricas del modelo BEVFusion con el conjunto de datos `AdverseOp3D`. Subconjunto de validación (`val`).

Comparando los resultados del entrenamiento con `AdverseOp3D` frente a nuScenes, se observan varias diferencias en el desempeño del modelo. En primer lugar, la precisión media de las detecciones ha mejorado en este último *dataset* para los coches, camiones y bicicletas. Sin embargo, la precisión en la detección de peatones se ha reducido prácticamente a la mitad, aunque sigue siendo una precisión aceptable. La razón de este efecto se debe a que `AdverseOp3D` cuenta con un menor número de peatones en las muestras, con lo que la red no ha podido aprender todo lo que podría. Por último, el resto de métricas no presentan variaciones significativas salvo

el error medio de atributo (AAE), cuyo valor en todos casos es bastante elevado, lo que indica que la red no ha podido aprender de los atributos del objeto.

5.3.2. Influencia de la distancia en la precisión media

Al igual que en la sección anterior, se ha realizado un análisis de la influencia de la distancia de los objetos en la precisión media. Los resultados del análisis se muestran el Tabla 5.4, para distancias d menores a 0.5 m, 1.0 m, 2.0 m, 4.0 m. La información de la Tabla 5.4 se pueden representar de forma gráfica como se muestra en la Figura 5.8.

Objeto	**AP**			
	d <0.5 m	**d <1.0 m**	**d <2.0 m**	**d <4.0 m**
Coches	0.9027	0.9371	0.9529	0.9548
Camiones	0.8442	0.8442	0.9468	0.9723
Autobuses	0.0000	0.0000	0.0000	0.0000
Trailers	0.0000	0.0000	0.0000	0.0000
Vehículos de construcción	0.0000	0.0000	0.0000	0.0000
Peatones	0.4895	0.4895	0.4895	0.4895
Motocicletas	0.0000	0.0000	0.0000	0.0000
Bicicletas	0.8681	0.8814	0.8918	0.8994
Señales de tráfico	0.0000	0.0000	0.0000	0.0000
Barreras	0.0000	0.0000	0.0000	0.0000

Tabla 5.4: Métricas de AP a diferentes distancias en el modelo BEVFusion para varias clases de objetos con `AdverseOp3D`.

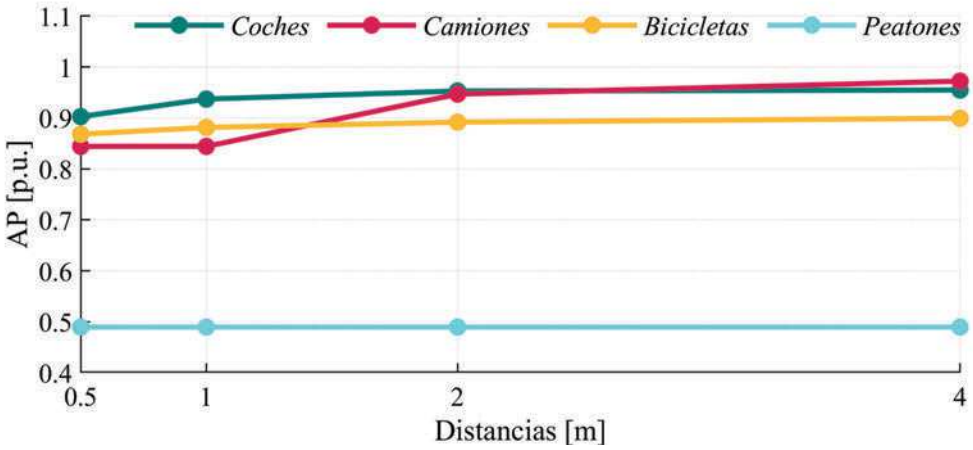


Figura 5.8: Evolución del AP frente a la distancia en `AdverseOp3D`.

Analizando la Figura 5.8, se puede observar, al igual que en el *dataset* de nuScenes que la precisión de las detecciones tiende a aumentar conforme mayor es la distancia

al vehículo. Cabe destacar que el AP de los peatones se mantiene constante ya que no se ha detectado ningún otro peatón a una distancia mayor a 0.5 m.

5.3.3. Resultados en el simulador CARLA

Tras haber entrenado el modelo BEVFusion, el siguiente paso es comprobar gráficamente las detecciones en el simulador CARLA. En este caso, hay que obtener el fichero `onnx` procesado a partir del fichero de pesos entrenado. Sin embargo, esto no ha sido posible debido a que el *script* proporcionado para realizar el procesamiento en TensorRT [23] no presenta un buen funcionamiento, proporcionando resultados totalmente erróneos. Por tanto, debido a esta limitación, en este apartado se proporcionarán como resultados las visualizaciones de la inferencia de la red sobre el subconjunto de validación (`val`) de `AdverseOp3D`. Con ello, algunos de los resultados más representativos se muestran a continuación.

En la Figura 5.9 se puede observar cómo la red detecta correctamente coches (caja naranja), camiones (caja roja) y peatones (caja azul). La precisión en las detecciones es bastante elevada para los coches, con un AP cercano a 0.7, mientras que para el peatón y el camión, la precisión cae en torno a 0.3. Sin embargo, como este *dataset* ha sido obtenido directamente de CARLA, ya se conocen con exactitud todas los parámetros de los sensores, con lo que las matrices de calibración son más precisas y las cajas se encuentran alineadas.

En la Figura 5.10 se pueden observar detecciones sobre coches, sin cambios significativos respecto a la Figura 5.9 y sobre un camión, el cuál presente una métrica no muy elevada, con AP=0.45. Además, se observa un pequeño descuadre en la caja sobre el mismo conforma más cerca se encuentra del vehículo sensorizado.

Figura 5.9: Detección de coches, camiones y peatones en `AdverseOp3D`.

Figura 5.10: Detección de coches y camiones en `AdverseOp3D`.

En la Figura 5.11, 5.12 y 5.13 se puede apreciar cómo los coches se detectan con una precisión similar a las anteriores muestras. Por otro lado, se pueden observar detecciones sobre bicicletas (cajas rojas), con una precisión media en torno a 0.3, aunque no se detectan aquellas con un peatón encima. Cabe destacar que las cajas se encuentran desplazadas en altura, lo cual puede deberse a un etiquetado impreciso, teniendo en cuenta que los sensores ya se encuentran calibrados correctamente.

A la vista de los resultados se puede observar que, tras el reentrenamiento del modelo BEVFusion, los coches siguen detectándose correctamente con una precisión similar al caso anterior. Por otro lado, en cuanto a los camiones y los peatones, se puede observar cómo ahora se detectan sin errores con una precisión media no superior a 0.4. Por último, en cuanto a las bicicletas, no se detectan aquellas con peatones encima. Además, las cajas de detección de estas últimas aparecen desviadas, lo cual podría deberse a un etiquetado impreciso de las escenas.

Figura 5.11: Detección de coches y bicis (i) en `AdverseOp3D`.

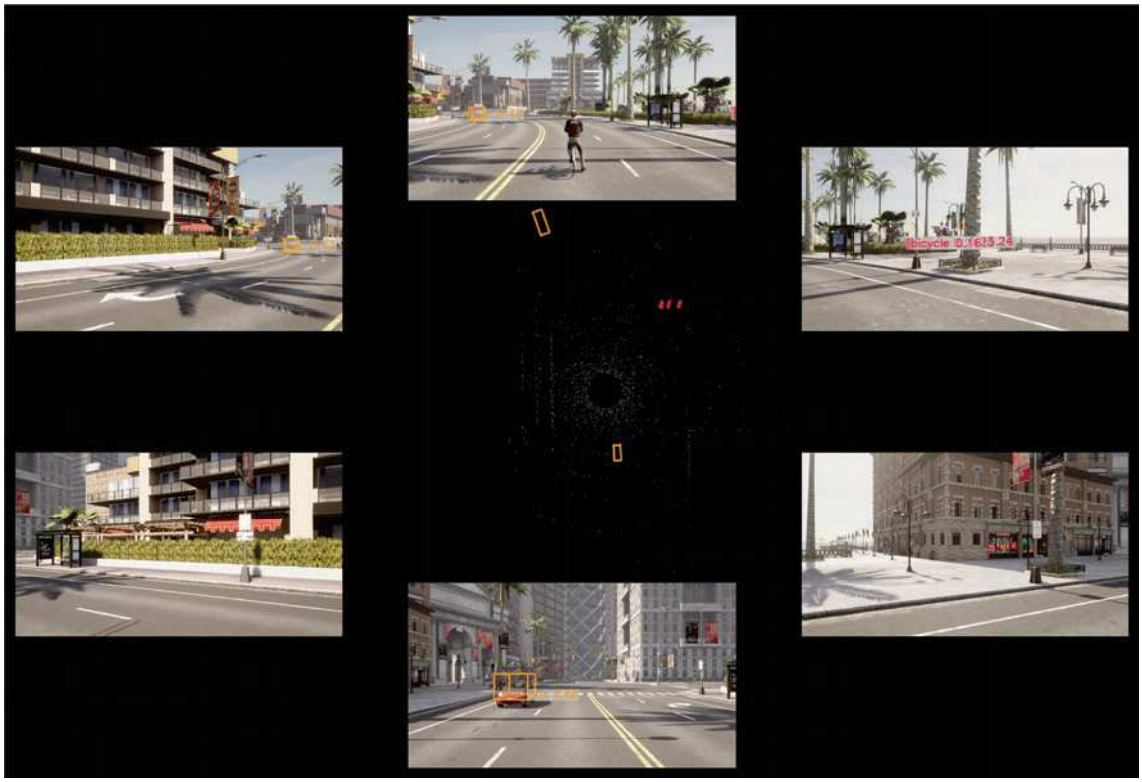


Figura 5.12: Detección de coches y bicis (ii) en `AdverseOp3D`.

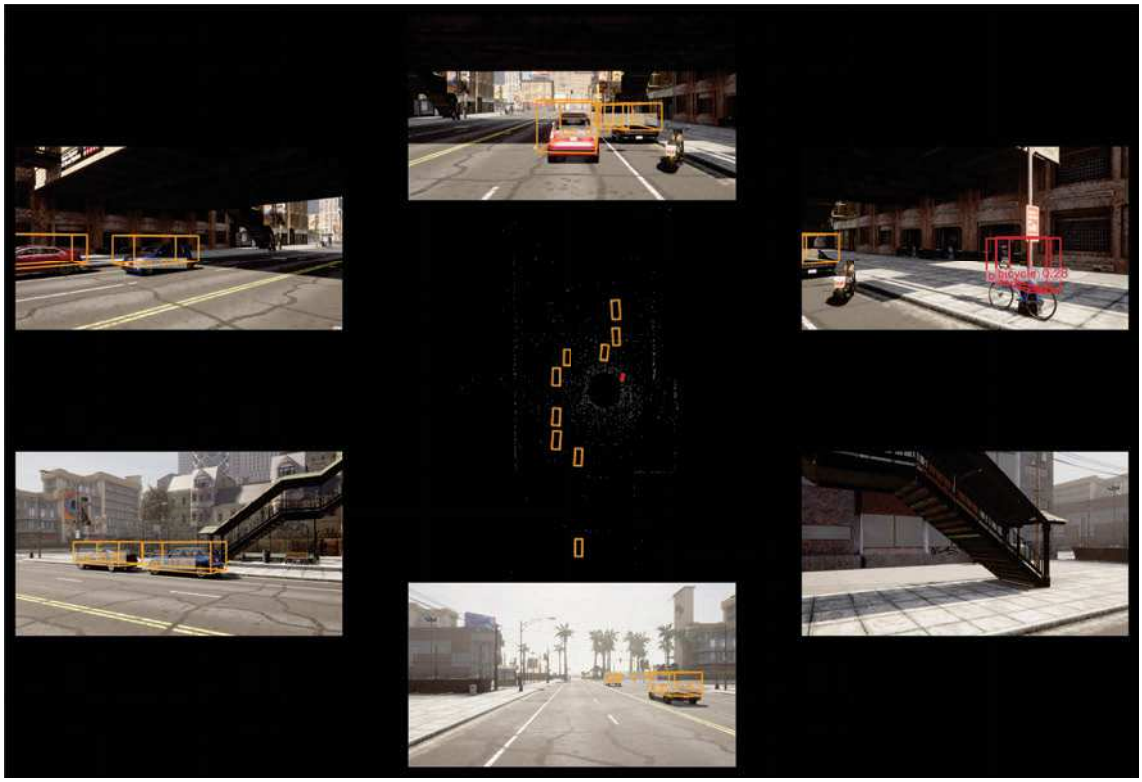


Figura 5.13: Detección de coches y bicis (iii) en `AdverseOp3D`.

Capítulo 6

Conclusiones y líneas futuras

Contenido

6.1. Conclusiones

En este trabajo se ha realizado una fusión multi-modal de las detecciones provenientes de un sensor LiDAR 3D y cámaras RGB con el objetivo de aplicarlo al proyecto PREMOVE. Para ello, se ha partido del Trabajo de Fin de Grado predecesor [10] y se ha hecho uso del simulador CARLA junto al modelo BEVFusion, empleando como puente ROS2, el cuál ha facilitado el trabajo de recopilación de los datos de los sensores.

En primer lugar, se ha estudiado la red neuronal BEVFusion para realizar la fusión multi-modal entre LiDAR y cámaras. Tras esto, se ha puesto en marcha en ROS2 empleando la herramienta BEVFusion-ROS-TensorRT, para lo cual se ha seguido la guía del Anexo A. En segundo lugar, se ha probado dicha red con datos del simulador CARLA, proporcionando unos resultados correctos, con la salvedad de que algunas cajas de detección se encontraban descuadradas y que no se detectaban correctamente ni los peatones ni algunos camiones. Además, se ha realizado un análisis de la influencia de la distancia en la precisión de las detecciones mostrando que, conforme mayor es la distancia de los participantes del tráfico respecto al vehículo sensorizado, mejor es son las detecciones

En tercer lugar, para solventar estos problemas, se ha optado por reentrenar la red neuronal BEVFusion con el conjunto de datos AdverseOp3D, con muestras tomadas del simulador CARLA. Tras reentrenar la red, con un número reducido de clases por problemas de memoria de la estación, se han obtenido métricas y se han analizado de forma cualitativa los resultados gráficos. Por un lado, las métricas han

mostrado unas detecciones medias aceptables con errores algo superiores que con el conjunto de datos de nuScenes. Por otro lado, se han analizado cualitativamente las predicciones sobre la distribución de validación, mostrando unas detecciones algo más imprecisas pero que ya incluyen peatones. Sin embargo, se observó que las bicicletas con un peatón encima no fueron detectadas correctamente y las cajas de detección de las bicicletas y de algunos camiones se encontraban descuadradas, lo cual podía deberse a un etiquetado impreciso de los participantes del tráfico. Por último, volvió a realizarse el análisis de la influencia de la distancia en la precisión de las detecciones, llegándose a las misma conclusiones que en el análisis realizado con el conjunto de datos nuScenes.

6.2. Líneas futuras

Las posibles líneas futuras del presente Trabajo de Fin de Máster se muestran a continuación:

- Utilizar una estación de trabajo para entrenar la red neuronal con todas sus clases.
- Dividir la carga computacional del simulador CARLA y del procesamiento de las redes neuronales en dos equipos diferentes.
- Mejorar la calibración tanto intrínseca como extrínseca de los sensores.
- Mejorar el etiquetado los participantes del tráfico a la hora de entrenar la red neuronal
- Utilizar la configuración sensorial propia del vehículo del proyecto PREMOVE.
- Añadir cámaras térmicas al sistema sensorial para mejorar su robustez en condiciones atmosféricas adversas.
- Desarrollar técnicas de fusión multi-modal para el sistema ampliado con cámaras térmicas.
- Incorporar un mapa del Campus Universitario de Teatinos (Málaga).
- Diseñar un mayor abanico de participantes del tráfico, como peatones en sillas de ruedas, con muletas, etc.

Apéndice A

Manual de instalación de Software

Contenido

A.1. Introducción

En el presente apéndice se recoge la guía con todos los comandos necesarios para la instalación del entorno, todo ello realizado bajo el sistema operativo Ubuntu 22.04. Todo el código desarrollado para este TFM se encuentra en el repositorio correspondiente [58].

A.2. ROS2 RoboStack

A.2.1. Desinstalación de distribuciones anteriores de ROS2

Primero es necesario eliminar todos los paquetes instalados de ROS2 utilizando el comando mostrado en el Listado A.1.

Listado A.1: Comando para eliminar paquetes ROS2

```
sudo apt remove ros-*
```

Tras esto, es recomendable ejecutar el comando mostrado en Listado A.2 para eliminar automáticamente los paquetes que ya no son necesarios.

Listado A.2: Comando para ejecutar autoremove

```
sudo apt autoremove
```

Una vez realizado esto, hay que asegurarse que la distribución de ROS2 ya no esté presente en el directorio `/opt`, eliminando los archivos residuales mediante el comando mostrado en el Listado A.3.

Listado A.3: Comando para eliminar archivos residuales de ROS2 en /opt

```
sudo rm -r /opt/<ros2_distribution>
```

Posteriormente, es necesario eliminar cualquier referencia a ROS2 en su archivo `.bashrc`, los espacios de trabajo de ROS2 que se hayan creado y las fuentes de ROS2 con el comando mostrado en el Listado A.4.

Listado A.4: Comando para eliminar las fuentes de ROS2

```
cd /etc/apt/sources.list.d/
sudo rm ros2.list
```

Finalmente, es necesario ejecutar el comando mostrado en el Listado A.5.

Listado A.5: Comandos finales para limpieza y reinicio

```
sudo apt autoremove
sudo reboot
```

A.2.2. Instalación de Miniforge

En primer lugar, para la instalación de ROS2 Robostack es necesario instalar previamente Miniforge, que es una distribución más ligera del administrador de paquetes y entornos Conda. Para ello es necesario descargar el paquete del repositorio oficial de Github e instalarlo mediante los comandos mostrados en el Listado A.6 para el sistema Linux.

Listado A.6: Comando para descargar e instalar Miniforge

```
curl -L -O "https://github.com/conda-forge/miniforge/releases/
   latest/download/Miniforge3-Linux-x86_64.sh"
bash Miniforge3-Linux-x86_64.sh
```

A.2.3. Instalación de Mamba

Una vez instalado el administrador de paquetes Miniforge, se procede a instalar Mamba, que es a su vez un gestor de paquetes basado en Miniforge que permite trabajar con entornos de datos más grandes y complejos. Para la instalación de Mamba es necesario seguir el comando mostrado en el Listado A.7.

Listado A.7: Comando para instalar Mamba

```
conda install mamba -c conda-forge
```

A.2.4. Instalación del paquete ROS2

Tras la instalación de Mamba, es necesario crear un nuevo entorno con el nombre `ros_humble` utilizando los comandos mostrados en el Listado A.8.

Listado A.8: Comandos para crear y activar un entorno para ROS2

```
cd /path/to/workspace/folder
mamba create -n ros_humble python=3.10
mamba init
mamba activate ros_humble
conda config --env --add channels conda-forge
conda config --env --add channels robostack-staging
conda config --env --remove channels defaults
```

Ahora, es posible instalar ROS2 en el entorno `ros_humble`. Para ello, es necesario utilizar el comando mostrado en el Listado A.9 para instalar el paquete `ros-humble-desktop`.

Listado A.9: Comando para instalar ROS2

```
mamba install ros-humble-desktop pocl
```

Tras instalar ROS2 es necesario comprobar la instalación ejecutando RViz2 con los comandos mostrados en el Listado A.10

Listado A.10: Comandos para desactivar y reactivar el entorno

```
mamba deactivate
mamba activate ros_humble
rviz2
```

Por último, si se desea que el entorno *ros_humble* se active automáticamente por defecto al abrir una terminal, es necesario añadir el comando mostrado en el ListadoA.11 al final del fichero /.bashrc.

Listado A.11: Comando para añadir al fichero .bashrc

```
mamba activate ros_humble
```

A.3. CARLA Simulator

De entre las dos versiones existentes de CARLA Simulator, para este TFM únicamente se hará uso de la versión *package*, dado que no es necesario modificar los escenarios, lo cual es posible con la versión *source*. Si fuera necesario consultar en mayor profundidad dicha versión, la documentación oficial [59] proporciona más detalles.

A.3.1. Descarga del paquete

Primero, es necesario elegir la versión que se desea utilizar y buscarla en el repositorio de CARLA Simulator en *GitHub* [60]. Para la versión 0.9.15, última hasta la fecha, se debe descargar el archivo denominado *CARLA_0.9.15.tar.gz*, como se muestra en la Figura A.1.

Figura A.1: Archivos *.whl* disponibles para la version *package* de CARLA Simulator 0.9.15.

Luego, tras descargar el archivo, es necesario extraer su contenido en un directorio de trabajo conocido.

A.3.2. Instalación de la librería del cliente de CARLA Simulator

De entre los métodos para instalar la librería del cliente de CARLA Simulator, se recomienda, por su facilidad, utilizar los archivos con extensión *.whl*. Para ello, es posible obtener dicho fichero directamente del repositorio *Pypi* [61], como se muestra en la Figura A.2.

Figura A.2: Archivos *.whl* disponibles para CARLA Simulator 0.9.15.

De entre las distintas versiones, hay que elegir la versión *carla-0.9.15-cp310-cp310-manylinux_2_27_x86_64.whl*, dado que la versión de CARLA Simulator es la 0.9.15 y la de Python es la 3.10. Una vez se encuentre descargado el fichero, es necesario ejecutar el comando mostrado en el Listado A.12 en el directorio donde se encuentre el fichero con extensión *whl*.

Listado A.12: Comando para instalar el archivo *.whl*

```
pip3 install carla-0.9.15-cp310-cp310-manylinux_2_27_x86_64.
    whl
```

A.3.3. Instalación de mapas adicionales

La versión *package* de CARLA Simulator ofrece una selección reducida de mapas para la simulación. Para el uso de un mayor abanico de mapas, es necesario descargar el archivo *AdditionalMaps_0.9.15.tar.gz*, mostrado en la Figura A.1.

Posteriormente, es necesario descomprimir el fichero dentro de la carpeta *Import* de la ruta de instalación de CARLA Simulator y ejecutar el archivo *ImportAssets.sh* para finalizar la instalación de los mapas adicionales, empleando para ello el código mostrado en el Listado A.13.

Listado A.13: Comando para instalar el paquete de mapas adicionales

```
cd path/to/CARLA_0.9.15
./ImportAssets.sh
```

A.4. BEVFusion-ROS-TensorRT

Para instalar el *wrapper* de la red neuronal BEVFusion para ROS2, es necesario hacer uso de los comandos mostrados en el Listado **??**

Listado A.14: Comando para instalar el paquete *BEVFusion-ROS-TensorRT*

```
# Ir al espacio de trabajo de ROS2
cd src/
git clone https://github.com/linClubs/BEVFusion-ROS-TensorRT.
   git
cd ..
colcon build --symlink-install --packages-select bevfusion
source install/setup.bash
```

Para hacer uso de este paquete es necesario instalar las librerías que se muestran a continuación.

A.4.1. CUDA 11.3

Para la instalación de CUDA, es necesario acudir a su página oficial [62], donde se debe seleccionar como sistema operativo Linux, arquitectura x86_64, distribución Ubuntu, versión 20.04 e instalador tipo *runfile*. Con ello, los comandos que se deben seguir para la instalación se muestran en el Listado A.15. Cabe destacar que, para evitar problemas con los *drivers* de la gráfica, es necesario desmarcar la opción de instalación de drivers en CUDA.

Listado A.15: Comandos a seguir para la instalación de CUDA 11.3

```
wget https://developer.download.nvidia.com/compute/cuda
   /11.3.0/local_installers/cuda_11.3.0_515.43.04_linux.run
sudo sh cuda_11.3.0_515.43.04_linux.run
```

En el caso de que hiciesen falta varias versiones de CUDA instaladas simultáneamente, se podría emplear la utilidad *switch-cuda* [63].

A.4.2. *PyTorch*

Para la instalación de PyTorch es necesario seguir el comando mostrado en el Listado A.16.

Listado A.16: Comando a seguir para instalar Pytoch en CUDA 11.3

```
pip install pytorch==1.10.1 torchvision==0.11.2 torchaudio
   ==0.10.1 cudatoolkit=11.3 -c pytorch
```

A.4.3. Pillow

Pillow es una biblioteca de procesamiento de imágenes en Python, que surge como alternativa a la antigua biblioteca PIL, y permite realizar tareas como redimensionar, recortar, aplicar filtros y convertir imágenes entre formatos. Para su instalación es necesario seguir el comando mostrado en el Listado A.17.

Listado A.17: Comando a seguir para instalar Pillow

```
pip install Pillow==8.4.0
```

A.4.4. tqdm

Este paquete facilita la creación de barras de progreso en los bucles de Python, mostrando tanto el progreso de procesos como la carga de datos o la ejecución de tareas largas. Para su instalación es necesario seguir el comando mostrado en el Listado A.18.

Listado A.18: Comando a seguir para instalar tqdm

```
pip install tqdm
```

A.4.5. torchpack

Torchpack es un conjunto de herramientas para PyTorch, diseñado para facilitar el entrenamiento y manejo de modelos en proyectos de aprendizaje profundo. Para su instalación es necesario seguir el comando mostrado en el Listado A.19.

Listado A.19: Comando a seguir para instalar torchpack

```
pip install torchpack
```

A.4.6. mmcv

MMCV es una biblioteca para visión por computador que proporciona herramientas de preprocesamiento, manejo de configuraciones y evaluación. Se utiliza en

combinación con otras bibliotecas como PyTorch para proyectos de aprendizaje profundo. Para su instalación es necesario seguir el comando mostrado en el Listado A.20.

Listado A.20: Comando a seguir para instalar mmcv

```
pip install mmcv==1.4.0 -f https://download.openmmlab.com/mmcv
   /dist/cu113/torch1.10.1/index.html
```

A.4.7. mmdet

Mmdet forma parte del conjunto de herramientas de MMDetection, una biblioteca basada en PyTorch para la detección de objetos. Ofrece implementaciones para algoritmos de detección de objetos, entrenamiento y evaluación. Para su instalación es necesario seguir el comando mostrado en el Listado A.21.

Listado A.21: Comando a seguir para instalar mmdet

```
pip install mmdet==2.20.0
```

A.4.8. mmdet3d

Este paquete es una extensión de la biblioteca MMDetection enfocada en la detección de objetos en 3D. Está diseñada para trabajar con datos 3D, como los de sensores LiDAR y cámaras en aplicaciones de conducción autónoma. Además, implementa algoritmos de detección y segmentación 3D utilizando conjuntos de datos estándar como nuScenes o KITTI. Para su instalación, en primer lugar se debe comprobar que no haya previamente ninguna versión previamente instalada de mmdet3d ejecutando el comando `pip show mmdet3d`. En caso de que haya alguna versión instalado, es necesario desinstalarla con `pip uninstall mmdet3d`. Tras esto, para la instalación de mmdet3d es necesario seguir el comando mostrado en el Listado A.22.

Listado A.22: Comando a seguir para instalar mmdet3d

```
# Ir al espacio de trabajo de ROS2
cd src/BEVFusion-ROS-TensorRT/bevfusion
python setup.py develop
```

Finalmente, se puede comprobar con el comando `pip show mmdet3d` que la versión de mmdet3d sea la 0.0.0.

A.4.9. nuscenes-devkit

nuscenes-devkit es un kit de herramientas de desarrollo diseñado para trabajar con el conjunto de datos de nuScenes, que está diseñado para la conducción autónoma. Se caracteriza por proporcionar *scripts* y utilidades para cargar, visualizar

y manipular los datos del entorno de vehículos autónomos. Para su instalación es necesario seguir el comando mostrado en el Listado A.23.

Listado A.23: Comando a seguir para instalar nuscenes-devkit

```
pip install nuscenes-devkit
```

A.4.10. mpi4py

Este paquete proporciona una interfaz para MPI (*Message Passing Interface*) en Python, permitiendo escribir programas paralelos en sistemas distribuidos. Para su instalación es necesario seguir el comando mostrado en el Listado A.24.

Listado A.24: Comando a seguir para instalar mpi4py

```
pip install mpi4py==3.0.3
```

A.4.11. numba

Numba es un compilador que permite acelerar los cálculos numéricos y científicos mediante la compilación de código Python a código máquina optimizado, útil con el trabajo de matrices o bucles. Para su instalación es necesario seguir el comando mostrado en el Listado A.25.

Listado A.25: Comando a seguir para instalar numba

```
pip install numba==0.48.0
```

A.4.12. setuptools

Setuptools es una biblioteca utilizada para gestionar la creación y distribución de paquetes Python, que permite organizar y desplegar proyectos Python en entornos de producción o PyPi. Para su instalación es necesario seguir el comando mostrado en el Listado A.26.

Listado A.26: Comando a seguir para instalar setuptools

```
pip install setuptools==59.5.0
```

A.4.13. numpy

Numpy es una biblioteca para el cálculo científico en Python que proporciona soporte para arrays multidimensionales y un gran número de funciones matemáticas y operaciones algebraicas para manipular datos numéricos de forma eficiente. Para su instalación es necesario seguir el comando mostrado en el Listado A.27.

Listado A.27: Comando a seguir para instalar numpy

```
pip install numpy==1.19.5
```

A.4.14. mmcv-full

MMCV-full es la versión completa de la biblioteca MMCV, que incluye módulos adicionales respecto a la versión base optimizados con aceleración CUDA para visión por computador. Además, proporciona herramientas más avanzadas y optimizadas para tareas que mayor carga computacionales, como la detección y segmentación de objetos. Para su instalación es necesario seguir el comando mostrado en el Listado A.28.

Listado A.28: Comando a seguir para instalar mmcv-full

```
pip install mmcv-full==1.4.0 -f https://download.openmmlab.com
   /mmcv/dist/cu113/1.10.1/index.html
```

A.4.15. yapf

Este paquete es un formateador automático de código Python que reformatea el código fuente de Python para hacerlo más legible y uniforme. Para su instalación es necesario seguir el comando mostrado en el Listado A.29.

Listado A.29: Comando a seguir para instalar yapf

```
pip install yapf==0.40.1
```

A.5. Paquetes adicionales

A parte de los paquetes que ya vienen instalados por defecto en ROS2, es necesario instalar una serie de paquetes adicionales en el directorio src del entorno de trabajo de ROS2.

A.5.1. *ROS2_Numpy*

Es un paquete que facilita la conversión entre los tipos de datos nativos de ROS2, como son los mensajes de imágenes, nubes de puntos, matrices de transformación, etc. y las estructuras de datos de NumPy. Para su instalación es necesario hacer uso de los comandos mostrados en el Listado A.30

Listado A.30: Comando para instalar el 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
```

A.5.2. *Navigation2*

Este paquete, también conocido como Nav2, se basa en el middleware ROS2 y es de utilidad para la planificación de rutas, la localización de robots en entornos dinámicos y el control de movimientos, permitiendo además, la integración con diversos sensores y la adaptabilidad a diferentes tipos de robots. Para su instalación es necesario hacer uso del comando mostrado en el Listado A.31

Listado A.31: Comando para instalar el paquete *Navigation2*

```
mamba install ros-humble-nav2-common
```

A.5.3. *Cv_ bridge*

Este paquete permite la conversión entre imágenes en formato de ROS2 y las estructuras de datos de OpenCV, una biblioteca utilizada para el procesamiento de imágenes en tiempo real [64]. Para su instalación, es necesario hacer uso de los comandos mostrados en el Listado A.32).

Listado A.32: Comandos para instalar el paquete *cv_ bridge*.

```
# 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 install/setup.bash
```

A.5.4. *Vision_ msgs_ rviz_ plugins* para ROS2 Humble

Este paquete es una extensión para la herramienta *rviz2* de ROS2 Humble que permite visualizar mensajes de la clase *vision_ msgs* entre otros. Esto permite visualizar en *rviz2* las detecciones realizadas por las redes neuronales en las nubes de puntos de LiDAR 3D. Para su instalación, es necesario hacer uso de los comandos mostrados en el Listado A.33).

Listado A.33: 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 ..
colcon build --packages-select vision_msgs_rviz_plugins
source install/setup.bash
```

A.5.5. *Carla-ROS-Bridge*

Esta herramienta bidireccional, que permite conectar el simulador CARLA con ROS2 para así poder interactuar con los vehículos simulados. Además permite acceder a datos de sensores como cámaras y LiDAR3D en tiempo real. Para su instalación es necesario hacer uso de los comandos mostrado en el Listado A.34

Listado A.34: Comando para instalar el paquete *Carla-ROS-Bridge*

```
# Ir al espacio de trabajo de ROS2
cd src/
sudo apt-get install libgl1-mesa-dev
mamba install ros-humble-vision-msgs
git clone --recurse-submodules https://github.com/ttgamage/
   carla-ros-bridge.git
cd ..
colcon build --packages-select carla-ros-bridge
source install/setup.bash
```

A.5.6. *Derived_ Object_ Msgs*

Este paquete es de utilidad para representar objetos detectados por sensores, como vehículos, peatones u obstáculos. Para su instalación es necesario hacer uso del comando mostrado en el Listado A.35

Listado A.35: Comando para instalar el paquete *Derived_ Object_ Msgs*

```
# Ir al espacio de trabajo de ROS2
cd src/
git clone https://github.com/astuff/astuff_sensor_msgs
cd ..
colcon build
source install/setup.bash
```

A.5.7. *Tf_ Transformations*

Este herramienta permite, de forma simple, realizar transformaciones geométricas en 3D entre distintas representaciones de posiciones y orientaciones, como matrices de transformación, cuaterniones y ángulos de Euler. Para su instalación es necesario hacer uso del comando mostrado en el Listado A.36

Listado A.36: Comando para instalar el paquete *Tf_ Transformations*

```
# Ir al espacio de trabajo de ROS2
cd src/
git clone https://github.com/DLu/tf_transformations
cd ..
colcon build
source install/setup.bash
```

Apéndice B

Manual de Usuario

Contenido

En el presente apéndice se recoge la guía con todos los comandos necesarios para el entrenamiento de la red neuronal y para procesar los datos del simulador.

B.1. Entrenamiento de la red neuronal

En primer lugar, para entrenar la red neuronal es necesario descargar el conjunto de datos de nuScenes [44] o un conjunto de datos con un formato equivalente en el directorio `/ros2_ws/src/BEVFusion-ROS-TensorRT/bevfusion/data/nuscenes`. La estructura del directorio debe quedar como se muestra en el siguiente árbol.

```
BEVFusion-ROS-TensorRT
└── bevfusion
    └── data
        └── nuscenes
```

```
├── maps
├── samples
├── sweeps
├── v1.0-test
├── v1.0-trainval
├── nuscenes_database
├── nuscenes_infos_train.pkl
├── nuscenes_infos_val.pkl
├── nuscenes_infos_test.pkl
└── nuscenes_dbinfos_train.pkl
```

Posteriormente, una vez descargado el conjunto de datos es necesario procesar dichos datos empleando el comando mostrado en el Listado B.1.

Listado B.1: Comando para procesar el conjunto de datos

```
python tools/create_data.py nuscenes --root-path ./data/
   nuscenes --out-dir ./data/nuscenes --extra-tag nuscenes
```

Una vez se ha procesado el conjunto de datos ya se puede proceder a entrenar la red neuronal. Para este caso, se ha omitido la segmentación, ya que no se emplea en el presente trabajo, y se ha empleado únicamente la detección. Para ello es necesario ejecutar el comando mostrado en el Listado B.2. Cabe destacar que, en el caso de que use un conjunto de dataset que no sea el original de nuScenes, es necesario modificar el archivo `splits.py` del paquete `nuscenes-devkit` para cambiar la asignación de escenas a los subconjuntos de entrenamiento, validación y pruebas.

Listado B.2: Comando para entrenar la red neuronal

```
torchpack dist-run -np 1 -v python tools/train.py configs/
   nuscenes/det/transfusion/secfpn/camera+lidar/resnet50/
   convfuser.yaml
```

Por último, este entrenamiento genera una carpeta con un archivo de configuración tipo *yaml* y el fichero de pesos con extensión *pth*. La estructura del directorio de salida debe ser similar a lo mostrado en el siguiente árbol.

```
BEVFusion-ROS-TensorRT
└── bevfusion
    └── runs
        ├── 20241011_145007.log
        ├── 20241011_145007.log.json
        ├── 20241011_233013.log
        ├── 20241011_233013.log.json
        ├── configs.yaml
        ├── epoch_1.pth
        ├── epoch_2.pth
        └── epoch_3.pth
```

```
├── epoch_4.pth
├── epoch_5.pth
├── epoch_6.pth
├── epoch_7.pth
├── epoch_8.pth
├── epoch_9.pth
├── epoch_10.pth
├── epoch_11.pth
├── epoch_12.pth
├── epoch_13.pth
├── epoch_14.pth
├── epoch_15.pth
├── epoch_16.pth
├── epoch_17.pth
├── epoch_18.pth
├── epoch_19.pth
├── epoch_20.pth
├── latest.pth ->epoch_20.pth
└── logging
    ├── 2024-10-11_14-50-07_361434.log
    └── 2024-10-11_23-30-13_363518.log
```

Tras esto, se puede realizar una evaluación de la red empleando el subconjunto *test* con el comando mostrado en el Listado B.3

Listado B.3: Comando para evaluar la red neuronal

```
torchpack dist-run -np 1 python tools/test.py configs/nuscenes
   /det/transfusion/secfpn/camera+lidar/resnet50/convfuser.
   yaml runs/epoch\_20.pth --eval bbox
```

Una vez se han comprobado las métricas de la red neuronal, es necesario convertir el fichero de pesos de la red a formato *onnx* para, posteriormente, compilarlo y poder ser usado por el *wrapper* de BEVFusion para ROS2. Para obtener los ficheros *onnx* es necesario primero generar un modelo PTQ (*Post Training Quantization*), tras lo cual, ya se pueden exportar los ficheros *onnx*. Para ello, los comandos necesarios se muestran en el Listado B.4

Listado B.4: Comandos para exportar los ficheros onnx

```
python qat/ptq.py --config=bevfusion/configs/nuscenes/det/
   transfusion/secfpn/camera+lidar/resnet50/convfuser.yaml --
   ckpt=runs/epoch\_20.pth --calibrate_batch 300
python qat/export-camera.py --ckpt=runs/bevfusion_ptq.pth
python qat/export-transfuser.py --ckpt=runs/bevfusion_ptq.pth
python qat/export-scn.py --ckpt=runs/bevfusion_ptq.pth --save=
   qat/onnx_int8/lidar.backbone.onnx
```

Finalmente, para compilar los ficheros *onnx* es necesario navegar a la carpeta `BEVFusion-ROS-TensorRT/` y ejecutar el comando mostrado mostrado en el Listado B.5

Listado B.5: Comandos para compilar los ficheros onnx

```
./tool/build_trt_engine.sh
```

B.2. Procesamiento de los datos y puesta en marcha del simulador

Para utilizar el simulador y poner en marcha el nodo de ROS2 de procesamiento de los datos, se ha empleado una terminal dividida en ocho, donde cada una realizará una función concreta como se muestra en la Figura B.1.

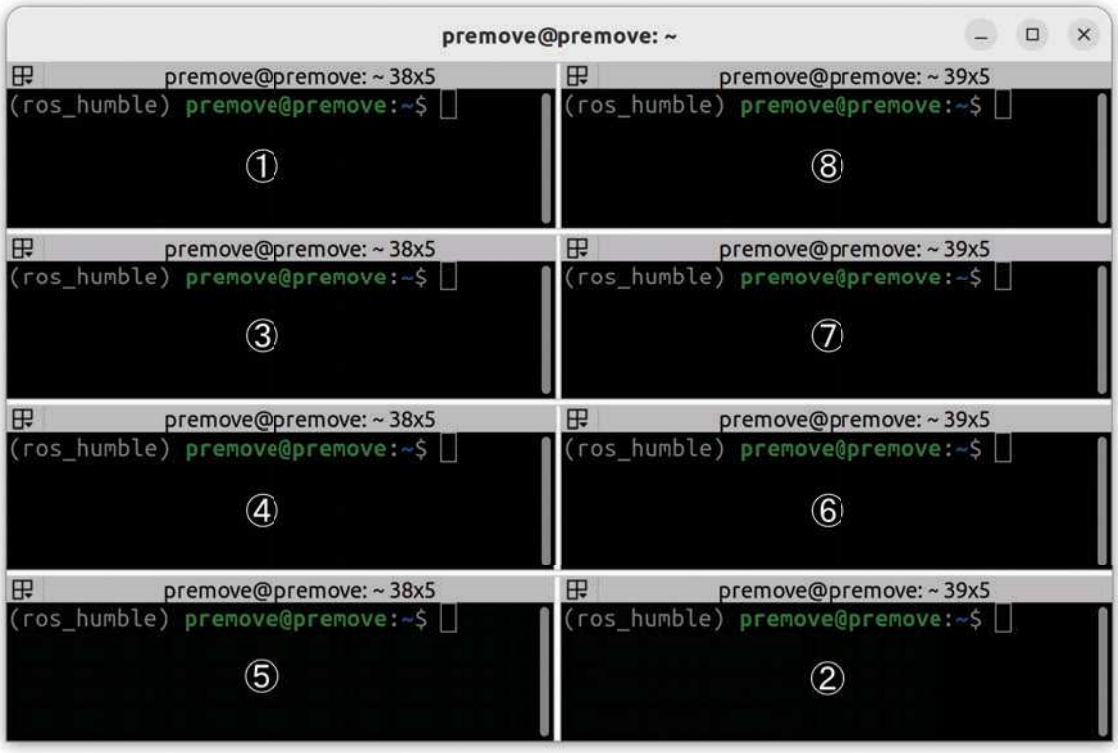


Figura B.1: Visualización de la terminal para la puesta en marcha del simulador.

B.2.1. Puesta en marcha del simulador CARLA

En primer lugar, es necesario ejecutar el simulador CARLA, en su versión *package* utilizando el comando mostrado en el Listado B.6

Listado B.6: Comando para ejecutar el simulador CARLA 0.9.15.

```
~/TFM/CARLA_0.9.15/./CarlaUE4.sh -vulkan -quality-level=Low -
   bThrottleCPUWhenNotForeground=False
```

Cabe destacar que como argumentos se ha escogido una baja calidad de gráficos para mejorar el rendimiento general de la máquina, el uso de Vulkan como motor gráfico en vez de OpenGl debido a que está mejor optimizado para trabajar con la última versión de CARLA. Finalmente, se ha establecido el argumento `-bThrottleCPUWhenNotForeground=False` para mejorar los FPS (*Frames Per Second*) durante el control del vehículo.

B.2.2. Cambio de mapa

En segundo lugar, si es necesario, es posible cambiar entre los diferentes mapas que ofrece CARLA. De entre ellos, se ha escogido el mapa *Town 15*, empleando para ello el comando mostrado en el Listado B.7

Listado B.7: Comando para cambiar de mapa en CARLA

```
python3 ~/TFM/CARLA_0.9.15/PythonAPI/util/config.py -m Town15
```

B.2.3. Puesta en marcha del punto entre CARLA y ROS2

En tercer lugar, es necesario ejecutar el puente de CARLA con ROS2, para ello se establece como modo de funcionamiento el síncrono con un *timeout* de 5 segundos para que los sensores tengan tiempo suficiente a inicializarse. Los comandos necesarios para realizar esto se muestran en el Listado B.8

Listado B.8: Comandos para iniciar el puente de CARLA con ROS

```
source ~/premove_ws/install/setup.bash
ros2 launch carla_ros_bridge carla_ros_bridge.launch.py
   passive:=True timeout:=5
```

B.2.4. Carga del vehículo sensorizado

En cuarto lugar, se procede con la carga del vehículo sensorizado dentro del simulador. Para ello se hace uso del comando mostrado en el Listado B.9

Listado B.9: Comandos para cargar el vehículo sensorizado en el simulador

```
source ~/premove_ws/install/setup.bash
ros2 launch carla_spawn_objects carla_spawn_objects.launch.py
   objects_definition_file:=/home/premove/TFM/code/
   objects_files/town15/fullRGB_lincoln_CF.json
```

B.2.5. Control manual del vehículo

Ahora se procede con la ejecución del *script* de Python, basado en *PyGame* que permite controlar el vehículo con un juego de volante y pedales. Para ello se hace uso del comando mostrado en el Listado B.10, donde cabe destacar que está particularizado para el vehículo que se ha cargado y se ha especificado una resolución de `1920x1080`.

Listado B.10: Comandos para controlar manualmente el vehículo sensorizado

```
python3 ~/TFM/code/vehicle_control/
   manual_lincoln_steering_and_traffic.py --res 1920x1080
```

B.2.6. Carga de los participantes del tráfico

Posteriormente, se procede a cargar los participantes del tráfico con el comando mostrado en el Listado B.11, donde el parámetro `-n` se refiere al número de vehículos y el parámetro `-w` se refiere al número de peatones. Además, cabe destacar que los vehículos se escogen de forma aleatoria entre coches, motos, bicicletas, camiones y autobuses.

Listado B.11: Comandos para generar tráfico

```
python3 ~/TFM/code/generate_traffic.py -n 30 -w 20
```

B.2.7. Ejecución de la red neuronal BEVFusion mediante ROS2

Una vez el simulador se encuentra en ejecución, ya se puede proceder a ejecutar el nodo de ROS2 que se conecta con la red neuronal BEVFusion. Para ello, es necesario utilizar los comandos mostrados en el Listado B.12.

Listado B.12: Comandos para poner en marcha el nodo de BEVFusion para ROS2

```
source ~/premove_ws/install/setup.bash
ros2 launch bevfusion bevfusion.launch.py
```

B.2.8. Visualización de los resultados

Finalmente, para visualizar los resultados es necesario emplear la herramienta RViz2, la cual se puede cargar junto con su archivo de configuración mediante los comandos mostrados en el Listado B.13

Listado B.13: Comandos para poner en marcha el visualizador de ROS2

```
source ~/premove_ws/install/setup.bash
rviz2 -d src/BEVFusion-ROS-TensorRT/launch/view.rviz
```

Cabe destacar que, en el caso de que se quiera grabar los tópicos de la escena se puede recurrir a la herramienta ROS2 Bag. Para grabar dichos tópicos es necesario hacer uso del comando mostrado en el Listado B.14, mientras que para reproducir el fichero *bag* creado hay que usar el comando mostrado en el Listado B.15

Listado B.14: Comandos para grabar tópicos de ROS2

```
ros2 bag record -o $NOMBRE$ $TOPICO\_1$ $TOPICO\_2$
```

Listado B.15: Comandos para reproducir ficheros bag de ROS2

```
ros2 bag play $fichero.bag$
```

En último lugar, tras ejecutar todos los comandos mostrados en el presente capítulo, se debe obtener una salida como la mostrada en la Figura B.2.

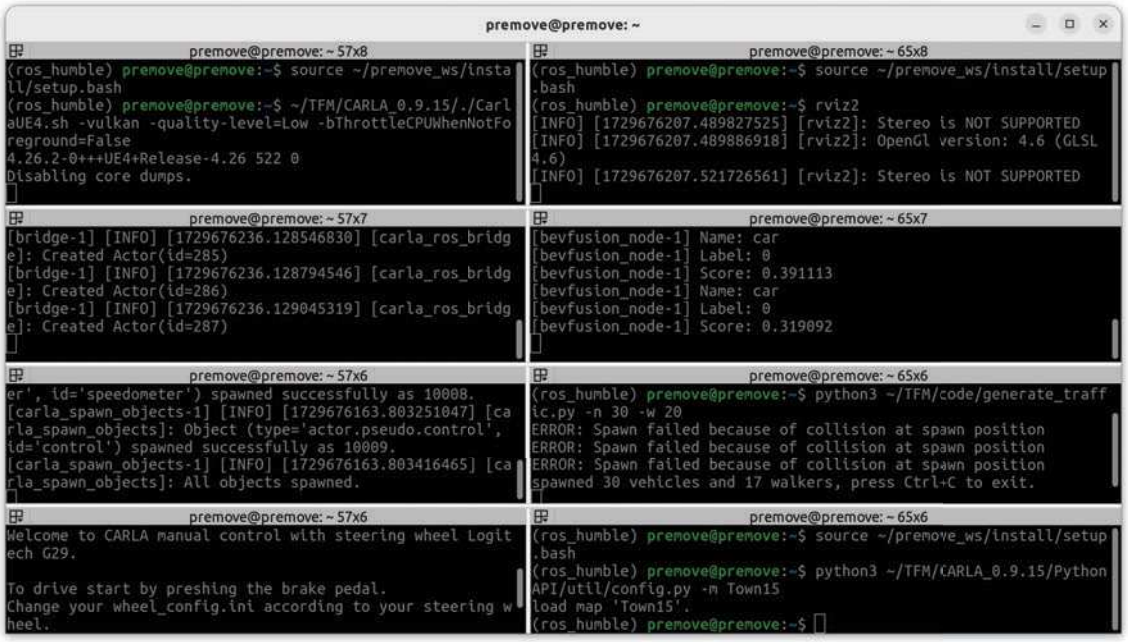


Figura B.2: Muestra de los mensajes de salida de las terminales.

Bibliografía

[1] Ovidiu-Aurelian Detesan and Iuliana Fabiola Moholea. Unimate and Beyond: Exploring the Genesis of Industrial Robotics. https://doi.org/10.1007/978-3-031-59257-7_27, 2024.

[2] Raul D. S. G. Campilho and Francisco J. G. Silva. Industrial Process Improvement by Automation and Robotics. https://doi.org/10.3390/machines11111011, 2023.

[3] IBM. Deep Blue. https://www.ibm.com/history/deep-blue, 2024. Accedido: 19-09-2024.

[4] Antonia Davison. The rise of robotics in the auto industry. https://www.ibm.com/blog/ai-robots-auto-industry/, 2024. Accedido: 19-09-2024.

[5] SAE. Coches autónomos en la UE: de la ciencia ficción a la realidad. https://www.europarl.europa.eu/topics/es/article/20190110STO23102/coches-autonomos-en-la-ue-de-la-ciencia-ficcion-a-la-realidad, 2021.

[6] Waymo. Sobre Waymo. https://waymo.com/intl/es/about/, 2024. Accedido: 19-09-2024.

[7] Divya Garikapati and Sneha Sudhir Shetiya. Autonomous Vehicles: Evolution of Artificial Intelligence and the Current Industry Landscape. https://doi.org/10.3390/bdcc8040042, 2024.

[8] Julien Moreau and Javier Ibanez-Guzman. Emergent Visual Sensors for Autonomous Vehicles. https://doi.org/10.1109/TITS.2023.3248483, 2023.

[9] O Urmila. and Rajesh Kannan Megalingam. Processing of LiDAR for Traffic Scene Perception of Autonomous Vehicles. https://doi.org/10.1109/ICCSP48568.2020.9182175, 2020.

[10] Alberto García Guillén. Trabajo de Fin de Grado: Detección de agentes del tráfico en entornos urbanos en simulación utilizando CARLA y ROS2. 2023.

[11] Zhongmou Dai, Zhiwei Guan, Qiang Chen, Yi Xu, and Fengyi Sun. Enhanced Object Detection in Autonomous Vehicles through LiDAR—Camera Sensor Fusion. https://doi.org/10.3390/wevj15070297, 2024.

[12] Simegnew Yihunie Alaba, Ali C. Gurbuz, and John E. Ball. Emerging Trends in Autonomous Vehicle Perception: Multimodal Fusion for 3D Object Detection. https://doi.org/10.3390/wevj15010020, 2024.

[13] J. Morales and J. L. Martínez. 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/, 2024.

[14] Zhijian Liu, Haotian Tang, Alexander Amini, Xinyu Yang, Huizi Mao, Daniela L. Rus, and Song Han. BEVFusion: Multi-Task Multi-Sensor Fusion with Unified Bird's-Eye View Representation. https://doi.org/10.1109/ICRA48891.2023.10160968, 2023.

[15] 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].

[16] Open Robotics. ROS 2 Documentation: Humble. https://docs.ros.org/en/humble/, 2024. Accedido: 19-09-2024.

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

[18] 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].

[19] NVIDIA. CUDA Toolkit Documentation. https://developer.nvidia.com/cuda-toolkit, 2024. Accedido: 19-09-2024.

[20] NVIDIA. CUDA Deep Neural Network Library (cuDNN). https://developer.nvidia.com/cudnn, 2024. Accedido: 19-09-2024.

[21] NVIDIA. TensorRT SDK. https://developer.nvidia.com/tensorrt, 2024. Accedido: 19-09-2024.

[22] NVIDIA. Lidar AI Solution. https://github.com/NVIDIA-AI-IOT/Lidar_AI_Solution, 2024. Accedido: 24-09-2024.

[23] LinClubs. BEVFusion-ROS-TensorRT. https://github.com/linClubs/BEVFusion-ROS-TensorRT, 2024. Accedido: 24-09-2024.

[24] Centro de Visión por Computador (CVC). Centro de visión por computador (cvc). https://www.cvc.uab.es/, 2024. Accedido: 26-09-2024.

[25] Intel Corporation. Intel research overview. https://www.intel.la/content/www/xl/es/research/overview.html, 2024. Accedido: 26-09-2024.

[26] Toyota Research Institute. Toyota Research Institute (TRI). https://www.tri.global/, 2024. Accedido: 26-09-2024.

[27] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. https://doi.org/10.48550/arXiv.1711.03938, 13–15 Nov 2017. arXiv:1711.03938 [cs.LG].

[28] CARLA Simulator Team. CARLA Tutorial: OpenStreetMap. https://carla.readthedocs.io/en/latest/tuto_G_openstreetmap/, 2024. Accedido: 27-09-2024.

[29] CARLA Simulator Team. CARLA Tutorial: RoadRunner. https://carla.readthedocs.io/en/latest/tuto_M_generate_map/, 2024. Accedido: 27-09-2024.

[30] Epic Games. CARLA democratizes autonomous vehicle R&D with free open-source simulator. https://www.unrealengine.com/en-US/spotlights/carla-democratizes-autonomous-vehicle-r-d-with-free-open-source-simulator. Accedido: 30-10-2024.

[31] CARLA Simulator. Issue #2879: Documentation improvement on installing dependencies for linux. https://github.com/carla-simulator/carla/issues/2879. Accedido: 30-10-2024.

[32] CARLA Simulator Team. CARLA Release 0.9.15. https://carla.org/2023/11/10/release-0.9.15/, 2023. Accedido: 27-09-2024.

[33] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. https://doi.org/10.48550/arXiv.2211.07752, 2022.

[34] ROS 2 Team. ROS1_Bridge. https://github.com/ros2/ros1_bridge, 2024. Accedido: 14-10-2024.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. https://doi.org/10.48550/arXiv.1912.01703, 2019.

[36] NVIDIA. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit, 2024. Accedido: 15-10-2024.

[37] NVIDIA. NVIDIA cuDNN: Biblioteca de primitivas eficientes para el aprendizaje profundo. https://developer.nvidia.com/cudnn, 2024. Accedido: 15-10-2024.

[38] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan M. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. https://doi.org/10.48550/arXiv.1410.0759, 2014.

[39] NVIDIA. NVIDIA TensorRT: Plataforma de optimización para la inferencia de IA, 2024. Accedido: 15-10-2024.

[40] Xuyang Bai, Zeyu Hu, Xinge Zhu, Qingqiu Huang, Yilun Chen, Hongbo Fu, and Chiew-Lan Tai. Transfusion: Robust lidar-camera fusion for 3D object detection with Transformers. https://doi.org/10.48550/arXiv.2203.11496, 2022.

[41] NVIDIA AI IOT. LiDAR AI Solution. https://github.com/NVIDIA-AI-IOT/Lidar_AI_Solution, 2024. Accedido: 21-10-2024.

[42] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, and Liong. nuscenes: A multimodal dataset for autonomous driving. https://doi.org/10.48550/arXiv.1903.11027, 2020.

[43] Shounak Sural, Nishad Sahu, and Ragunathan Raj Rajkumar. ContextualFusion: Context-Based Multi-Sensor Fusion for 3D Object Detection in Adverse Operating Conditions. https://doi.org/10.48550/arXiv.2404.14780, 2024.

[44] nuScenes by Motional. nuScenes: A Public Dataset for Autonomous Driving. https://www.nuscenes.org/, 2024. Accedido: 19-09-2024.

[45] Velodyne. Sensor LIDAR 3D Velodyne HDL-32E. https://www.aeroexpo.online/es/prod/velodyne/product-176220-32080.html, 2024. Accedido: 27-10-2024.

[46] Basler AG. Cámara Basler ACE 1600-60gc - Basler AG. https://www.baslerweb.com/en/shop/aca1600-60gc/, 2024. Accedido: 27-10-2024.

[47] Daniel Steven Gamba Correa et al. Trabajo de Fin de Grado: Incorporación de sensores realistas en la simulación en CARLA de la navegación de coches autónomos. 2023.

[48] Sumbal Malik, Manzoor Khan, Aadam , Hesham El-Sayed, Farkhund Iqbal, Muhammad Khan, and Obaid Ullah. CARLA+: An Evolution of the CARLA Simulator for Complex Environment Using a Probabilistic Graphical Model. https://doi.org/10.3390/drones7020111, 02 2023. 10.3390/drones7020111.

[49] 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].

[50] 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].

[51] 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].

[52] 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].

[53] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. https://doi.org/10.1109/CVPR.2018.00472, 2018.

[54] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. https://doi.org/10.48550/arXiv.2103.14030, 2021.

[55] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. https://doi.org/10.48550/arXiv.1512.03385, 2016.

[56] Jonah Philion and Sanja Fidler. Lift, splat, shoot: Encoding images from arbitrary camera rigs by implicitly unprojecting to 3d. https://doi.org/10.48550/arXiv.2008.05711, 2020.

[57] Junjie Huang, Guan Huang, Zheng Zhu, Yun Ye, and Dalong Du. BEVDet: High-performance multi-camera 3D object detection in Bird-Eye-View. https://doi.org/10.48550/arXiv.2112.11790, 2021.

[58] J. Montenegro. Integración de técnicas de detección de participantes del tráfico en entornos urbanos mediante un sensor LIDAR 3D y cámaras RGB con CARLA y ROS2. https://drive.google.com/drive/folders/1xTiMPBpz3RAAy3-vY427c4YOBAmwR6a5?usp=drive_link, 2024.

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

[60] CARLA Simulator Team. CARLA Docs Download. https://github.com/carla-simulator/carla/blob/master/Docs/download.md, 2024.

[61] CARLA Simulator Team. CARLA Pypi. https://pypi.org/project/carla/, 2024.

[62] NVIDIA. CUDA Toolkit 11.7.0 - Download Archive. https://developer.nvidia.com/cuda-11-7-0-download-archive, 2024. Accedido: 22-10-2024.

[63] Philipp Hohenecker. Switch CUDA. https://github.com/phohenecker/switch-cuda, 2024. Accedido: 22-10-2024.

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