

MEMORIA DEL TRABAJO FINAL DE GRADO

Grado en Inteligencia Robótica

Universidad Jaume I de Castellón

Robots *Robotnik* en *O3DE* y *CoppeliaSim*: Integración y Capacidades con *ROS2*

Estudiante: Albert Gabriel Matei

Tutor: José Vicente Martí Avilés

Curso académico: 2024–2025

Fecha de defensa: 23 de julio de 2025

Agradecimientos: A mis padres, por haber confiado en mí al cambiarme a este grado.

Palabras clave: ROS2, O3DE, CoppeliaSim, Robotnik, robots móviles, SLAM 3D, integración, rendimiento.

CASTELLÓN, a 9 de Julio de 2025

Todos los derechos reservados

TABLA DE CONTENIDOS

TABLA DE CONTENIDOS	2	
1	Introducción	7
1.1	Contexto	7
1.2	Motivación del trabajo	8
1.3	Alcance del proyecto	9
1.4	Aportaciones e innovación	9
1.5	Descripción de la empresa	10
1.6	Estructura del documento	10
2	Marco Teórico	13
2.1	Funcionamiento general de los simuladores robóticos	13
2.1.1	Motor físico	13
2.1.2	Motor gráfico	14
2.1.3	Modelado del robot y el entorno	14
2.1.4	Simulación de sensores	15
2.1.5	Comunicación y control externo en simuladores robóticos	15
2.2	<i>ROS2 y su funcionamiento</i>	16
2.3	Paquetes en <i>ROS2</i>	17
2.4	<i>SLAM 3D en ROS2</i>	17
3	Planificación	19
3.1	Objetivos	19
3.2	Planificación Inicial	20
3.3	Desviaciones del plan inicial	20

3.4	Estimación de costes	23
3.4.1	Coste asociado a las horas de trabajo en el desarrollo del proyecto	23
3.4.2	Coste proporcional del <i>hardware</i>	23
3.4.3	Coste asociado al <i>software</i> utilizado en el proyecto	24
3.4.4	Otros gastos operativos	24
3.4.5	Presupuesto total del proyecto	25
4	Análisis	27
4.1	Descripción de los 2 robots implementados	27
4.1.1	<i>Rb_Watcher</i>	27
4.1.2	<i>Rb_Robout</i>	29
4.2	Requisitos funcionales	30
4.2.1	Requisitos funcionales de los simuladores	30
4.2.2	Requisitos funcionales de los paquetes <i>SLAM 3D</i>	31
4.3	Requisitos no funcionales	31
4.3.1	Requisitos no funcionales de los robots en los simuladores	31
4.3.2	Requisitos no funcionales de los paquetes <i>SLAM 3D</i>	32
4.4	Diagrama de Actividades	32
4.5	Arquitectura de simulación robótica para <i>CoppeliaSim</i>	33
4.6	Arquitectura de la simulación robótica para <i>O3DE</i>	34
5	Implementación	39
5.1	Implementación de robots móviles en <i>CoppeliaSim</i>	40
5.1.1	<i>Setup</i> del simulador	40
5.1.2	Obtención del archivo <i>URDF</i> del robot a partir del archivo <i>Xacro</i>	40
5.1.3	Colocación de los sensores en los elementos del robot	41
5.1.4	Sistema dinámico de asignación de <i>namespaces</i> para múltiples robots	42
5.1.5	Implementación de los sensores	44
5.1.6	Publicación de <i>tf</i> en <i>ROS2</i>	48
5.1.7	Control de movimiento	48
5.1.8	<i>Script</i> de <i>spawn</i> dinámico de robots	49
5.1.9	<i>Script</i> de carga de escena	50
5.2	Implementación de robots móviles en <i>O3DE</i>	50
5.2.1	<i>Setup</i> del simulador	50
5.2.2	Importación de archivos <i>URDF</i>	51
5.2.3	Configuración de sensores en <i>O3DE</i>	51
5.2.4	Implementación del control	55
5.2.5	Implementación de la odometría	56
5.2.6	Implementación del <i>spawn</i> dinámico	57
5.3	Configuración y uso de paquetes de <i>SLAM 3D</i> en robots simulados	57
5.3.1	<i>RTAB-MAP</i>	58
5.3.2	<i>Lidarslam_ros2</i>	58
5.3.3	<i>MOLA</i>	59
6	Resultados	61
6.1	Resultados de la integración de los robots con <i>ROS2</i>	61
6.1.1	Resultados en <i>CoppeliaSim</i>	62

6.1.2	Resultados en <i>O3DE</i>	66
6.2	Resultados del rendimiento de los robots	70
6.2.1	Rendimiento de los robots en <i>O3DE</i>	71
6.2.2	Rendimiento de los robots en <i>CoppeliaSim</i>	74
6.3	Resultados del rendimiento de los paquetes de <i>SLAM 3D</i>	79
7	Conclusiones y posible trabajo futuro	86
	Bibliografía	88

1. Introducción

En este capítulo se presenta el marco general del proyecto. Se inicia con el *Contexto*, donde se expone la importancia de los simuladores en la robótica móvil y la integración con *middleware* como *ROS2*. A continuación, en la sección de *Motivación*, se detallan las razones técnicas y prácticas que impulsan este trabajo, destacando los retos actuales y la evolución del ecosistema *ROS*.

El apartado de *Aportaciones e Innovación* describe las contribuciones principales del proyecto. Seguidamente, en la *Descripción de la empresa*, se ofrece información sobre *Robotnik Automation*, entidad vinculada con la realización de este trabajo.

Finalmente, la *Estructura del documento* detalla la organización y el contenido de los capítulos que componen el resto del informe, proporcionando una guía para su lectura y comprensión.

1.1 Contexto

En el desarrollo de sistemas de robótica móvil, los simuladores han adquirido un papel fundamental como herramientas de diseño, validación y experimentación. Estos entornos virtuales permiten evaluar tanto el *hardware* como el *software* de un robot sin necesidad de disponer de una plataforma física, lo que contribuye a minimizar riesgos, reducir costes y acortar los ciclos de desarrollo.

En la práctica, trabajar directamente sobre robots físicos presenta múltiples desafíos: desgaste de componentes, posibilidad de accidentes, consumo de tiempo en cada iteración, o la necesidad de condiciones de prueba controladas. Frente a ello, la simulación ofrece una alternativa eficaz y segura, al permitir ejecutar pruebas en entornos virtuales configurables y repetibles, sin consecuencias materiales. De esta forma, los errores se pueden detectar y corregir tempranamente, y los algoritmos se validan en condiciones consistentes.

Una característica clave de los simuladores modernos de robots es su capacidad para integrarse con *middleware* robótico como *ROS2* (*Robot Operating System 2*), que proporciona una arquitectura modular y distribuida para el desarrollo de *software* en robótica. Un *middleware* es una capa intermedia de *software* que facilita la comunicación y gestión de datos entre diferentes componentes o nodos de un sistema. Esta integración permite simular de forma realista sensores como cámaras

RGBD (*Red, Green, Blue, Depth*) o *LIDAR* (*Light Detection and Ranging*), y emular la comunicación entre nodos de procesamiento, tal como ocurriría en un sistema físico.

Herramientas como *Gazebo Ignition* (evolución de *Gazebo Classic*) y *Webots*, ampliamente utilizadas en entornos académicos e industriales, disponen de extensiones oficiales para *ROS2*. Por otro lado, otros simuladores menos conocidos como *CoppeliaSim* y *O3DE* (*Open 3D Engine*) también ofrecen compatibilidad mediante *plugins* (extensiones de software que aportan nuevas capacidades) y puentes personalizados, ampliando el abanico de opciones disponibles según las necesidades del proyecto.

En suma, los simuladores no solo permiten anticipar problemas de implementación, sino que también sirven como plataformas versátiles para la formación, la investigación y el desarrollo de sistemas robóticos complejos, especialmente en contextos donde se requiere una alta precisión sensorial, entornos multirrobot o algoritmos avanzados de navegación y mapeo.

1.2 Motivación del trabajo

Durante años, *ROS* ha sido el estándar de facto para el desarrollo de *software* en robótica, gracias a su enfoque modular, su amplia comunidad y su adaptabilidad a múltiples dominios. No obstante, su arquitectura inicial presentaba limitaciones importantes [2] : dificultades para garantizar comunicaciones en tiempo real, escasa compatibilidad con sistemas operativos distintos de *Linux* y una estructura centralizada poco adecuada para entornos distribuidos.

Para superar estas barreras, se desarrolló *ROS2*, una evolución que mantiene los principios fundamentales de *ROS* pero adopta una arquitectura basada en *DDS* (Data Distribution Service). Esto ha permitido una mayor fiabilidad en la comunicación entre nodos, soporte nativo para múltiples plataformas (*Linux*, *Windows*, *macOS*), y una mejor adecuación a sistemas embebidos y aplicaciones industriales. Esta transición ha favorecido una migración progresiva hacia *ROS2*, que hoy constituye la base tecnológica de numerosos proyectos avanzados de robótica.

Como consecuencia, los principales simuladores robóticos también han tenido que adaptarse a este nuevo paradigma. *Gazebo Classic*, por ejemplo, ha sido reemplazado por *Gazebo Ignition*, con una arquitectura más moderna y mejor integración con *ROS2*. Del mismo modo, *Webots* ha evolucionado para ofrecer soporte oficial a este nuevo *middleware*. Sin embargo, el proceso de integración no ha estado exento de desafíos: uno de los principales problemas ha sido la complejidad de implementar entornos multirrobot en *ROS2*, una tarea aún poco documentada y propensa a errores.

Ante este escenario, ha surgido un interés creciente por explorar simuladores alternativos, como *CoppeliaSim* y *O3DE*, que si bien no cuentan con soporte oficial, podrían representar opciones más accesibles o potentes en determinadas aplicaciones. Estas plataformas permiten configuraciones personalizadas, integraciones mediante puentes *ROS2* y una mayor flexibilidad en el diseño de entornos tridimensionales complejos.

En este contexto, se plantea una motivación técnica clave: evaluar la viabilidad de estas plataformas emergentes para simular robots equipados con sensores avanzados y ejecutar algoritmos de *SLAM 3D* (Simultaneous Localization and Mapping tridimensional), una de las tareas más exigentes de la robótica móvil. El *SLAM 3D* permite que un robot construya un mapa tridimensional de su entorno mientras se localiza dentro de él, lo cual requiere una gestión eficiente de datos sensoriales en tiempo real y una alta precisión en la sincronización entre simulador y *middleware*. Evaluar esta funcionalidad es una forma eficaz de medir el rendimiento, la estabilidad y la compatibilidad técnica de las distintas plataformas.

Una motivación adicional para este proyecto surgió de la experiencia profesional durante las prácticas en *Robotnik Automation*, donde se analizó la complejidad de migrar desde *Gazebo Classic* a simuladores como *CoppeliaSim* y *O3DE*, así como la capacidad de estas plataformas para

reproducir fielmente el comportamiento de los robots de la empresa, tanto en rendimiento como en gestión simultánea de múltiples unidades.

1.3 Alcance del proyecto

El presente proyecto se centra en evaluar la viabilidad técnica y operativa de implementar robots móviles interconectados mediante *ROS2* en dos plataformas de simulación específicas: *CoppeliaSim* y *O3DE*. La investigación abarca la configuración y simulación de robots virtuales equipados con sensores avanzados, como cámaras *RGBD* y *LIDAR 3D* y *2D*, así como la integración y comunicación efectiva entre estos robots y el *middleware ROS2*.

Asimismo, el proyecto incluye la ejecución y validación de algoritmos de *SLAM 3D* en estos entornos simulados y el análisis de la calidad de mapeado 3D resultante. Este enfoque permite determinar en qué medida los simuladores seleccionados soportan tareas complejas y críticas para la robótica móvil autónoma.

Para garantizar la comparabilidad de resultados y facilitar futuras replicaciones, se desarrollan escenarios simples y estandarizados que se pueden implementar en ambos simuladores bajo condiciones equivalentes. Este diseño permite una evaluación objetiva de las capacidades técnicas, rendimiento y estabilidad de cada plataforma en un contexto controlado.

Cabe destacar que el proyecto no contempla la experimentación con *hardware* físico ni la implementación de redes multirrobot a gran escala, ni el desarrollo de nuevos algoritmos *SLAM*, sino que se enfoca en analizar la capacidad de los simuladores para soportar estas funciones, evaluando la integración con *ROS2* y la ejecución efectiva de algoritmos existentes en un entorno virtual. Del análisis de resultados de *SLAM 3D* se enfocará únicamente en el mapeado tridimensional, al ser la tarea de mayor complejidad; no obstante, todos los paquetes evaluados demostraron también la capacidad de registrar la trayectoria seguida por el robot.

1.4 Aportaciones e innovación

Este trabajo pretende aportar valor en un área crítica del desarrollo robótico moderno: la evaluación y selección de simuladores capaces de reproducir con fidelidad entornos tridimensionales complejos y ejecutar tareas computacionalmente exigentes, como el *SLAM 3D*, dentro del ecosistema *ROS2*. Aunque existen múltiples estudios centrados en la simulación con *Gazebo Sim* o *Webots*, son escasos los trabajos que profundizan en el análisis comparativo de simuladores no tradicionales como *CoppeliaSim* y *O3DE* en el contexto de *ROS2* y *SLAM* tridimensional.

Las principales aportaciones e innovaciones del proyecto se pueden sintetizar en los siguientes puntos:

- **Exploración de plataformas emergentes:** se evalúan dos simuladores poco documentados en la literatura científica en combinación con *ROS2*, lo cual puede abrir nuevas vías de investigación y desarrollo en simulación robótica más allá de las soluciones clásicas.
- **Comparación detallada y cuantitativa:** se lleva a cabo un análisis técnico riguroso del rendimiento y compatibilidad de *CoppeliaSim* y *O3DE* usando los robots de Robotnik Automation, mediante un conjunto amplio de métricas objetivas que permiten medir la eficiencia y estabilidad del sistema bajo dos tipos de escenarios y diferente número de robots.
- **Implementación de *SLAM 3D*:** se integran y prueban algoritmos avanzados de localización y mapeo tridimensional (*RTAB-MAP*, *MOLA* y *Lidar slam_ros2*) sobre simuladores alternativos, para comprobar su factibilidad técnica.

- **Diseño de entornos multirrobot y sensorización avanzada:** se construyen configuraciones virtuales completas que replican robots dotados de sensores complejos, permitiendo evaluar tanto el rendimiento como la utilidad práctica en proyectos reales.
- **Contribución a la documentación:** se generan pautas, configuraciones y soluciones prácticas que pueden ser aprovechadas por otros desarrolladores o investigadores que enfrenten los mismos retos de integración con *ROS2* en simuladores no oficiales.

En conjunto, este trabajo no solo amplía el conocimiento sobre herramientas de simulación robótica, sino que también ofrece una guía útil para evaluar nuevas plataformas en un contexto técnico realista.

1.5 Descripción de la empresa

A lo largo del proyecto se hará referencia frecuentemente a la empresa *Robotnik Automation*, empresa en la cual el autor del presente documento tuvo la oportunidad de realizar la estancia en prácticas. Es por ello que se hace necesario describirla.

Robotnik Automation S.L., con CIF B97223630, es una sociedad limitada ubicada en la *Comunidad Valenciana*. Su domicilio fiscal se encuentra en la *Ronda Auguste y Louis Lumière*, números 6 y 8, en el municipio de *Paterna* (46980), mientras que su oficina técnica y operativa está situada en la *Calle Berní y Catalá*, número 53, en *Valencia* (46019).

La empresa se especializa en el diseño, fabricación y comercialización de productos de robótica de servicio. Además, ofrece servicios de ingeniería e investigación y desarrollo (I+D) orientados a la automatización y la robótica móvil. Su principal objetivo consiste en proporcionar soluciones tecnológicas avanzadas para sectores como la industria, la logística y la inspección.

Entre sus principales líneas de actividad se encuentran el desarrollo de plataformas móviles, brazos robóticos, manipuladores y robots cooperativos. También participa en proyectos de I+D tanto a nivel nacional como internacional, y mantiene una actividad destacada en comercio exterior, con operaciones de importación y exportación de sus productos. La empresa integra tecnologías punteras como navegación autónoma, visión artificial y conectividad 5G en sus desarrollos.

Robotnik cuenta con una estructura organizativa compuesta por un equipo multidisciplinar de aproximadamente 70 empleados (dato actualizado a 2024), especializados en áreas como ingeniería robótica, *software*, diseño mecánico, electrónica y gestión de proyectos tecnológicos.

Sus instalaciones principales, con una superficie de unos 7.000 metros cuadrados en *Paterna*, incluyen oficinas administrativas, zonas de fabricación e integración, laboratorio de pruebas y áreas logísticas. Estas instalaciones están equipadas para cubrir todas las fases de diseño, montaje, validación y demostración de sus sistemas robóticos.

Actualmente, la empresa se encuentra en activo y ocupa una posición relevante dentro del sector de la robótica de servicio, con una clara proyección internacional. Según su clasificación *CNAE 7120* (“*Ensayos y análisis técnicos*”)[1], se sitúa en la posición 99 a nivel sectorial, en el puesto 3.334 dentro del *ranking* provincial de *Valencia* y en el lugar 57.977 a nivel nacional.

1.6 Estructura del documento

El presente documento se organiza en siete capítulos que abordan de manera ordenada y sistemática el desarrollo integral del proyecto.

El capítulo 1, Introducción, establece el contexto general, la motivación que sustenta el trabajo, el alcance definido y las principales aportaciones e innovaciones logradas. Asimismo, incluye una descripción de la empresa involucrada, brindando un marco referencial para la comprensión del proyecto.

El capítulo 2, Marco Teórico, expone los fundamentos conceptuales esenciales relacionados con los simuladores robóticos, abarcando aspectos como su funcionamiento general, la integración con *ROS2* y una revisión de los paquetes de *SLAM* tridimensional, elementos clave para el desarrollo posterior.

En el capítulo 3, Planificación, se presenta la definición de objetivos, la planificación inicial, así como las desviaciones y ajustes realizados a lo largo del proyecto. Además, se realiza un análisis detallado de los costes asociados, permitiendo evaluar la inversión de recursos en tiempo, *hardware* y *software*.

El capítulo 4, Análisis, se centra en la descripción y caracterización de los robots implementados, así como en la identificación de los requisitos funcionales y no funcionales que guían su desarrollo. De igual forma, se detalla la arquitectura de la simulación de robots para las plataformas utilizadas explicando su integración con el entorno *ROS2*.

El capítulo 5, Implementación, detalla los procesos técnicos y metodológicos empleados para la construcción de los robots móviles en ambos simuladores. Se abordan aspectos como la configuración de sensores, el control del movimiento, la gestión dinámica de múltiples robots y la incorporación de los paquetes *SLAM 3D*.

El capítulo 6, Resultados, recopila y analiza los datos obtenidos durante las pruebas realizadas, evaluando el desempeño tanto de los robots como de los algoritmos de mapeo, lo cual permite validar el cumplimiento de los objetivos planteados.

Finalmente, el capítulo 7, Conclusiones y posibles líneas de trabajo futuro, sintetiza las principales conclusiones derivadas del proyecto y propone recomendaciones para el desarrollo posterior y la mejora continua.

2. Marco Teórico

En este capítulo se explicará los fundamentos más importantes para entender este proyecto. En primer lugar, al tratarse de un proyecto que busca implementar robots en diferentes simuladores, se explicará el funcionamiento general de un simulador robótico explicando su partes más relevantes. En segundo lugar, al tener que comunicar los robots con *ROS2*, se explicará el funcionamiento de este. Finalmente en las últimas dos secciones, se detallará qué es un paquete de *ROS2* y cómo funcionan los paquetes de *SLAM 3D*.

2.1 Funcionamiento general de los simuladores robóticos

Los simuladores robóticos son plataformas de *software* que recrean de forma virtual el comportamiento de robots y sus entornos en *3D*. Permiten replicar tanto la dinámica física como el funcionamiento de sensores y actuadores, facilitando así el diseño, prueba y validación de algoritmos y sistemas robóticos sin la necesidad de disponer del *hardware* físico, lo que ahorra costes y riesgos. A continuación se presentará los diferentes elementos de los que se compone un simulador robótico moderno.

2.1.1 Motor físico

El motor físico es el corazón del simulador. Su función es aplicar las leyes de la física para calcular cómo se comportan los objetos en el entorno simulado. Esto incluye movimientos, colisiones, fuerzas, fricción, gravedad y otros efectos dinámicos. Por ejemplo, si un robot choca contra un obstáculo, el motor físico determina cómo reaccionan ambos objetos: si se detienen, se deslizan o se deforman (en simuladores más avanzados). Existen diferentes motores físicos utilizados en simuladores, cada uno con distintas capacidades y niveles de realismo, entre los cuales destacan los siguientes:

- *ODE (Open Dynamics Engine)*: popular por su rapidez y simplicidad. Incluye tipos de articulaciones avanzados e incorpora detección de colisiones con fricción [3]. *ODE* resulta especialmente útil para la simulación de vehículos, la gestión de objetos en entornos de realidad virtual y la animación de criaturas virtuales.

- *Bullet*: se encarga de simular la detección de colisiones y la dinámica tanto de cuerpos rígidos como deformables. Su uso se extiende desde motores de videojuegos hasta la creación de efectos visuales en el cine [4].
- *PhysX*: desarrollado por *NVIDIA*, ofrece simulación avanzada en tiempo real y aceleración por *GPU*[5].
- *DART*: orientado a la simulación de robots con alta precisión dinámica. Ofrece estructuras de datos y algoritmos precisos y estables para la cinemática y dinámica de sistemas multicuerpo articulados. A diferencia de otros motores físicos, *DART* expone directamente cantidades internas (matriz de masas, fuerzas de *Coriolis*, jacobianos, etc.)[6].

La elección del motor depende del equilibrio necesario entre precisión y rendimiento, adaptándose a las necesidades del proyecto.

2.1.2 Motor gráfico

Un motor gráfico es un *software* esencial en la representación visual en tiempo real de entornos simulados o interactivos. Es ampliamente utilizado en el desarrollo de videojuegos, pero también en campos como la animación, la arquitectura y la ingeniería para visualizar proyectos en 3D. Su función principal es renderizar modelos tridimensionales, aplicar texturas, gestionar iluminación y sombras, y generar efectos visuales atractivos e inmersivos mediante el uso de librerías como *OpenGL*, *Vulkan* o *DirectX*.

Este componente opera gracias a la integración de varios subsistemas, entre ellos el sistema de iluminación, cámaras, partículas y físicas[7]. La iluminación crea una sensación de realismo mediante sombras, reflejos y luces; el sistema de cámaras permite observar la escena desde diferentes perspectivas; el sistema de partículas genera efectos visuales como explosiones o niebla, y el motor de físicas simula comportamientos del mundo real, como colisiones o gravedad.

Aunque no participa directamente en cálculos físicos ni en la generación de datos sensoriales, el motor gráfico es clave para visualizar el comportamiento de un robot y su entorno de manera clara e intuitiva. Esto facilita la detección de errores en la simulación, la supervisión automática del sistema y la presentación de resultados durante sesiones de análisis o demostraciones.

Un motor gráfico de calidad mejora considerablemente la experiencia del usuario, ya que proporciona una interfaz visual más atractiva y receptiva, además de simplificar tanto el control manual como la monitorización de las simulaciones en tiempo real.

2.1.3 Modelado del robot y el entorno

Los simuladores trabajan con modelos digitales que describen los robots y su entorno:

- **Modelos de robots:** Incluyen la geometría (forma y tamaño), propiedades físicas (masa, inercia), articulaciones (ángulos y grados de libertad), puntos de montaje para sensores o herramientas, y características de los actuadores. Estos modelos pueden definirse con archivos estándar como *URDF* (Unified Robot Description Format) o *SDF* (Simulation Description Format), que permiten una descripción detallada y reutilizable. Además de esos tipos de archivos, cada simulador puede tener su formato específico, por ejemplo, en *CoppeliaSim* es *TTM* y en *O3DE* es *Prefab*.
- **Modelos de entorno:** Representan elementos como suelos, paredes, obstáculos, objetos manipulables o áreas de navegación. Estos pueden ser estáticos o dinámicos, interactuando con el robot o el motor físico. Se suelen llamar escenas o niveles y es dónde se añaden los robots para simularlos.

Un modelado preciso es clave para obtener simulaciones realistas que reflejen con fidelidad las condiciones del mundo real.

2.1.4 Simulación de sensores

Para que los algoritmos robóticos puedan funcionar en simulación, es necesario emular sensores reales, generando datos sintéticos que imitan las lecturas que un robot físico obtendría en su entorno. Para ello, en simuladores no dedicados nativamente a la robótica, puede haber *plugins* que realicen esa labor. Los simuladores suelen incluir la simulación de:

- **Cámaras RGB (Red,Green,Blue) y RGBD:** proporcionan imágenes o mapas de profundidad.
- **LIDAR:** emite haces láser para medir distancias y crear mapas del entorno.
- **Sensores ultrasónicos:** detectan obstáculos cercanos mediante ondas sonoras.
- **GPS (Global Positioning System) y sistemas de posicionamiento:** simulan la localización global.
- **IMU (Unidades de Medida Inercial):** ofrecen datos de aceleración y orientación.

Estos sensores simulados pueden configurarse para introducir niveles controlados de ruido, retrasos o fallos, ayudando a probar la robustez de los algoritmos frente a condiciones imperfectas.

2.1.5 Comunicación y control externo en simuladores robóticos

Una característica fundamental de los simuladores robóticos modernos es su capacidad para interactuar con otros sistemas y *software* externos. Esta comunicación permite que los algoritmos de control, navegación, percepción u otras aplicaciones puedan recibir datos del simulador en tiempo real y enviar comandos para controlar los robots virtuales. De esta manera, el simulador actúa como un entorno de prueba seguro y flexible para desarrollar y validar sistemas robóticos complejos.

Existen diversas formas y protocolos para establecer esta comunicación, entre las que destacan:

- **Interfaces de red :** como por ejemplo *sockets* [8] (puntos finales de una conexión de red que permiten el envío y recepción de datos entre procesos, ya sea en la misma máquina o en equipos distintos), *TCP/IP (Transmission Control Protocol / Internet Protocol)* o *UDP (User Datagram Protocol)*. Estos mecanismos permiten el intercambio de datos mediante paquetes enviados por la red.
- **APIs (Application Programming Interface) específicas del simulador:** Algunos simuladores ofrecen bibliotecas o interfaces de programación propias que permiten controlar robots o acceder a sensores de forma directa mediante llamadas a funciones desde lenguajes como *Python*, *C++* o *Java*.
- **Mensajería basada en middleware:** el *middleware* [9] simplifica el trabajo de los programadores en la compleja tarea de generar las conexiones y sincronizaciones que son necesarias en los sistemas distribuidos mediante sistemas de publicación/suscripción o servicios remotos. Un ejemplo destacado en robótica es *ROS2* y es el método que vamos a utilizar en este proyecto.

2.2 ROS2 y su funcionamiento

ROS2 (Robot Operating System 2) es una infraestructura de *software* diseñada específicamente para facilitar el desarrollo de aplicaciones robóticas complejas. Se trata de un sistema modular que proporciona herramientas, bibliotecas y convenciones destinadas a simplificar el proceso de programación de robots, permitiendo que distintos componentes del sistema interactúen entre sí de forma ordenada, eficiente y escalable.

El funcionamiento de *ROS2* se basa en una arquitectura distribuida compuesta por nodos, que son unidades de ejecución independientes capaces de realizar tareas específicas. Estos nodos pueden comunicarse entre sí de manera flexible a través de diferentes mecanismos de mensajería. Uno de los principales es el sistema de tópicos (*topics*), que permite el intercambio de información de manera asíncrona y desacoplada.

Los tópicos funcionan como canales de datos a los que los nodos pueden publicar o suscribirse. Un nodo puede publicar información en un tópico sin necesidad de saber cuántos nodos la recibirán, y cualquier nodo interesado puede suscribirse a ese tópico para recibir automáticamente los mensajes emitidos. Este modelo de publicador/suscriptor fomenta un diseño modular, donde los componentes del sistema están menos acoplados entre sí, lo que facilita el desarrollo, la escalabilidad y el mantenimiento del *software*.

Cada tópico tiene un nombre único, como */scan*, */odom* o */cmd_vel*, y está asociado a un tipo de mensaje bien definido, que determina la estructura de los datos que pueden enviarse por ese canal. Por ejemplo, un nodo que procesa datos de un sensor *LIDAR 2D* puede publicar en el tópico */scan* utilizando el tipo de mensaje *sensor_msgs/msg/LaserScan*, el cual contiene un conjunto de medidas de distancia y ángulos. Otro nodo encargado del mapeo o la navegación puede suscribirse a ese mismo tópico para procesar dicha información en tiempo real.

Del mismo modo, el tópico */cmd_vel*, comúnmente usado para controlar el movimiento de robots móviles, utiliza el tipo de mensaje *geometry_msgs/msg/Twist*, que especifica velocidades lineales y angulares. La clara separación entre quién genera los datos (publicador) y quién los utiliza (suscriptor), junto con la definición explícita de los tipos de mensaje, permite que distintos nodos interactúen de forma coherente, segura y predecible.

Además de los tópicos, *ROS2* ofrece otros mecanismos de comunicación, como los *servicios*. A diferencia de los tópicos, que son canales unidireccionales y continuos, los servicios permiten una comunicación síncrona, donde un nodo solicita una acción concreta y espera una respuesta. Esta funcionalidad resulta útil para tareas puntuales como obtener una lectura específica de un sensor o ejecutar una acción discreta, como mover un brazo robótico a una posición determinada.

Otro componente esencial en la arquitectura de este *middleware* es el sistema de transformación de coordenadas, conocido como *TF (transform)*. Este sistema permite realizar un seguimiento preciso de la posición y orientación relativa entre distintas partes de un robot, y entre el robot y su entorno, a lo largo del tiempo. Esta funcionalidad es crítica en cualquier sistema robótico móvil, ya que gran parte del procesamiento de sensores y la planificación de movimientos depende de una representación coherente del espacio tridimensional.

En *ROS2*, la versión actualizada de este sistema se denomina *tf2*, y funciona gestionando una estructura jerárquica de marcos de referencia (*frames*), como *base_link*, *odom*, *map*, *camera_link*, *laser_frame*, entre otros. Cada uno de estos *frames* representa un sistema de coordenadas asociado a una parte específica del robot o del entorno. Por ejemplo, el *frame* *base_link* suele estar en el centro del chasis del robot, mientras que *camera_link* indica la posición de una cámara montada en el robot.

El sistema *tf2* permite conocer en cualquier instante cómo se transforma un punto de un *frame* a otro, es decir, calcular la posición relativa entre dos marcos. Esto es fundamental, por ejemplo, para saber dónde está un objeto detectado por una cámara respecto al centro del robot, o para transformar las mediciones de un sensor al sistema de coordenadas global.

Estas transformaciones se publican periódicamente en el tópico */tf*, utilizando el tipo de mensaje *tf2_msgs/msg/TFMessage*. Este mensaje contiene una lista de transformaciones que indican, para cada par de *frames*, la posición (*x*, *y*, *z*) y orientación de uno respecto al otro, junto con una marca de tiempo. Para transformaciones más estáticas (como la posición fija de un sensor respecto al cuerpo del robot), se utiliza el tópico */tf_static*.

A nivel de diseño, *ROS2* también introduce mejoras técnicas que no estaban presentes en su versión anterior. Entre ellas se encuentran la compatibilidad con sistemas de *tiempo real*, una mayor seguridad en las comunicaciones y un mejor soporte para arquitecturas distribuidas, lo que permite que un sistema robótico pueda estar compuesto por múltiples nodos ejecutándose en distintas máquinas y redes, manteniendo una coordinación efectiva entre ellos.

2.3 Paquetes en *ROS2*

Uno de los pilares fundamentales sobre los que se construye *ROS2* es su estructura basada en paquetes. Un paquete en *ROS2* es una unidad lógica que agrupa todo lo necesario para ejecutar una funcionalidad robótica concreta. Esto incluye nodos, *scripts*, archivos de configuración, descripciones del robot (como modelos en *URDF* o archivos de sensores), bibliotecas y otros recursos auxiliares.

Esta organización modular permite reutilizar y compartir componentes fácilmente dentro de la comunidad, ya que cada paquete puede ser desarrollado, versionado y probado de manera independiente. Gracias a esto, proyectos complejos pueden construirse combinando múltiples paquetes que interactúan entre sí para lograr comportamientos sofisticados.

Para que un paquete funcione correctamente dentro del entorno de *ROS2*, necesita algunos elementos mínimos. En primer lugar, debe contar con un archivo *package.xml* que define las dependencias del paquete, su nombre, versión y autores. También es necesario un archivo de compilación *CMakeLists.txt* (o su equivalente si se usa *Python*) que indica cómo debe ser construido el *software*. Además, la lógica funcional se implementa mediante *scripts* o binarios que se ejecutan como nodos, y suelen acompañarse de archivos *.launch.py* que permiten orquestar varios nodos al mismo tiempo, definiendo cómo deben iniciarse y qué parámetros deben emplear.

Los paquetes pueden estar diseñados para múltiples propósitos: algunos se centran en el control de movimiento, otros en el procesamiento de sensores, en la navegación autónoma, en la visualización de datos o en la simulación de entornos. Por ejemplo, existe un paquete específico para la navegación 2D (*nav2*) y también otros para navegación 3D mediante *SLAM 3D*. Hay paquetes de visualización como *rviz2*, y herramientas de depuración y análisis como *rqt*.

Una característica destacada de los paquetes en *ROS2* es su capacidad para adaptarse a diferentes entornos. Es decir, un mismo paquete puede ser utilizado tanto en un entorno simulado como en un robot físico, simplemente modificando los archivos de configuración o los parámetros de ejecución. Esto permite que el desarrollo y la validación se realicen primero en simuladores, para luego pasar al entorno real con los mínimos cambios posibles. También se pueden usar los mismos paquetes en distintos simuladores como *Gazebo*, *O3DE* o *CoppeliaSim*, siempre que se ajusten correctamente los parámetros de configuración del paquete.

Esta flexibilidad convierte a los paquetes en bloques reutilizables y portables, fundamentales para el desarrollo eficiente y escalable de sistemas robóticos. Gracias a ellos, el ecosistema *ROS2* cuenta con una gran variedad de soluciones listas para usar, muchas de las cuales son mantenidas por una activa comunidad de desarrolladores y centros de investigación.

2.4 *SLAM 3D* en *ROS2*

El término *SLAM* (*Simultaneous Localization and Mapping*) hace referencia a la capacidad de un robot de crear un mapa del entorno mientras se localiza a sí mismo dentro de él, todo al mismo

tiempo y de forma autónoma. Cuando hablamos específicamente de *SLAM 3D*, nos referimos a esta misma técnica aplicada en tres dimensiones, lo cual permite representar entornos complejos y realistas, incluyendo elementos verticales como escaleras, estanterías o desniveles, que no pueden capturarse adecuadamente con modelos en *2D*.

En el ecosistema de *ROS2*, el *SLAM 3D* se implementa habitualmente a través de paquetes especializados, como *RTAB-MAP*, *Lidarslam_ros2*, o *MOLA*, entre otros. Estos paquetes ofrecen nodos que se encargan de recibir datos en tiempo real desde sensores como cámaras *RGBD*, *LIDAR 3D* o estereocámaras, y procesarlos para generar mapas tridimensionales y estimaciones precisas de la posición del robot.

Estos paquetes funcionan sobre la arquitectura distribuida de *ROS2*, intercambiando información a través de *topics*, utilizando *services* para consultas específicas (por ejemplo, guardar un mapa o reiniciar el proceso), y manteniendo relaciones espaciales mediante la librería *tf* para representar las posiciones relativas entre el robot y su entorno. En resumen, integran completamente los fundamentos de *ROS2* para cumplir su propósito.

Un aspecto muy potente del *SLAM* en *ROS2* es que puede utilizarse tanto en entornos físicos como en simulados, con cambios mínimos. Esto se logra gracias a la abstracción que ofrecen los paquetes: mientras se reciban datos con el formato adecuado (por ejemplo, una nube de puntos desde un sensor *LIDAR 3D*), el algoritmo funcionará independientemente de si los datos provienen de un sensor real o de un simulador. Lo único necesario es adaptar los archivos de configuración del paquete, definiendo correctamente los parámetros del sensor, el *frame* de referencia (*base_link*, *odom*, *map*, etc.), y las frecuencias de publicación de los datos.

Además, estos paquetes suelen ser altamente configurables, permitiendo ajustar factores como el algoritmo de optimización, la frecuencia de actualización del mapa, los límites de detección del sensor o las estrategias de gestión de bucles cerrados (*loop closure*). Esto brinda al desarrollador un control fino sobre el comportamiento del paquete, lo que resulta especialmente útil en entornos complejos o cambiantes.

3. Planificación

En este capítulo se detallaran los objetivos que persigue este trabajo y la planificación de las tareas que estos llevan. Además se mostrará las desviaciones respecto de la planificación inicial siendo relevante en esta parte el paso de la implementación de los robots en paralelo a la implementación en serie.

3.1 Objetivos

El objetivo principal de este trabajo es analizar la viabilidad del uso de *CoppeliaSim* y *O3DE* como entornos de simulación para sistemas robóticos móviles integrados con *ROS2*. Para ello, se plantea una evaluación comparativa (usando dos de los robots de Robotnik) que permita valorar hasta qué punto estas plataformas emergentes pueden constituir una alternativa real a los simuladores tradicionalmente empleados en el ámbito de la robótica móvil.

Con este fin, se establecen los siguientes objetivos específicos:

- Implementar dos robots virtuales, *Rb_Watcher* y *Rb_Robout* de la empresa *Robotnik Automation*, en cada uno de los simuladores seleccionados, configurando adecuadamente sus sensores —como cámaras *RGB-D* y *LIDAR 2D* y *3D*— y asegurando una integración funcional con *ROS2*, tanto a nivel de publicación como de suscripción de datos entre nodos y simuladores.
- A partir de la implementación de los robots virtuales, valorar el grado de integración con *ROS2* que ofrece cada simulador, analizando las funcionalidades compatibles y el esfuerzo requerido para su implementación.
- Comparar el rendimiento de ambos robots en los simuladores mediante el uso de métricas objetivas tales como el tiempo de arranque (*startup*), el *real-time factor* (*RTF*), los fotogramas por segundo (*FPS*), el consumo de recursos (*RAM*, *CPU* y *GPU*) y la latencia de los sensores.
- Determinar si los simuladores pueden utilizar paquetes de *SLAM 3D* compatibles con *ROS2* y en qué medida es posible integrarlos con la implementación del robot en cada simulador.

- En caso afirmativo del objetivo anterior, comparar los resultados con otros simuladores más compatibles con *ROS2*.

Para asegurar una comparación justa entre plataformas y facilitar la reproducibilidad del estudio, se ha desarrollado escenarios virtuales simples y estructurados, fácilmente replicables en cada uno de los simuladores analizados. Estos entornos permiten evaluar el comportamiento de los sistemas bajo condiciones equivalentes, minimizando variables externas y centrando el análisis en las capacidades técnicas propias de cada plataforma.

3.2 Planificación Inicial

La planificación inicial del proyecto se diseñó con el objetivo de desarrollar de forma simultánea y paralela la implementación de los robots en los entornos de *CoppeliaSim* y *O3DE*, como se puede observar en la figura 3.1 . Se pretendía realizar las mismas tareas en ambos simuladores antes de pasar a la siguiente tarea. El primer paso consistiría en la instalación y configuración básica de ambos simuladores, asegurando que estuvieran correctamente preparados para la importación de modelos y la comunicación con *ROS2*. Esta fase inicial de *setup* incluiría verificar la compatibilidad con los *plugins* necesarios, así como realizar pruebas sencillas para garantizar que ambos entornos funcionaban adecuadamente. Una vez que los simuladores estuvieran instalados y configurados, la tarea siguiente consistiría en la importación de los robots desde archivos *URDF* en ambos simuladores. En esta fase, se pretendía incorporar todos los elementos posibles del robot, tales como los eslabones, articulaciones, mallas, colisiones, sensores y controladores, para tener los robots lo más completos posibles dentro de cada simulador. Sin embargo, los robots importados no eran funcionales debido a que ni *O3DE* ni *CoppeliaSim* reconocían los sensores ni los controladores del archivo *URDF*. Después de tener los robots importados, la planificación establecía que se añadirían los sensores en las posiciones adecuadas dentro del modelo, integrándolos con *ROS2* para permitir la simulación de sus datos. Esta fase era esencial para lograr que el robot simulado tuviera capacidades sensoriales reales, necesarias para futuras pruebas de navegación y *SLAM*. Posteriormente, se desarrollaría el control del robot, empleando el tópico */cmd_vel* para enviar comandos de movimiento y comprobar que el robot podía desplazarse correctamente dentro del entorno simulado. Una vez conseguido el control básico, el plan continuaba con la creación de escenarios simples, como un suelo vacío y entornos con paredes y obstáculos básicos, que servirían para evaluar el rendimiento de los robots en ambos simuladores. Esta evaluación incluiría medir aspectos técnicos como la latencia de sensores, uso de recursos y estabilidad, con el fin de detectar posibles limitaciones. Con base en estos resultados, se decidiría si algún simulador presentaba inconvenientes que justificaran descartarlo para las pruebas más avanzadas, como las relacionadas con *SLAM 3D*. Finalmente, tras validar el entorno y el robot, la planificación preveía la instalación y configuración de varios paquetes de *SLAM 3D* reconocidos, adaptándolos para su uso con los sensores de los robots en cada simulador. El objetivo sería realizar pruebas comparativas en escenarios controlados, evaluando la calidad de los mapas generados y la eficiencia de los algoritmos, en paralelo con simuladores como *Webots* y *Gazebo Ignition* para tener un referente.

3.3 Desviaciones del plan inicial

Durante la primera semana, sin embargo, se presentó una desviación significativa respecto a lo previsto. Mientras que la instalación y configuración de *CoppeliaSim* se realizó sin mayores problemas, la instalación de *O3DE* se complicó. Inicialmente se intentó utilizar una versión precompilada del motor, que en teoría debería haber soportado los *plugins* necesarios para *ROS2*, pero la integración fallaba sin que se encontrara fácilmente la causa. Para resolverlo, fue necesario investigar profundamente el proceso de compilación del motor y finalmente se optó por instalar

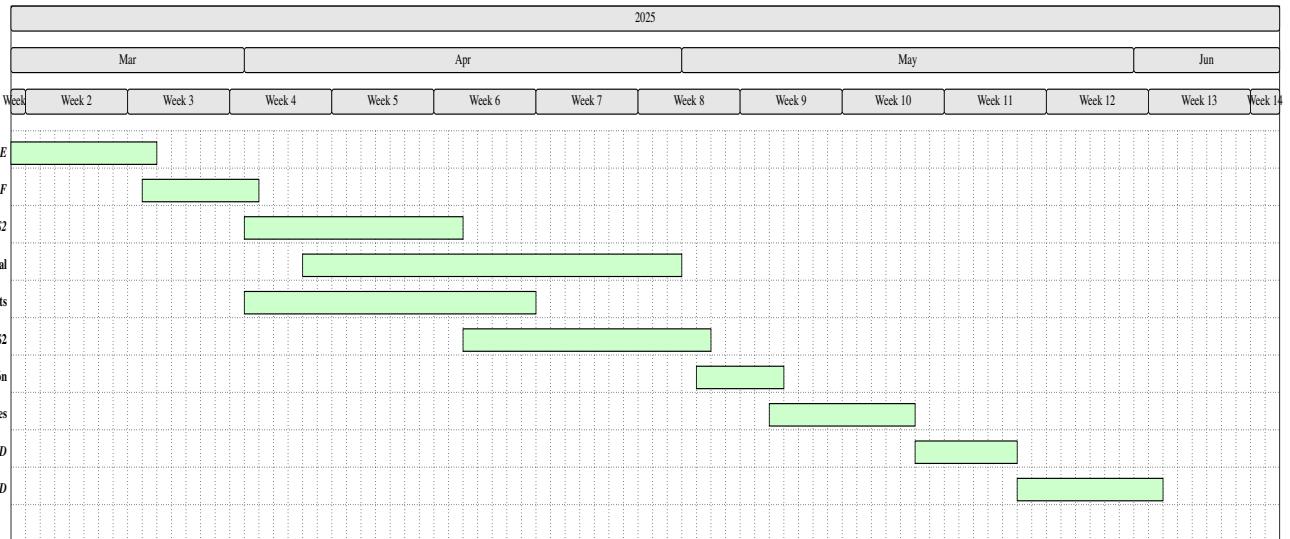


Figura 3.1: Diagrama de Gantt inicial.

O3DE compilándolo directamente desde el código fuente del repositorio oficial en *GitHub*. Esta tarea implicó una revisión exhaustiva de los archivos de configuración de *CMake* y las dependencias, consumiendo una cantidad considerable de tiempo.

Adicionalmente, durante la importación de los robots, se llevó a cabo una investigación sobre los archivos *URDF* y *XACRO* empleados en simuladores como *Gazebo*, con la intención de mejorar la compatibilidad y lograr que tanto *CoppeliaSim* como *O3DE* reconocieran más elementos del modelo, más allá de los eslabones y articulaciones. A pesar de los esfuerzos, no se consiguió ampliar el reconocimiento más allá de estos componentes básicos, por lo que se decidió avanzar con esta limitación en mente.

Otra dificultad inesperada surgió durante la fase de implementación en *CoppeliaSim*. Para programar los sensores y obtener datos del simulador fue necesario emplear el lenguaje *Lua*, con el que no se tenía experiencia previa. Por este motivo, se dedicó aproximadamente una semana a familiarizarse con este lenguaje, realizando pruebas y *scripts* para manejar la comunicación con *ROS2* y la obtención de información sensorial.

Debido a estos contratiempos, la planificación original se modificó. En lugar de trabajar en paralelo con ambos simuladores desde el inicio, se decidió importar y configurar completamente el robot en un simulador antes de comenzar con el otro. Esta estrategia permitió un avance más ordenado y eficiente, permitiendo llegar a tiempo a los objetivos de las tareas inicialmente planeadas. Por otro lado, algunos objetivos secundarios, como la creación de una escena realista industrial, se descartaron por falta de tiempo, priorizando las tareas esenciales para la simulación y evaluación de los robots. Antes de proceder con la evaluación del rendimiento de los simuladores, se añadió una tarea no prevista originalmente: la validación de una guía de implementación de robots en otro simulador elaborada por un compañero. Esta actividad consistió en seguir las instrucciones de dicha guía y añadir dos robots adicionales en el entorno propuesto, con el fin de comprobar su aplicabilidad y detectar posibles mejoras. Esta tarea aportó una visión externa del proceso y enriqueció la evaluación final al incorporar criterios de usabilidad y claridad documental.

Posteriormente, tras completar el *Setup* de los paquetes de *SLAM 3D*, surgió otra desviación importante. En algunos simuladores se evidenció la necesidad de realizar implementaciones adicionales para lograr la funcionalidad esperada. Por ejemplo, en *Webots*, el sensor *LIDAR 3D* no incluía el campo de intensidad, lo que resultaba problemático para ciertos algoritmos. Para solucionarlo, se desarrolló un nodo en *ROS2* que generaba este campo y lo añadía a los mensajes

sensor_msgs/PointCloud2.

También se presentaron dificultades con la visualización de los mapas generados por los diferentes paquetes , ya que cada uno exportaba en formatos distintos. Para homogeneizar este aspecto, se decidió convertir los mapas al formato *.ply* y utilizar la librería *Open3D* para *Python* como visor unificado. Asimismo, en el caso de *Gazebo Ignition*, algunos mapas contenían valores infinitos que impedían su visualización, por lo que fue necesario implementar un filtro de postprocesamiento que eliminara esos valores antes de exportarlos.

En cronograma de la figura 3.2 se puede observar las tareas planeadas inicialmente frente a las tareas que se realizaron realmente.

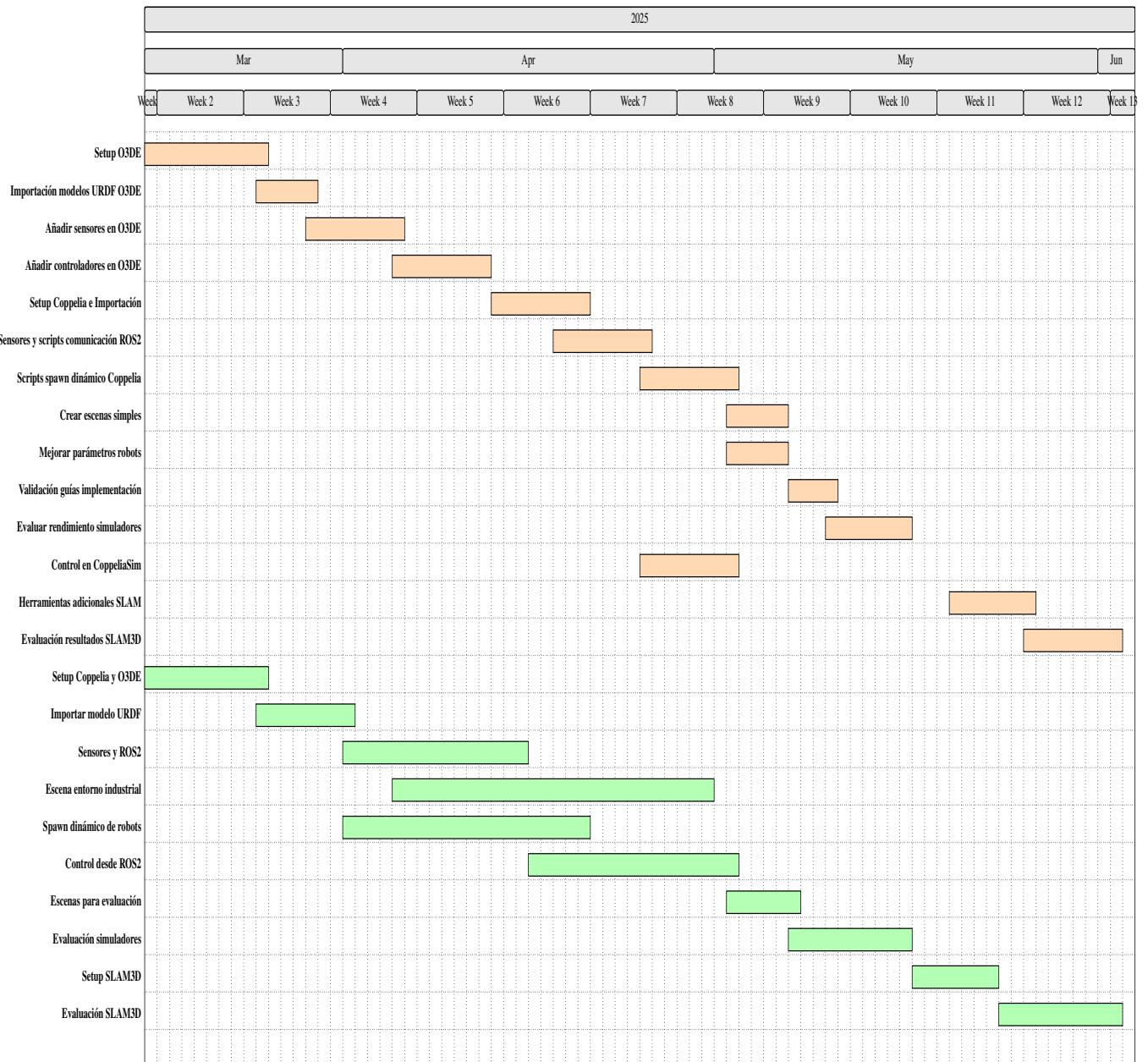


Figura 3.2: Diagrama comparativo entre la planificación inicial y las tareas realmente ejecutadas. Las barras verdes representan las tareas planificadas inicialmente, mientras que las barras naranjas corresponden a las actividades que realmente se llevaron a cabo.

3.4 Estimación de costes

En esta sección se calcula los costes totales de este proyecto desglosándlos en costes de las horas de trabajo humano, costes *hardware*, costes *software* y otros gastos operativos.

3.4.1 Coste asociado a las horas de trabajo en el desarrollo del proyecto

Este apartado detalla el coste asociado a las horas de trabajo necesarias para la realización completa del proyecto. Se ha considerado la participación de dos perfiles. Para ello se ha partido de los valores de salarios orientativos del portal *tecnoempleo.com* [11] durante el año 2025. Para el salario del realizante del proyecto, al ser un estudiante y al no tener experiencia en el sector, se le ha aplicado el salario mínimo anual correspondiente a un "Programador Junior". En cambio, para el del supervisor se ha calculado la media del salario de un "Jefe de Equipo" al tener una experiencia mayor a 8 años en la empresa.

- **Programador junior:** coste horario mínimo derivado de un salario anual de 34 700€. Equivale a 133,50€/día (260 días/año). Asumiendo una jornada de 8 h, el coste horario mínimo es:

$$\frac{133,50\text{€}}{8\text{h}} \approx 16,69\text{€/h.}$$

- **Supervisor senior:** coste horario medio derivado de un salario anual medio de 51 100€. Equivale a 196,54€/día (260 días/año). Asumiendo una jornada de 8h, el coste horario medio es:

$$\frac{196,54\text{€}}{8\text{h}} \approx 24,57\text{€/h.}$$

A continuación en la tabla 3.1 , se presenta la distribución de las horas dedicadas a cada fase y el coste resultante:

Fase	Horas	Coste (€)
Implementación de los robots en <i>O3DE</i>	90	$90 \times 16,69 = 1502,10$
Implementación de los robots en <i>CoppeliaSim</i>	140	$140 \times 16,69 = 2336,60$
Evaluación de los robots	10	$10 \times 16,69 = 166,90$
<i>Setup</i> de paquetes <i>SLAM 3D</i>	50	$50 \times 16,69 = 834,50$
Evaluación de mapas 3D	12	$12 \times 16,69 = 200,28$
Documentación	50	$50 \times 16,69 = 834,50$
Reuniones con el supervisor	7,5	$7,5 \times 24,57 = 184,28$
Total trabajo humano	359,5	6059,16

Cuadro 3.1: Desglose de los gastos de trabajo humano en cada fase del proyecto.

Estos valores permiten dimensionar de forma precisa el presupuesto de personal para este proyecto.

3.4.2 Coste proporcional del *hardware*

Este apartado recoge el coste proporcional del *hardware* empleado durante el desarrollo del proyecto, tabla 3.2. Aunque el equipamiento ya se encontraba disponible, se ha realizado una estimación del gasto imputable a partir de su valor total, considerando su vida útil y el tiempo de uso en este proyecto concreto.

Para ello, se ha adoptado el criterio habitual de amortización lineal, dividiendo el precio total del equipo entre su vida útil (en meses), y multiplicando el resultado por los meses utilizados. En este caso, se ha asumido una vida útil de 5 años (60 meses), y un periodo de uso de 3 meses.

Hardware	Precio (€)	Vida útil (meses)	Uso (meses)	Coste proporcional (€)
Ordenador portátil	1.200	60	3	60
Ordenador de sobremesa	1.800	60	3	90
Total hardware				150

Cuadro 3.2: Desglose de los gastos de *hardware* necesario para el proyecto.

3.4.3 Coste asociado al *software* utilizado en el proyecto

En este apartado se detalla el *software* empleado a lo largo del desarrollo del proyecto. Aunque no se ha incurrido en ningún gasto económico directo, ya que todos los paquetes y herramientas utilizadas son de código abierto o cuentan con licencias gratuitas, su inclusión es relevante por su papel clave en la arquitectura y ejecución del sistema. Podemos ver el desglose en la tabla 3.3 .

Software / Paquete	Licencia	Descripción / Uso	Coste (€)
<i>ROS2 Humble</i>	<i>Open Source</i>	Infraestructura de comunicación robótica en tiempo real.	0
<i>CoppeliaSim EDU</i>	Gratis	Simulador 3D utilizado para modelar robots y entornos.	0
<i>O3DE</i>	<i>Open Source</i>	Simulador alternativo evaluado durante la fase inicial.	0
<i>Visual Studio Code</i>	<i>Open Source</i>	Entorno de desarrollo para edición de código.	0
<i>Ubuntu 22.04 LTS</i>	<i>Open Source</i>	Sistema operativo base del entorno de trabajo.	0
<i>RTAB-MAP</i>	<i>Open Source</i>	Paquete <i>SLAM 3D</i> .	0
<i>Lidarslam_ros2</i>	<i>Open Source</i>	Paquete <i>SLAM 3D</i>	0
<i>MOLA</i>	<i>Open Source</i>	Paquete <i>SLAM 3D</i>	0
Total software			0

Cuadro 3.3: Desglose de los gastos de *software* necesario para el proyecto.

3.4.4 Otros gastos operativos

Aquí se incluyen costes adicionales derivados del desarrollo, como el consumo energético, desplazamientos y conectividad.

Consumo energético

Estimación del coste eléctrico durante 3 meses de trabajo (~3 h/día de media):

- **Portátil (90W):**

$$90\text{ W} \times 3\text{ h} \times 90\text{ días} \approx 24,3\text{ kWh} \rightarrow 6,08 \text{ (a } 0,25 \text{ €/kWh)}$$

- **Sobremesa (400W):**

$400\text{ W} \times 3\text{ h} \times 90\text{ días} \approx 108\text{ kWh} \rightarrow 27$ (a 0,25 €/kWh)

Desplazamientos y comidas

Durante la fase de implementación en la empresa *Robotnik*, fue necesario acudir presencialmente en 18 ocasiones (~90 km solo ida). Se consideran:

- 18 desplazamientos \times 20 € (combustible): 360 €
- 18 comidas \times 10 €: 180 €

Conectividad

Para garantizar acceso a documentación, herramientas en la nube y reuniones *online*, se ha estimado un coste mensual de 60 € en internet durante 3 meses: 180 €

Otros servicios

Se estima un coste simbólico por el uso parcial de las instalaciones de *Robotnik* (electricidad, espacio y red): 100 €

A continuación en la tabla 3.4 se resumen los gastos operativos adicionales.

Concepto	Detalle	Coste (€)
Energía (portátil)	24,3 kWh a 0,25 €/kWh	6,08
Energía (sobremesa)	108 kWh a 0,25 €/kWh	27,00
Desplazamientos (18 \times 20 €)	Visitas presenciales a <i>Robotnik</i>	360,00
Comidas asociadas (18 \times 10 €)	Dietas en jornada completa	180,00
Internet (3 meses \times 60 €)	Conectividad	180,00
Uso parcial oficina <i>Robotnik</i>	Estimación	100,00
Total otros gastos		853,08

Cuadro 3.4: Desglose de otros gastos operativos.

3.4.5 Presupuesto total del proyecto

Categoría	Coste (€)
Trabajo humano	6.059,16
Hardware	150,00
Software	0,00

Otros gastos operativos	853,00
Total estimado	7062.16

Cuadro 3.5: Desglose de presupuesto total del proyecto

Como podemos observar en la tabla 3.5 el coste total asciende a 7062.16 euros siendo el principal coste el del trabajo humano y el de menor coste el *software*.

4. Análisis

En este capítulo se presenta un análisis detallado de los robots implementados y los requisitos que guiaron el diseño del sistema. En primer lugar, se describen las características de los dos robots utilizados, *Rb_Watcher* y *Rb_Robout*, incluyendo sus componentes y configuraciones específicas. A continuación, se abordan los requisitos funcionales y no funcionales, tanto de los simuladores como de los paquetes de localización y mapeo (*SLAM 3D*). Posteriormente, se introduce un diagrama de actividades que representa el flujo general realizado para la evaluación de las capacidades de los robots *Robotnik* en los simuladores en los que se han integrado, seguido de la arquitectura de simulación robótica implementada en *CoppeliaSim* y *O3DE*.

4.1 Descripción de los 2 robots implementados

4.1.1 *Rb_Watcher*



Figura 4.1: Imagen del Rb_Watcher real.

El *Rb_Watcher* [12] es un robot móvil autónomo diseñado para tareas de vigilancia y seguridad en entornos interiores y exteriores. Su estructura compacta, con unas dimensiones, como se puede observar en la figura 4.2 de $736 \times 614 \times 894$ mm y un peso de 73 kg, le proporciona robustez y

movilidad. Está protegido con una certificación *IP54*, lo que le permite operar en ambientes con presencia de polvo y salpicaduras.

Desde el punto de vista cinemático, utiliza un sistema de tracción *skid steering* con cuatro motores de 500 W con freno, que le permiten alcanzar velocidades de hasta 2,5 m/s y superar pendientes de hasta el 80 %.

Para la percepción del entorno, incorpora sensores avanzados como una cámara RGB-D para visión 3D, un sensor *LIDAR 3D* para detección de obstáculos y mapeo, una cámara bi-espectral *Pan-Tilt-Zoom* para inspecciones detalladas, un sistema de posicionamiento global (*GPS*) de alta precisión y una unidad de medida inercial (*IMU*) para la estimación del movimiento y la orientación. Además, cuenta con balizas *LED* de 360° para iluminación, así como micrófono y altavoces con protección *IP65*, destinados a comunicaciones y emisión de alertas.

Gracias a estos sistemas, el robot está orientado a aplicaciones de seguridad y vigilancia autónoma, detección de intrusos, inspección de anomalías térmicas y recolección de datos en sectores como la energía, la construcción o el entorno industrial.

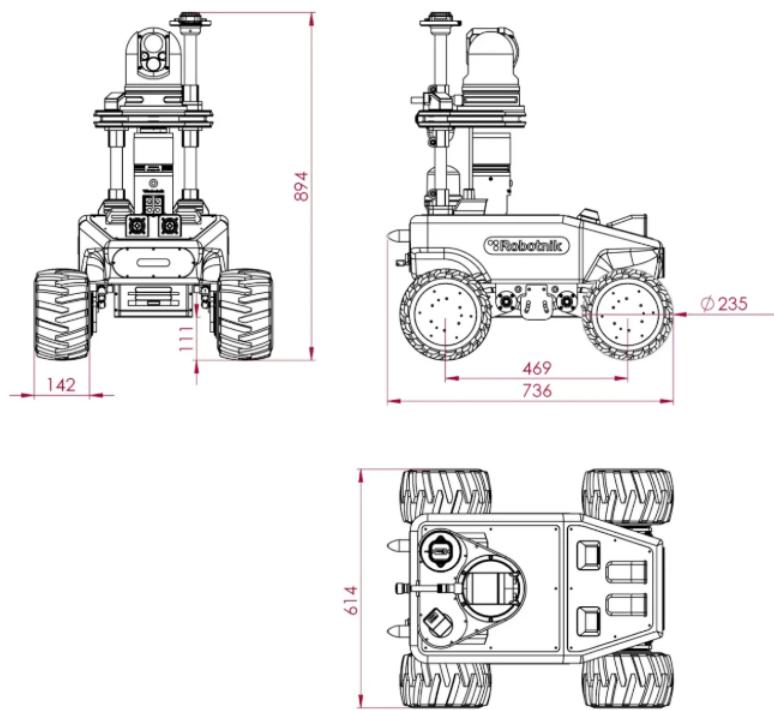


Figura 4.2: Planos del robot Rb_Watcher.

4.1.2 Rb_Rabout



Figura 4.3: Imagen del robot Rb_Rabout.

El *Rb_Rabout* [13] es un robot móvil autónomo *AMR* (*Autonomous Mobile Robot*) diseñado específicamente para tareas de logística interna y transporte de cargas pesadas en entornos industriales de interior, como fábricas o almacenes.

En cuanto a su estructura, presenta un diseño compacto y de baja altura, con unas dimensiones, como se puede observar en la figura 4.4, de $1.758 \times 891 \times 372$ mm y un peso total de 350 kg. Su chasis está construido en acero y dispone de una plataforma plana superior que permite transportar cargas directamente sobre su superficie o mediante carros adaptados.

Desde el punto de vista cinemático, emplea una configuración omnidireccional basada en cuatro ruedas *mecanum* motorizadas.

Está preparado para operar de forma autónoma y segura en entornos compartidos con personas. Para ello, incorpora un conjunto de sensores integrados que permiten una navegación inteligente y colaborativa. Este sistema de percepción incluye: dos escáneres láser 2D para la detección de obstáculos y delimitación de zonas seguras; dos cámaras *RGB-D*, una frontal y otra trasera, que amplían las capacidades de visión y permiten aplicaciones basadas en percepción 3D; una *IMU*, fundamental para la estimación de la orientación y estabilidad del movimiento; un *router 5G* para comunicaciones de baja latencia; y un sistema de seguridad con *Programmable Logic Controller (PLC)* dedicado.

Gracias a su capacidad de carga, navegación autónoma y sensores de seguridad, este robot es utilizado principalmente en el transporte de materiales pesados en interiores, la automatización de procesos logísticos, la carga y descarga automatizada, y la integración en líneas de producción industriales.

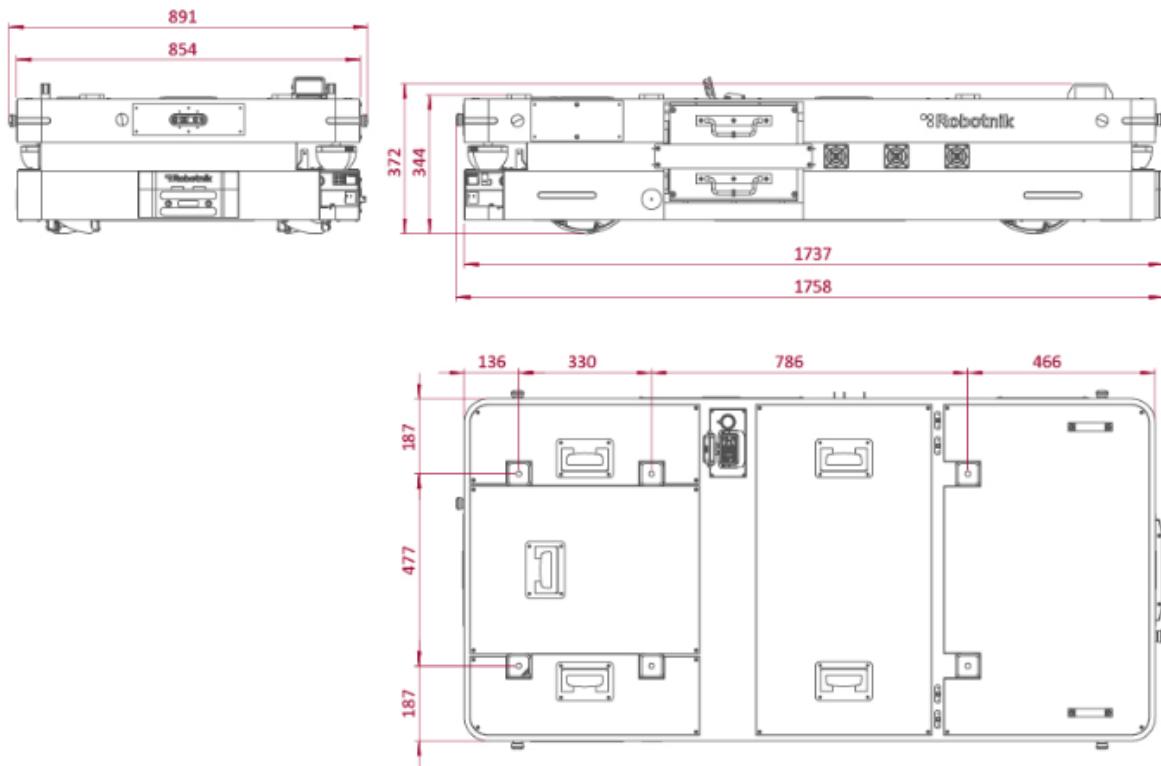


Figura 4.4: Planos del robot Rb_Robout.

4.2 Requisitos funcionales

Antes de abordar el desarrollo e implementación de un sistema robótico , resulta fundamental establecer de manera clara los requisitos funcionales. Estos definen el conjunto de comportamientos, capacidades y restricciones que deben cumplirse para garantizar el correcto funcionamiento del sistema, así como su utilidad práctica en el contexto previsto. A continuación, se presentarán los requisitos funcionales de los simuladores y de los paquetes de *SLAM 3D*.

4.2.1 Requisitos funcionales de los simuladores

En el contexto de este trabajo, se hace necesario evaluar si los simuladores ofrecen una integración sólida con *ROS2* y permiten desarrollar, probar y validar algoritmos de *SLAM 3D* en entornos controlados. Para ello, se han definido una serie de requisitos funcionales esenciales. Estos requisitos buscan asegurar que la simulación represente de forma precisa y sincronizada la operación de un robot real, permitiendo que cualquier solución desarrollada sea fácilmente transferible a un sistema físico. Los requisitos funcionales que debe cumplir un simulador robótico adecuado para este estudio son los siguientes:

- **Publicación de sensores virtuales en formato *ROS2*:** los sensores simulados (*LIDAR*, cámaras, *IMU*, *GPS*) deben generar mensajes en los formatos estándar definidos por *ROS2*.
- **Publicación de transformaciones *TF*:** es indispensable que el simulador publique transformaciones entre marcos de coordenadas ($/tf$) para que el sistema pueda interpretar correctamente la posición relativa entre los distintos componentes del robot.

- **Simulación de movimiento:** el simulador debe ser capaz de recibir mensajes *geometry_msgs/Twist* en el tópico */cmd_vel* y aplicarlos correctamente al modelo del robot para permitir su desplazamiento.
- **Control de múltiples sensores sincronizados:** debe soportar la simulación de varios sensores funcionando simultáneamente y en sincronía temporal, un aspecto clave para algoritmos como *SLAM* o navegación autónoma.
- **Carga dinámica de robots y entornos:** se espera que el simulador permita cargar distintos modelos de robot y mapas sin necesidad de reiniciar el entorno completo. Esto mejora la eficiencia durante las pruebas iterativas.

4.2.2 Requisitos funcionales de los paquetes *SLAM 3D*

Dado el objetivo de este proyecto de comparar diferentes algoritmos de *SLAM 3D* en un entorno simulado y controlado, se establecen una serie de requisitos funcionales que todo paquete considerado debe cumplir para poder ser incluido en la evaluación. Estos requisitos aseguran que los algoritmos seleccionados puedan ejecutarse bajo condiciones homogéneas y producir resultados comparables, maximizando así la validez del análisis. Los requisitos específicos definidos para este trabajo son:

- **Entrada desde sensores simulados:** el algoritmo debe aceptar como entrada los datos generados por sensores *LIDAR 3D* simulados en *Ignition Gazebo*, *Webots* y *O3DE* sin necesidad de modificar la arquitectura del simulador.
- **Salida en forma de mapa 3D exportable:** el paquete debe ser capaz de generar un mapa tridimensional del entorno y permitir su exportación a un formato común y procesable desde un único *software* externo de visualización y análisis (por ejemplo, *.pcd*, *.ply* o *.obj*). Esto es fundamental para poder comparar los resultados obtenidos por cada algoritmo de forma objetiva.

4.3 Requisitos no funcionales

En este proyecto, los requisitos no funcionales son fundamentales para garantizar que tanto los robots en los simuladores como los paquetes *SLAM 3D* funcionen de forma estable, eficiente y comparable. Estos requisitos no describen las funciones del sistema, sino las condiciones bajo las cuales debe operar para asegurar resultados fiables y reproducibles.

4.3.1 Requisitos no funcionales de los robots en los simuladores

Para garantizar que los robots virtuales en los distintos simuladores puedan desempeñar adecuadamente cualquier función para la que sean requeridos en el proyecto, es fundamental que cumplan ciertos requisitos no funcionales comunes. Estos requisitos aseguran que los robots mantengan características y comportamientos equivalentes en cuanto a sensores, movilidad, capacidad de interacción con el entorno y compatibilidad con los sistemas de control. La uniformidad en estas características permite que los resultados obtenidos en los diferentes simuladores sean comparables y fiables, independientemente de las tareas específicas que se realicen con los robots.

- **Homogeneidad entre simuladores:** los modelos deben tener dimensiones y configuraciones sensoriales equivalentes en ambos simuladores para evitar sesgos debidos a diferencias en el *hardware* virtual. Por ejemplo, si el robot en *CoppeliaSim* dispone de un sensor *LIDAR* con cierto rango y resolución, el robot en *O3DE* debería contar con características similares. Esto evita que las diferencias en los resultados del rendimiento de cada simulador se deban al *hardware* virtual .

- **Movilidad homogénea:** se priorizará un tipo de control uniforme, como un control mecanum, para facilitar la modelización y el control en ambos entornos. Esto garantiza que las diferencias observadas no se deban a distintos modelos de movimiento.
- **Compatibilidad con *ROS2 Humble*:** los robots deben poder integrarse y controlarse mediante *ROS2 Humble*.
- **Interoperabilidad con los simuladores:** los robots deben integrarse sin problemas en ambos simuladores, aprovechando los *plugins* o interfaces que cada uno ofrece. Esto facilita la automatización y evita tener que realizar adaptaciones manuales para cada entorno.

4.3.2 Requisitos no funcionales de los paquetes *SLAM 3D*

Los paquetes seleccionados deben cumplir una serie de requisitos no funcionales que aseguren su buen desempeño y compatibilidad dentro del proyecto:

- **Estabilidad y rendimiento en tiempo casi real:** los algoritmos deben ejecutarse sin interrupciones, manteniendo sincronía con los simuladores y evitando desfases que puedan comprometer la calidad de los resultados obtenidos.
- **Configurabilidad:** los parámetros clave deben poder ajustarse fácilmente para adaptar el algoritmo a diferentes escenarios y necesidades experimentales, facilitando la optimización del rendimiento.
- **Documentación y ejemplos:** se requiere una documentación básica que facilite la instalación, configuración y uso del paquete, incluyendo ejemplos prácticos que permitan el lanzamiento rápido del sistema en distintos entornos.

4.4 Diagrama de Actividades

El diagrama de actividades de la figura 4.5 representa de manera estructurada el flujo de trabajo y las tareas principales llevadas a cabo durante este proyecto.

En primer lugar, se realiza el *setup* de dos simuladores seleccionados: *O3DE* y *CoppeliaSim*. En cada uno de estos entornos, se importan los modelos de los robots en formato *URDF*, se añaden y configuran los sensores necesarios, y se establece la comunicación de estos sensores con *ROS2*, en caso de que no estén integrados por defecto a través de los *plugins* disponibles. Posteriormente, se implementa el control de los robots para permitir su operación dentro de los simuladores.

Tras esta configuración inicial, se evalúa el rendimiento, la capacidad multirrobot y la integración de cada simulador con *ROS2*, así como el rendimiento de cada tipo de robot. Aquellos simuladores que cumplen con los requisitos técnicos para el *SLAM 3D* son utilizados para ejecutar tres paquetes específicos: *RTAB-MAP*, *Lidarslam_ros2* y *MOLA*. Por otro lado, los simuladores que no demuestran ser aptos para esta tarea son descartados para evitar un uso ineficiente de recursos.

De manera paralela, se realizan pruebas de los mismos algoritmos en dos simuladores adicionales, *Gazebo Ignition* y *Webots*, donde los robots ya estaban previamente implementados y configurados, facilitando así la ejecución directa de las pruebas.

Finalmente, se procede a un análisis comparativo de los resultados obtenidos en todos los simuladores evaluados, lo que permite identificar fortalezas y debilidades de cada plataforma. Este análisis es la base para la corrección de errores y la realización de mejoras, con el objetivo de optimizar el rendimiento y la precisión de los sistemas de navegación basados en *SLAM 3D*.

4.5 Arquitectura de simulación robótica para *CoppeliaSim*

La arquitectura propuesta para el sistema de simulación y control de robots móviles en *CoppeliaSim*, figura 4.6, se estructura en una serie de capas funcionales que interactúan entre sí mediante el *middleware ROS2*. Su diseño modular permite una integración clara entre los componentes de simulación, comunicación y automatización, sentando las bases para experimentos en entornos complejos y con múltiples robots virtuales.

En el nivel de simulación, el entorno principal es *CoppeliaSim*, el cual se encarga de representar físicamente y visualmente los robots y su entorno de operación. La simulación física de los modelos de los robots en la escena es gestionada por el motor *Bullet v2.78*, que asegura un comportamiento realista del sistema mediante la modelación de masas, fricciones, colisiones y fuerzas.

Cada robot está controlado internamente por un *script* principal en lenguaje *LUA*, el cual cumple varias funciones fundamentales. Por un lado, se encarga de publicar las transformaciones entre marcos de referencia en el tópico */tf*, proporcionando información espacial clave para el sistema *ROS2*. Por otro, actúa como suscriptor al tópico */cmd_vel*, por el cual recibe comandos de velocidad lineal y angular que son convertidos, mediante cinemática inversa *mecanum*, en órdenes individuales para cada rueda del robot. Este procesamiento se realiza teniendo en cuenta que los robots utilizan una configuración de cuatro ruedas *mecanum* que permite movimientos omnidireccionales en el plano.

Además del *script* principal, cada sensor virtual del robot (como el *LIDAR 2D* o la *IMU*) es gestionado por su propio *script* *LUA* independiente. Estos *scripts* extraen los datos directamente del simulador, los formatean como mensajes *ROS2* válidos, y los publican en los tópicos correspondientes. Para asegurar la separación de los datos cuando hay múltiples robots, cada *script* consulta un archivo externo que indica la cantidad de robots activos y genera un *namespace* único por instancia, es decir, un nombre distintivo para los tópicos, *frames* y parámetros correspondientes a cada robot.

La conexión entre *CoppeliaSim* y *ROS2* se realiza mediante el *plugin* oficial *coppeliaSim_ros2*, que permite a los *scripts* *LUA* actuar como nodos *ROS2* ligeros, publicando y suscribiéndose a tópicos en tiempo real. Aunque el *plugin* reconoce los tipos de mensajes estándar, los mensajes deben construirse manualmente desde los *scripts* *LUA*, garantizando un control total sobre el formato de los datos transmitidos.

En la capa intermedia del sistema se encuentra *ROS2*, que actúa como el sistema de comunicación entre los módulos de simulación, percepción y teleoperación. En esta capa se centraliza el intercambio de mensajes mediante tópicos como */scan* (para el *LIDAR 2D*), */odom* (odometría), */imu* (datos iniciales), */tf* (transformaciones espaciales) y */cmd_vel* (comandos de movimiento). Uno de los nodos clave en esta capa es el nodo de teleoperación, que interpreta las entradas del usuario —por ejemplo, desde el teclado— y genera los mensajes adecuados en */cmd_vel*, permitiendo un control manual e interactivo del robot dentro del entorno simulado.

Inicialmente se contempló el uso de paquetes para *SLAM 3D*, con el objetivo de generar un mapa tridimensional del entorno. Sin embargo, debido a las latencias observadas en la transmisión de datos desde el sensor *LIDAR 3D* en la subsección 6.2.2, esta funcionalidad no ha podido integrarse de manera efectiva. La falta de sincronización y el retardo en la entrega de datos impiden que los algoritmos de *SLAM 3D* operen de forma estable en este contexto, lo que comprometería la fidelidad de los resultados.

Por último, el sistema se apoya en una capa externa de automatización, compuesta por *scripts* en *Python* que emplean la *API* remota de *CoppeliaSim* basada en *ZeroMQ* (una librería de mensajería asincrónica de alto rendimiento). Estos *scripts* permiten cargar escenas automáticamente, desplegar múltiples robots en posiciones iniciales definidas y lanzar la simulación sin intervención manual. Este enfoque facilita la realización de experimentos repetibles y el escalado a simulaciones con múltiples agentes robóticos.

4.6 Arquitectura de la simulación robótica para *O3DE*

La arquitectura de simulación robótica en *O3DE*, figura 4.7 , cuando se integra con el *plugin ROS2 Gem*, está diseñada para ofrecer una interacción fluida y directa con el ecosistema *ROS2*. Este enfoque elimina la necesidad de puentes de comunicación, lo que mejora significativamente el rendimiento y simplifica el desarrollo. La simulación se estructura en varios niveles de componentes que interactúan de forma modular dentro del motor y hacia el entorno *ROS2*.

En el núcleo de esta arquitectura se encuentra el motor de simulación de *O3DE*, el cual utiliza sistemas de física avanzados como *PhysX* para modelar de forma realista la dinámica del entorno, las colisiones y la respuesta física de los robots.

Uno de los elementos clave del sistema es el conjunto de componentes de sensores proporcionado por *ROS2 Gem*. Estos sensores incluyen *LIDAR 2D y 3D*, cámaras *RGB* y de profundidad, *IMU*, sensores de odometría y *GPS*. Cada sensor está implementado como un componente autónomo y configurable basado en una clase abstracta común. Esta base permite estandarizar aspectos fundamentales como la frecuencia de publicación, el nombre del tópico *ROS2* asociado y la inclusión automática en el espacio de nombres del robot correspondiente.

La publicación de transformaciones entre marcos de referencia es gestionada automáticamente por el sistema. El componente de *TF* integrado en el motor se encarga de emitir los datos en el tópico */tf* de forma transparente. Esto libera al desarrollador de la necesidad de escribir *scripts* personalizados para calcular y publicar relaciones entre *frames*, como sucede en otros simuladores. También se ofrece compatibilidad total con los mecanismos de nombres en *ROS2*, permitiendo que cada robot tenga su propio espacio de nombres, facilitando la simulación simultánea de múltiples agentes sin colisiones lógicas.

En cuanto al control de movimiento, *O3DE* proporciona el *Robot Control Component*, un sistema que permite recibir y procesar comandos de movimiento desde *ROS2* sin necesidad de código adicional. Este componente es compatible tanto con el mensaje *Twist*, común en robots con tracción diferencial u omnidiireccional, como con el mensaje *AckermannDrive*, utilizado en vehículos con dirección delantera. Gracias a esta integración, se pueden conectar fácilmente herramientas como *teleop_twist_keyboard* (un nodo de *ROS2* para la teleoperación) o planificadores de movimiento para enviar órdenes al robot y observar su comportamiento en el entorno virtual.

El proceso de *spawn*, es decir, la generación dinámica de robots durante la ejecución de una simulación, también está completamente integrado mediante servicios de *ROS2*. El simulador expone un servicio que permite solicitar la instanciación de nuevos modelos en tiempo de simulación, con nombres únicos y posiciones iniciales definidas. Esto elimina la necesidad de *scripts* externos o configuraciones temporales y proporciona un método estructurado y escalable para generar entornos complejos de pruebas.

Además, *O3DE* publica automáticamente el reloj de simulación en el tópico */clock*, lo que permite sincronizar de forma precisa los nodos que dependan del tiempo simulado.

Finalmente, es importante destacar que, aunque el simulador proporciona todos los datos necesarios mediante los sensores simulados y componentes de movimiento, la ejecución de algoritmos de *SLAM 3D* no se realiza dentro del simulador. En su lugar, estos algoritmos —como *RTAB-MAP*— se ejecutan como nodos *ROS2* externos, consumiendo los datos publicados por el simulador para realizar tareas de mapeo, odometría y localización.

En resumen, la arquitectura de *O3DE* con *ROS2 Gem* representa un enfoque moderno, nativo y altamente modular para la simulación robótica. Al eliminar capas intermedias y automatizar la configuración de sensores, transformaciones y control, se posiciona como una plataforma robusta para entornos de prueba de sistemas robóticos complejos, especialmente en escenarios multirrobot o de integración avanzada con *ROS2*.

A continuación, en la tabla 4.1, se puede observar una tabla comparativa con las arquitecturas de los dos sistemas.

Cuadro 4.1: Tabla comparativa de las arquitecturas de simulación robótica en los simuladores *O3DE* y *CoppeliaSim*.

Característica	O3DE con ROS2 Gem	<i>CoppeliaSim con plugin ROS2</i>
Integración <i>ROS2</i>	Nativa, sin puentes	Vía <i>plugin</i> y <i>scripts Lua</i>
Formato de modelos	<i>URDF</i> , <i>SDFFormat</i> , <i>XACRO</i>	<i>URDF</i> , formatos <i>CoppeliaSim</i> (<i>TTM</i> y <i>TTT</i> .)
Sensores	Componentes <i>ROS2</i> configurables (<i>LIDAR</i> , <i>IMU</i> , cámaras <i>RGBD</i>)	<i>Scripts</i> por sensor que publican manualmente
Control	Componentes integrados para <i>Twist</i> y <i>AckermannDrive</i>	<i>Script</i> principal con cinemática inversa
<i>Spawn</i> de robots	Servicio <i>ROS2</i> para <i>spawn</i> dinámico	<i>Script Python</i> que usa <i>ZMQ Remote API</i>
Transformaciones <i>TF</i>	Automáticamente publicadas por componentes <i>ROS2</i>	Manualmente generadas en <i>Lua</i>
Motor físico	<i>PhysX</i> , <i>NvCloth</i> , etc. (según configuración)	<i>Bullet v2.78</i>

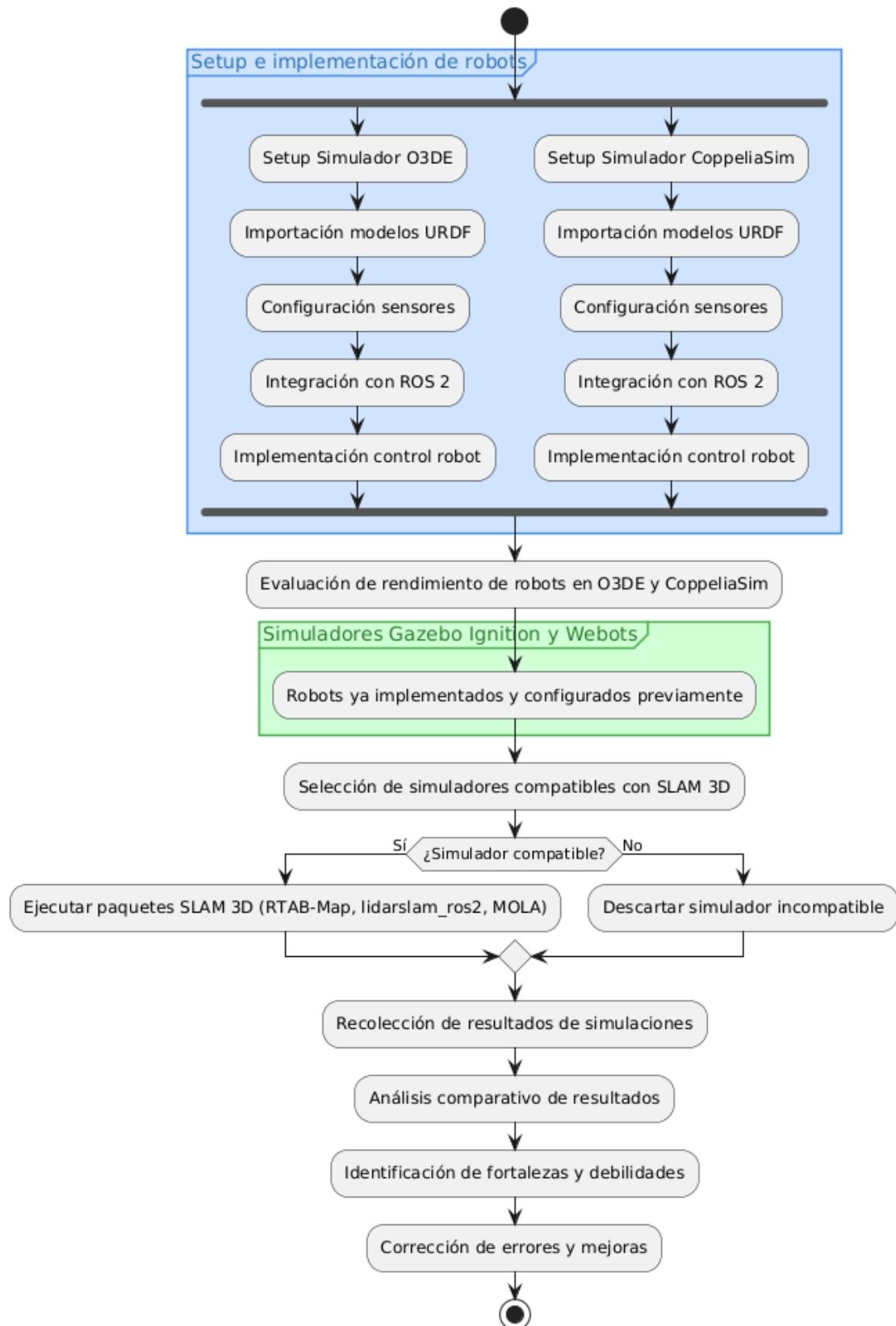


Figura 4.5: Diagrama de actividades del proyecto.

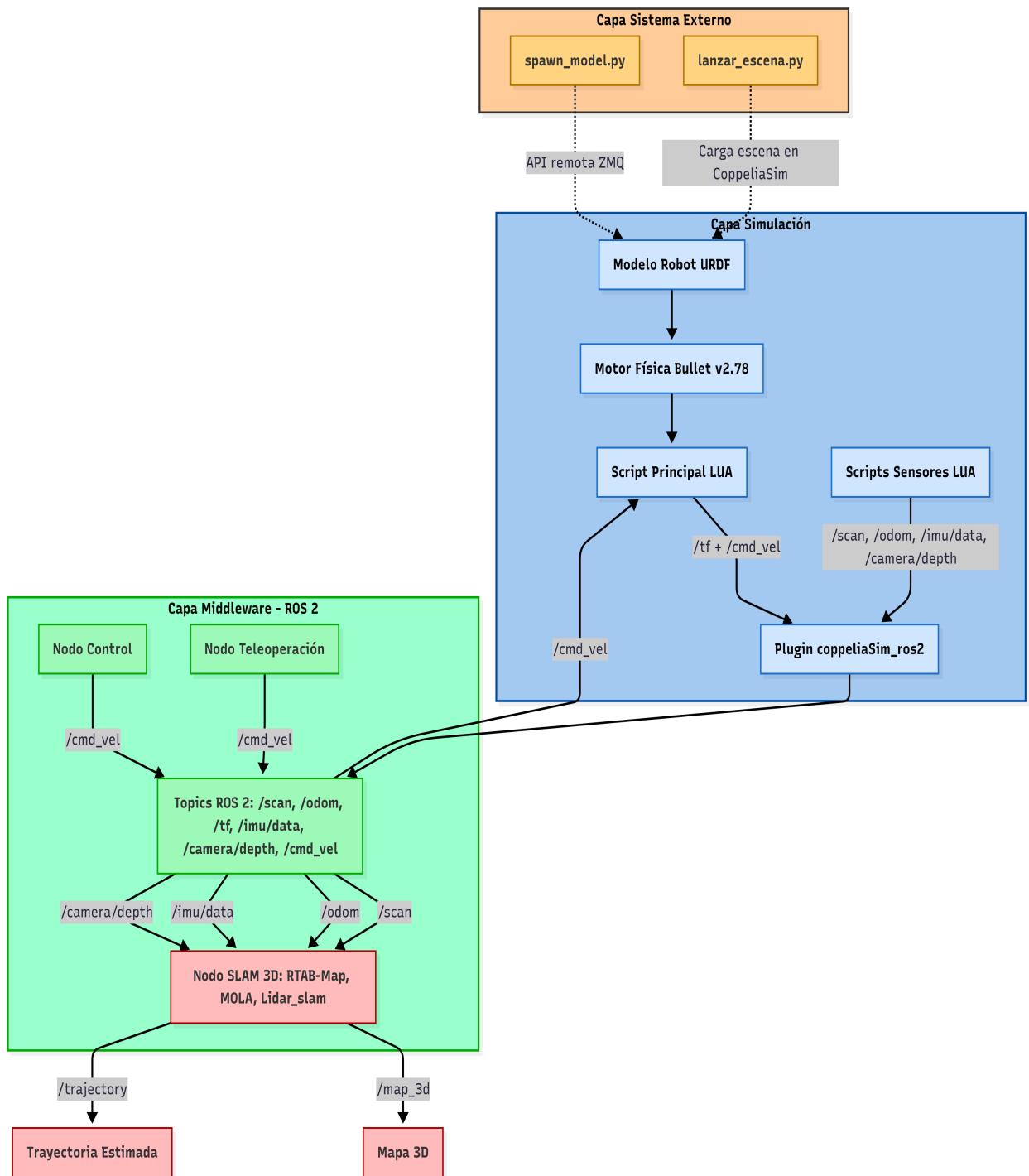
Figura 4.6: Arquitectura para la simulación robótica en *CoppeliaSim*.

Figura 4.7: Arquitectura para la simulación robótica en *O3DE*.

5. Implementación

Este capítulo recoge el proceso de implementación de dos robots móviles en distintos entornos de simulación: *CoppeliaSim* y *O3DE*, y en la posterior integración de diferentes soluciones de *SLAM 3D* para el mapeado del entorno. Aunque no se trata de una guía, sino de un seguimiento de un proceso realizado, se documentan todos los pasos técnicos relevantes, de manera que esta sección también pueda servir como referencia para otros desarrollos similares.

La implementación se divide en tres secciones principales: la implementación de robots móviles en *CoppeliaSim*, la implementación de robots móviles en *O3DE*, y la configuración de los principales paquetes de *SLAM 3D* empleados en los experimentos.

En primer lugar, en la sección dedicada a *CoppeliaSim*, se presenta el *setup* inicial necesario para que el simulador pueda utilizar correctamente tipos de mensajes que no están incluidos de forma nativa en su *plugin* de *ROS2*. Se explica el proceso de conversión del archivo de partida en formato *Xacro* a *URDF* y su posterior importación a *CoppeliaSim*. Además, se detallan los sensores utilizados en los robots, describiendo tanto su selección como su colocación precisa dentro de los modelos (que es prácticamente la misma que en *O3DE*, exceptuando las cámaras de profundidad), así como su integración con *ROS2* para una comunicación efectiva.

También se explica un sistema dinámico de asignación de *namespaces* únicos para permitir el uso simultáneo de múltiples robots dentro de una misma escena. Se aborda la implementación completa del sistema de transformaciones *tf* para definir la relación entre los distintos componentes del robot. Además, dado que no existe un *plugin* específico para el control de movimiento en *CoppeliaSim* con *ROS2*, se explica una solución alternativa para el control de los robots. Finalmente, se explica cómo se implementaron *scripts* externos para el *spawn* dinámico de robots y para la carga automatizada de escenas, facilitando la configuración y ejecución del entorno de simulación.

La siguiente sección se enfoca en la implementación en *O3DE*, donde se detalla el proceso de integración de los robots dentro de este. Se aborda el *setup* general del simulador, la importación de los modelos *URDF*, y la configuración de sensores adaptados al entorno de *O3DE*. Asimismo, se describe la implementación del control de movimiento y de la odometría del robot, así como el mecanismo para el *spawn* de robots.

Finalmente, en la última sección del capítulo se analiza la configuración y uso de paquetes

de *SLAM 3D* para realizar mapeado tridimensional en los entornos simulados. Se presentan tres soluciones: *RTAB-MAP*, *Lidarslam_ros2* y *MOLA*. Para cada una de ellas se explican los pasos de instalación, las modificaciones requeridas en los archivos de lanzamiento y configuración, y los ajustes realizados para asegurar la compatibilidad con los sensores y simuladores utilizados. Además, se detallan los procedimientos para la exportación y visualización de los mapas generados, así como los problemas técnicos encontrados durante la integración.

5.1 Implementación de robots móviles en *CoppeliaSim*

5.1.1 Setup del simulador

Esta fase contempló la preparación del entorno de simulación, con el objetivo de habilitar todas las funcionalidades necesarias para el despliegue de los robots desarrollados.

En primer lugar, se instaló la versión *Edu* de *CoppeliaSim*, la cual proporcionó acceso completo a las funcionalidades del simulador sin fines comerciales. Esta versión se encontraba disponible en la página oficial de *CoppeliaSim* [14] y se descargó de acuerdo con el sistema operativo *Linux*.

Tras la instalación, fue necesario integrar manualmente ciertas interfaces que no se incluían por defecto, pero que resultaban imprescindibles para la comunicación con sensores y la gestión de transformaciones espaciales. En particular, se añadieron las definiciones correspondientes a los mensajes del sensor de inercia (*sensor_msgs/msg/Imu*) y al sistema de transformaciones espaciales (*tf2_msgs/msg/TFMessage*) [15]. Estas configuraciones se incorporaron en el archivo destinado a la especificación de interfaces, localizado dentro del directorio de desarrollo del simulador.

Una vez añadidas las nuevas interfaces, se procedió a recompilar el paquete interno de *ROS2* empleado por el simulador, con el fin de asegurar la correcta detección y utilización de los nuevos tipos de mensajes incorporados.

5.1.2 Obtención del archivo *URDF* del robot a partir del archivo *Xacro*

Una vez completada la compilación, *CoppeliaSim* quedó habilitado para operar con las nuevas interfaces de *ROS2*, permitiendo así la integración de sensores personalizados y la publicación de transformaciones espaciales mediante *tf*.

Para describir la estructura física de los robots empleados, se utilizó el archivo *URDF*, un formato basado en *XML* ampliamente adoptado en *ROS* para representar enlaces, articulaciones, propiedades físicas, geometrías de colisión y sensores. En particular, los modelos base de los robots utilizados fueron proporcionados por la empresa *Robotnik Automation* a través de unos archivos *Xacro* (*XML Macros*), una extensión del formato *URDF* que permite definir modelos de forma modular y reutilizable mediante macros y parámetros, lo que facilitó la incorporación de una estructura ya validada y compatible con el entorno de desarrollo. Posteriormente, dichos archivos fueron convertidos a formato *.urdf* [16], permitiendo así sus importaciones al formato *.tt* que usa *CoppeliaSim* para sus modelos.

Cabe señalar que, tras la importación, el robot incorporado en la escena no contaba aún con sensores ni controladores funcionales, ya que el importador no logró interpretar los complementos (*plugins*) originalmente definidos para *Gazebo Classic*, desde el cual se estaba llevando a cabo la migración. El modelo consistía únicamente en una representación estructural formada por mallas tridimensionales que definían los distintos componentes físicos del robot. Estos elementos estaban organizados como eslabones conectados mediante articulaciones, fijas o móviles, que permitían reflejar de forma aproximada su geometría y configuración cinemática dentro del entorno de simulación proporcionado por *CoppeliaSim*.

5.1.3 Colocación de los sensores en los elementos del robot

Durante el proceso de implementación de los modelos robóticos en el entorno de simulación *CoppeliaSim*, una etapa esencial fue la incorporación adecuada de sensores virtuales, cuyo propósito consistió en emular el comportamiento sensorial de los dispositivos físicos reales. Esta incorporación se llevó a cabo utilizando las herramientas integradas en el simulador, que ofrecía tanto sensores nativos como componentes preconfigurados disponibles en su catálogo de modelos.

Los sensores nativos, como cámaras *RGB*, fueron integrados directamente en los *frames* estructurales de los robots simulados. Por otro lado, los sensores preconfigurados —accesibles desde la sección *components/sensors* del explorador de modelos— aportaron ventajas adicionales, incluyendo configuraciones visuales, *scripts* asociados y conexiones predefinidas, lo que facilitó una integración más directa y funcional dentro del entorno de simulación.

Durante este proceso se prestó especial atención a la orientación y posicionamiento relativo de cada sensor respecto al *frame* padre al que quedó vinculado. Cualquier desviación angular o posicional hubiera requerido la definición de transformaciones adicionales en *ROS2* mediante publicaciones en el sistema de coordenadas espaciales *tf*. Por esta razón, se procuró mantener las coordenadas y orientaciones relativas igualadas a cero, asegurando así una correspondencia coherente con el modelo físico.

Respecto a los sensores incorporados, se configuraron dos conjuntos diferenciados para los modelos *Rb_Watcher* y *Rb_Robout*. Ambos robots fueron equipados con una combinación de sensores preconfigurados y nativos, seleccionados en función de las capacidades sensoriales necesarias para su operación simulada.

El modelo ***Rb_Watcher*** incorporó los siguientes sensores preconfigurados y nativos:

- Un **Gyrosensor** y un **Accelerometer**, situados como hijos del componente *robot_imu_joint* para simular datos iniciales.
- Un sensor **GPS**, anclado al *robot_gps_joint* para emular la localización global.
- Un sensor **Velodyne-VPL16**, correspondiente a un *LIDAR 3D* de alta resolución, vinculado al *robot_top_3d_laser_base_joint*.
- Dos **cámaras RGB** como sensores nativos: una en la parte frontal del robot, asociada al *robot_front_rgbd_camera_joint*, y otra en la parte superior, montada sobre el *robot_top_ptz_camera_frame_joint*.

Por su parte, el modelo ***Rb_Robout*** fue dotado de:

- Un **Gyrosensor** y un **Accelerometer** como hijos del *rbrobot_imu_joint*, para proveer capacidad de medición inercial.
- Dos sensores **fastHokuyo_ROS**, correspondientes a *LIDAR 2D* de corto alcance y alta frecuencia de muestreo; uno colocado en la parte frontal, vinculado al *rbrobot_front_laser_base_joint*, y el otro en la parte trasera, conectado al *rbrobot_rear_laser_base_joint*.
- Dos **cámaras RGB** como sensores nativos: una en la parte frontal del robot, asociada al *robout_front_rgbd_camera_joint*, y otra en la parte trasera, montada sobre el *robout_rear_rgbd_camera_joint*.

Estas configuraciones sensoriales permitieron replicar de forma realista la percepción del entorno por parte de cada robot.

5.1.4 Sistema dinámico de asignación de *namespaces* para múltiples robots

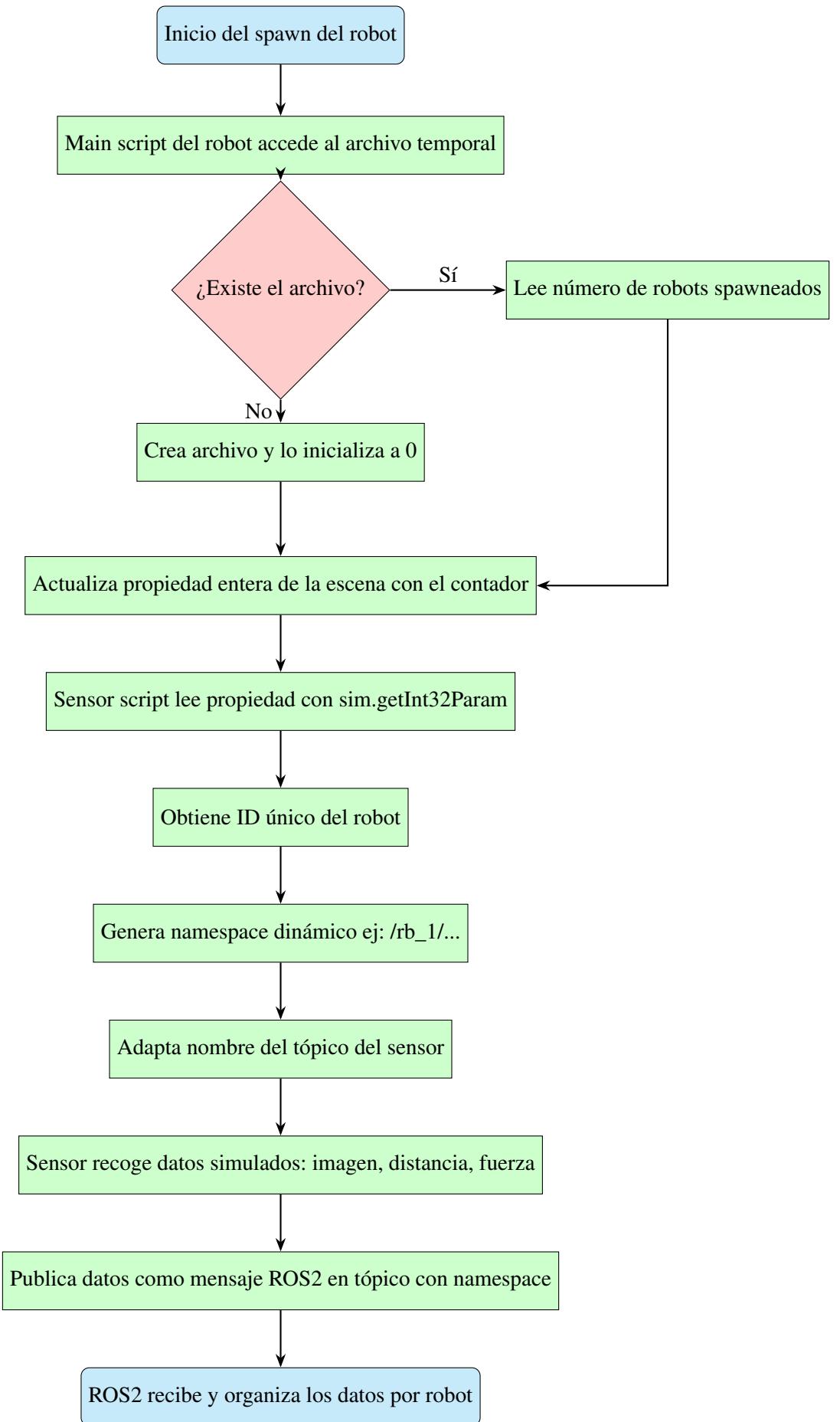
Con el objetivo de permitir la coexistencia de múltiples robots en la misma escena sin conflictos entre sus comunicaciones, cada uno de ellos debe operar dentro de un espacio de nombres (*namespace*) independiente. En un primer planteamiento, se consideró que cada sensor leyera directamente un contador almacenado en un archivo temporal cada vez que se *spawneaba* un robot, para determinar su identificador único. Sin embargo, esta solución presentaba problemas potenciales, como la latencia causada por múltiples accesos concurrentes al archivo, además de posibles colisiones de lectura y escritura simultánea. Por ello, **se implementó un mecanismo no visto hasta ahora en ninguna implementación de robots en *CoppeliaSim*.** Un sistema alternativo, más eficiente y alineado con la estructura jerárquica de *CoppeliaSim*, que evita la lectura repetitiva del archivo por parte de cada sensor.

Cuando un nuevo robot es insertado en la escena, su *script* principal (ubicado en la raíz del modelo) accede a un fichero temporal donde se guarda el número de robots que han sido *spawneados* hasta ese momento; si no existe, lo crea e inicializa a cero los contadores. Este *script* realiza una única lectura del archivo y utiliza esa información para actualizar una propiedad de la escena, concretamente una propiedad entera asociada al objeto de mayor jerarquía del simulador. Las propiedades de escena en *CoppeliaSim* funcionan como variables globales que pueden ser leídas por cualquier objeto o *script* situado jerárquicamente por debajo de la escena principal. Una vez actualizada la propiedad, cada *script* individual de los sensores puede acceder a ella a través de la función:

```
sim.getInt32Param(sim.handle_scene, "signal.roboutCounter", {noError = true}) [17].
```

Esta función permite leer directamente el valor de la propiedad que contiene el identificador del robot, sin necesidad de acceder al archivo. Gracias a este mecanismo, cada sensor adapta dinámicamente el nombre del tópico en el que publica sus datos, incluyendo un *namespace* único derivado del identificador del robot. Así, se garantiza que todos los tópicos publicados por sensores de distintos robots permanezcan aislados y organizados, lo que facilita tanto la gestión como el análisis posterior de la información.

Este enfoque, no solo mejora la eficiencia del sistema al evitar operaciones redundantes de entrada/salida, sino que además se integra de forma natural con la arquitectura jerárquica de *CoppeliaSim*, haciendo posible una comunicación robusta y escalable entre los sensores simulados y el entorno de *ROS2*.

Diagrama 3: Flujo del proceso de spawn y asignación de namespaces en *CoppeliaSim* para ROS2.

5.1.5 Implementación de los sensores

En esta sección se describe cómo se han implementado los distintos sensores virtuales utilizados por los robots simulados en *CoppeliaSim*. Todos ellos siguen un mismo patrón de integración que consiste en:

1. Seleccionar un *plugin* o modelo que simule el sensor dentro del simulador.
2. Situarlo en el lugar correspondiente del robot.
3. Obtener los datos del sensor mediante un *script* en *Lua*.
4. Crear un publicador en *ROS2* del tipo de mensaje correspondiente.
5. Adaptar los datos recogidos al formato de *ROS2*.
6. Publicar el mensaje en un tópico con un espacio de nombres (*namespace*) único para cada robot.

En las secciones anteriores se explicaron los puntos 1, 2 y 6. A continuación, explicaremos en detalle los puntos 3, 4 y 5.

Cámara *RGB*

Para transmitir imágenes al sistema *ROS2*, se ha implementado un publicador que emite mensajes del tipo *sensor_msgs/msg/Image* en un tópico con un identificador único para cada cámara.

La adquisición de imágenes del sensor de visión se realizó mediante la función *sim.getVisionSensorImg()*. Esta función recibe como parámetro la referencia a la cámara *RGB* y retorna una imagen, junto con sus dimensiones en píxeles. A partir de estos datos, se construye en tiempo de ejecución un mensaje en forma de tabla de *Lua* [18]. Esta tabla se completa en tiempo de ejecución con las claves y valores de la tabla 5.1

Campo	Valor	Descripción
header.stamp	<code>simROS2.getTime()</code>	Marca de tiempo del mensaje <i>ROS2</i>
header.frame_id	"rb_rbout_front_rgb_camera_frame"	Identificador del <i>frame</i> de referencia para el sensor
height	<code>height</code> (variable previamente definida)	Altura de la imagen en píxeles
width	<code>width</code> (variable previamente definida)	Anchura de la imagen en píxeles
encoding	"rgb8" (RGB de 8 bits por canal)	Formato de codificación de la imagen
is_bigendian	1	Define el orden de los bytes (1 = <i>big endian</i> , 0 = <i>little endian</i>)
step	<code>width * 3</code>	Número de bytes por fila de imagen (ancho × bytes por píxel)
data	<code>imgUint8</code> (imagen procesada previamente)	Datos de la imagen en formato <i>array</i> de bytes (<code>uint8</code>)

Cuadro 5.1: Campos de la tabla *Lua* para el mensaje de imagen *ROS2*.

Una vez rellenada la tabla que constituye el mensaje, se envía la imagen periódicamente por el tópico de la cámara.

LIDAR 2D

Para la implementación de un sensor *LIDAR 2D* en el entorno de *CoppeliaSim* [20], se evaluaron distintas opciones disponibles en la biblioteca del simulador.

En primer lugar, se probó un sensor de proximidad con forma de haz cilíndrico. Este resultó inadecuado, ya que únicamente permite detectar la posición del objeto más cercano, es decir, emite un único rayo y no proporciona un barrido angular completo como el requerido por un sensor *LIDAR 2D*. Posteriormente, se exploró el uso del modelo *2D Laser Scanner* disponible en la carpeta *Model_browser/components/sensors*. No obstante, este sensor no incluía un *script* preparado para la comunicación con *ROS*, motivo por el cual fue descartado.

Finalmente, se optó por utilizar el modelo *fastHokuyo_ROS*, ya que contaba con un *script* funcional para la publicación de datos en *ROS1*. Para su integración en el sistema, fue necesario modificar el *script* del sensor con el fin de utilizar el *plugin* correspondiente a *ROS2*, reemplazando las funciones del antiguo *middleware* por sus equivalentes actuales, y ajustando tanto el nombre del tópico como el *frame_id* del mensaje.

LIDAR 3D

Para la simulación de un sensor *LIDAR 3D* en el entorno de *CoppeliaSim*, se empleó el modelo *Velodyne VPL-16* [19], el cual ofrece una representación realista del sensor físico homónimo, ampliamente utilizado en tareas de mapeado, navegación autónoma y percepción del entorno. Este sensor emula internamente el funcionamiento de múltiples haces láser mediante cuatro cámaras virtuales distribuidas angularmente, replicando el proceso de adquisición de datos del sensor real. A partir de las imágenes generadas por estas cámaras, y conociendo la geometría del entorno, se calculan distancias a los objetos circundantes, generando una nube de puntos tridimensional. El resultado consiste en un *array* (estructura de datos) unidimensional en el que cada punto está definido por tres valores: sus coordenadas *X*, *Y* y *Z* en el sistema de referencia del sensor.

Con el objetivo de integrar esta nube de puntos en *ROS2* se convirtió dicha información al formato estándar *sensor_msgs/msg/PointCloud2*. Este tipo de mensaje encapsula no solo los puntos, sino también metadatos relevantes como el marco de referencia, marca temporal y la estructura de los datos. Para ello, se creó un *publisher* de tipo *sensor_msgs/msg/PointCloud2* utilizando el *plugin* de *ROS2* en *CoppeliaSim*. Dentro del *script* asociado al sensor, se codificaron los datos agrupando las coordenadas *X*, *Y* y *Z* de cada punto, y se convirtieron al formato binario adecuado (*float32* empaquetado como *uint8[]*). Una vez estructurado el mensaje conforme al estándar *PointCloud2*, tal como muestra la tabla 5.2, este se publica periódicamente en un tópico definido, quedando así disponible para su uso en nodos *ROS2* en tiempo real.

Campo	Valor	Descripción
header.stamp	simROS2.getTime()	Marca de tiempo del mensaje, sincronizada con el tiempo de simulación.
header.frame_id	"rb_velodyne_frame"	Identificador del marco de referencia del sensor en <i>ROS2</i> .
height	1	Altura de la nube de puntos (1 para nube unidimensional). Ya que el sensor devuelve un array 1D.
width	tamaño_array / 3	Número de puntos en la nube (cada punto contiene 3 valores: x, y, z).
fields	x, y, z	Lista de campos que definen el contenido de cada punto (formato float32).
is_bigendian	false	Indica si los datos están en formato <i>big-endian</i> (habitualmente false).
point_step	12	Tamaño en bytes de cada punto (3 floats de 4 bytes cada uno).
row_step	width * point_step	Tamaño total en bytes de una fila. Como height = 1, equivale al tamaño total.
data	array de uint8	Datos binarios empaquetados (coordenadas codificadas como float32).
is_dense	false	Especifica si todos los puntos son válidos (false si puede haber NaN).

Cuadro 5.2: Campos del mensaje *sensor_msgs/msg/PointCloud2* para el *LIDAR 3D*.

IMU

La Unidad de Medida Inercial (*IMU*, por sus siglas en inglés) es un sensor que combina acelerómetros y giróscopios con el fin de medir el movimiento de un cuerpo en el espacio. Este tipo de sensor proporciona información esencial sobre la aceleración lineal, la velocidad angular y, en algunos casos, la orientación del objeto. La aceleración lineal se refiere a los cambios en la velocidad a lo largo de los tres ejes cartesianos (x, y, z), mientras que la velocidad angular describe la rotación del cuerpo respecto a dichos ejes.

En el entorno de simulación *CoppeliaSim*, los *scripts* encargados de calcular tanto la aceleración lineal como la velocidad angular ya se encontraban implementados por separado. La tarea realizada consistió en integrar ambos *scripts* en uno solo, permitiendo así recoger simultáneamente los datos de acelerómetro y giroscopio. La aceleración lineal se obtiene a partir de la fuerza registrada por un sensor de fuerza/torque, dividida entre la masa del cuerpo simulado. Por otro lado, la velocidad angular se calcula a partir del cambio de orientación entre dos pasos consecutivos de la simulación, utilizando para ello matrices de transformación o ángulos de *Euler*, en función del tiempo transcurrido.

Una vez obtenidos ambos conjuntos de datos, se construye en tiempo de ejecución un mensaje del tipo *sensor_msgs/msg/Imu* con los campos de la tabla 5.3. Este mensaje encapsula tanto la aceleración como la velocidad angular y finalmente se publica mediante un *publisher* de *ROS2*, como en los sensores anteriores.

Campo	Valor	Descripción
header.stamp	simROS2.getTime()	Tiempo de la medición
header.frame_id	rb_watcher_imu_joint	Identificador del marco de referencia
orientation	{ x=0, y=0, z=0, w=1 }	Orientación del sensor (cuaternion)
orientation_covariance	{ -1, 0, ..., 0 }	Covarianza de orientación (no se usa)
angular_velocity	Calculada por diferencia de ángulos entre pasos de simulación	Velocidad angular (rad/s)
angular_velocity_cov	Matriz diagonal con valores 0.01	Covarianza de velocidad angular
linear_acceleration	Fuerza medida en el sensor dividida entre masa	Aceleración lineal (m/s ²)
linear_acceleration_cov	Matriz diagonal con valores 0.1	Covarianza de aceleración lineal

Cuadro 5.3: Campos del mensaje *sensor_msgs/msg/Imu* para la simulación del sensor *IMU* en *ROS2*.

GPS

El sistema de posicionamiento global (*GPS*) es un sensor utilizado para obtener la posición absoluta de un objeto en el espacio, expresada comúnmente en coordenadas de latitud, longitud y altitud. En entornos de simulación como *CoppeliaSim*, estas coordenadas se representan dentro del marco del mundo simulado, generalmente mediante posiciones cartesianas en un sistema de referencia global.

En este caso, la implementación del *GPS* se ha basado en un *script* que accede a la posición absoluta de un objeto dentro del entorno de simulación. Esta posición es obtenida directamente a través de la función *sim.getObjectPosition*, la cual devuelve las coordenadas en los ejes X, Y y Z. A esta información se le puede añadir ruido artificial y desplazamiento (*offset*) configurable, con el fin de simular imprecisiones típicas de sensores reales.

El *script*, que ya contenía la lógica para obtener la posición, fue adaptado para estructurar estos datos en un mensaje del tipo *geometry_msgs/msg/Point*, tal como se muestra en la tabla 5.4. Posteriormente, el mensaje es publicado mediante un *publisher* de *ROS2* de forma periódica al igual que los demás sensores.

Campo	Valor	Descripción
x	objectAbsolutePosition[1]	Coordenada X (posición en el eje X) con ruido opcional
y	objectAbsolutePosition[2]	Coordenada Y (posición en el eje Y) con ruido opcional
z	objectAbsolutePosition[3]	Coordenada Z (posición en el eje Z) con ruido opcional

Cuadro 5.4: Campos del mensaje *geometry_msgs/msg/Point* correspondiente a la simulación del *GPS*.

5.1.6 Publicación de *tf* en ROS2

La publicación de las transformaciones (*TF*) del robot se realiza desde el *script* principal de cada robot para que *ROS2* pueda interpretar correctamente la estructura espacial del robot simulado. Esto es esencial para que el sistema conozca en todo momento la posición y orientación relativas entre las distintas partes del robot, lo que resulta fundamental para tareas como navegación, percepción y control de movimiento.

Para ello, se implementa una función llamada *publishTransforms* que se ejecuta periódicamente durante la simulación. Esta función construye un mensaje del tipo *tf2_msgs/msg/TFMessage*, que contiene una lista de transformaciones entre diferentes *frames* o marcos de referencia. La marca temporal actual de la simulación se obtiene mediante *simROS2.getTime()*, lo que garantiza que todas las transformaciones comparten una referencia temporal común.

Dentro de esta función, se utiliza una función auxiliar denominada *addTransform*. Esta función recibe el identificador y nombre del *frame* padre, así como el nombre y el identificador del *frame* hijo. Con esta información calcula la posición relativa del *frame* hijo respecto al padre y su orientación relativa en formato cuaternión, utilizando las funciones internas de *CoppeliaSim*. Estos datos se organizan en un bloque que luego se añade a la lista de transformaciones del mensaje.

Cada transformación contiene los campos mostrados en la tabla 5.5

Campo	Descripción
<code>header.stamp</code>	Marca temporal que sincroniza la transformación
<code>header.frame_id</code>	Nombre del frame padre
<code>child_frame_id</code>	Nombre del frame hijo
<code>transform.translation</code>	Vector con la posición relativa
<code>transform.rotation</code>	Vector con la orientación relativa en cuaternión

Cuadro 5.5: Campos de cada transformación en el mensaje *tf2_msgs/msg/TFMessage*.

La función *publishTransforms* añade de manera jerárquica todas las transformaciones necesarias para describir la estructura espacial del robot, respetando la relación padre-hijo entre sus componentes. Esto permite que *ROS2* conozca la posición y orientación relativa de cada parte del robot en tiempo real.

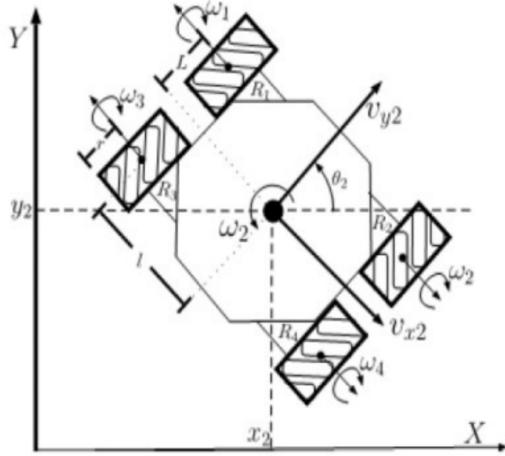
Una vez todas las transformaciones están agregadas, el mensaje completo se publica en el tópico */tf* mediante un *publisher* de *ROS2*, facilitando a cualquier nodo del sistema el acceso a esta información para realizar transformaciones de coordenadas.

5.1.7 Control de movimiento

Para controlar el movimiento de los robots móviles en *CoppeliaSim*, se exploró inicialmente la posibilidad de utilizar algún *plugin* de controlador genérico para *ROS2* que facilitara la gestión de las velocidades de las ruedas. Sin embargo, durante la búsqueda solo se encontró un *plugin* disponible para *ROS1*, lo que impidió su utilización directa en el entorno *ROS2*. Debido a esta limitación, se optó por implementar el control del robot de forma manual.

La estrategia adoptada consistió en usar la cinemática inversa para la configuración *mecanum* de cuatro ruedas, ilustrada en la figura 5.1, y que comparten ambos robots. Esta configuración permite calcular las velocidades $\omega_1, \omega_2, \omega_3$ y ω_4 que deben tener cada una de las ruedas a partir de la fórmula 5.1 para conseguir las velocidades lineales (V_x y V_y) y angulares (ω) deseadas del robot en conjunto.

Para lograrlo, se creó un *subscriber* que se suscribe al tópico */cmd_vel*, donde se reciben los comandos de velocidad lineal y angular para el robot. A través de un *callback* (una función que se ejecuta cada vez que se recibe un mensaje en el tópico) se extraen estas velocidades y se calculan

Figura 5.1: Configuración *Mecanum* de 4 ruedas [21].

las velocidades específicas que debe tener cada rueda utilizando las fórmulas de la cinemática inversa propias de los robots con ruedas *mecanum*.

$$\begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{bmatrix} = \frac{1}{r} \begin{bmatrix} 1 & 1 & -(L+l) \\ -1 & 1 & (L+l) \\ -1 & -1 & -(L+l) \\ 1 & -1 & (L+l) \end{bmatrix} \begin{bmatrix} V_x \\ V_y \\ \omega \end{bmatrix} \quad (5.1)$$

Antes de aplicar estas velocidades, es necesario obtener las referencias o *handles* de las articulaciones de las ruedas, que son los identificadores internos que permiten enviar comandos directos a cada rueda.

Finalmente, las velocidades calculadas para cada rueda se asignan a sus respectivas articulaciones mediante la función *sim.setJointTargetVelocity*, que ajusta la velocidad objetivo de las ruedas para que el robot se mueva según las instrucciones recibidas. De esta manera, el control del movimiento se realiza en tiempo real y de forma efectiva, a pesar de no contar con un *plugin* de controlador específico para *ROS2*.

Esta solución garantiza que el robot responda correctamente a los comandos */cmd_vel*, transformando las velocidades globales del robot en velocidades individuales para cada rueda y controlando directamente las articulaciones para lograr el movimiento deseado.

Para mover el robot en la simulación podemos usar el nodo *teleop_twist_keyboard* de *ROS2* con:

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard --ros-args -r /cmd_vel:=/watcher_1/rb_watcher_cmd_vel
```

5.1.8 Script de spawn dinámico de robots

Al inicio del proyecto se estableció como requisito la posibilidad de *spawnear* robots dinámicamente mientras la simulación está en ejecución. Para ello, en *CoppeliaSim* se implementó un *script* que utiliza el *plugin Remote API*, el cual viene activado por defecto en la versión educativa del simulador.

Para interactuar con este *plugin* desde *Python*, se instaló la *API coppeliasim-zmqremoteapi-client*. El proceso para *spawnear* robots una vez se creó el *script* consistió en ejecutarlo desde la

línea de comandos usando *Python*, especificando el archivo del modelo *.ttm*, la posición (*x*, *y*, *z*) y la orientación (*roll*, *pitch*, *yaw*) en grados. Por ejemplo:

```
$ python3 spawn_model.py rb_watcher.ttm 1.0 0.5 0.2 0 0 0
```

Esto permitió crear instancias de robots en ubicaciones y orientaciones específicas sin detener la simulación, facilitando pruebas y escenarios dinámicos.

5.1.9 Script de carga de escena

Para facilitar el uso automatizado del simulador y agilizar las pruebas en distintos entornos, se creó un *script* en *Python* que permite lanzar *CoppeliaSim* y cargar una escena específica de forma directa desde la línea de comandos. Con este enfoque se evitó la necesidad de abrir manualmente el simulador y seleccionar la escena deseada, lo que resultó especialmente útil en la experimentación repetitiva.

El *script* hace uso de la biblioteca estándar *subprocess* de *Python*, la cual permite ejecutar comandos del sistema operativo. En concreto, el *script* invoca el ejecutable de *CoppeliaSim* (archivo *.sh* en sistemas Linux) y le pasa como argumento el archivo de la escena (*.ttt*) que se desea cargar. Una vez creado, se pudo ejecutar sin fallos mediante *Python* pasándole como argumento el nombre de la escena de la siguiente forma:

```
$ python3 lanzar_escena.py escena_simple.ttt
```

5.2 Implementación de robots móviles en *O3DE*

5.2.1 Setup del simulador

Para comenzar a desarrollar proyectos de simulación de robots móviles en *O3DE* integrados con *ROS2*, fue necesario realizar una preparación completa del entorno, siguiendo la guía del simulador [22], tanto a nivel de sistema como del simulador.

En primer lugar, fue indispensable verificar que el sistema contara con las herramientas de construcción actualizadas, entre ellas *CMake* (versión igual o superior a 3.22.0), *Clang* (versión 12 o superior, usando la versión 14 disponible en *Ubuntu 22.04*) y *Ninja*, que es el sistema de construcción utilizado por *O3DE*. Junto con esto, se instalaron múltiples dependencias relacionadas con bibliotecas gráficas, entrada de dispositivos y otros elementos necesarios para compilar correctamente el motor.

Una vez preparado el entorno, se procedió a instalar *O3DE*. En lugar de utilizar una versión precompilada del motor, se optó por descargar el código fuente desde *GitHub* y compilarlo manualmente. Esta decisión respondió a la necesidad de personalización, compatibilidad con versiones específicas y un rendimiento significativamente mejor. En pruebas preliminares, se observó que la versión compilada desde código alcanzaba entre 50 y 60 fotogramas por segundo, frente a los 9 a 12 FPS que ofrecía la versión precompilada bajo ciertas condiciones.

El proceso de instalación incluyó la incorporación de *Git LFS (Large File Storage)*, ya que el repositorio de *O3DE* contiene numerosos archivos de gran tamaño que no se gestionan mediante *Git* estándar. Posteriormente, se clonó el repositorio del motor en la rama estable correspondiente, y se descargaron los archivos necesarios.

En paralelo, se instalaron los paquetes específicos requeridos por *ROS2*, así como los mensajes utilizados en simulaciones (como los de navegación, visión o control). Esta parte fue fundamental para garantizar que los elementos de *ROS2* se integraran correctamente con el entorno de simulación.

Una vez descargado y preparado el motor, se incorporaron los llamados *GEMS*, que son *plugins* que extienden las capacidades de *O3DE*. En este caso, se integró específicamente la *ROS2 Gem* [23], encargada de proporcionar compatibilidad con nodos, mensajes y tópicos de *ROS2*.

directamente desde el entorno del motor. Para ello, fue necesario clonar el repositorio oficial de *O3DE Extras*, establecer las rutas correspondientes en el entorno y registrar tanto la *ROS2 Gem* como las demás *GEMS* y plantillas que proporciona esta extensión. Todo este proceso se centralizó mediante la configuración del archivo `o3de_manifest.json`, un archivo clave del ecosistema de *O3DE* que almacena las rutas y relaciones entre motores, proyectos, plantillas y *plugins*. Su correcta gestión es fundamental, ya que una configuración errónea en este archivo puede impedir la compilación o ejecución del entorno.

Con el motor ya configurado e integrado con *ROS2*, se procedió a crear un nuevo proyecto de simulación. Para ello, se definió un nombre de proyecto y una ubicación en el sistema de archivos. La creación del proyecto se basó en una plantilla específica para proyectos *ROS2*, que proporciona una estructura inicial adecuada para el desarrollo de robots. Finalmente, se compiló el proyecto completo quedando así listo para ser utilizado.

Tras completar todos estos pasos, se obtuvo un entorno de simulación completamente funcional, integrado con *ROS2*, y preparado para comenzar el desarrollo de robots móviles.

5.2.2 Importación de archivos *URDF*

Durante esta fase del desarrollo, se procedió a la incorporación del modelo del robot en el entorno de simulación mediante la importación de su archivo *URDF*. En aquellos casos en que se partía de un archivo `.urdf.xacro`, fue necesario realizar previamente su conversión al formato adecuado, siguiendo el mismo procedimiento que para la misma tarea en *CoppeliaSim*.

Una vez importado, el robot apareció correctamente en la escena de simulación. No obstante, se observó que el modelo resultante no incluía sensores y que algunos de sus *frames* presentaban orientaciones incorrectas, lo que obligó a realizar ajustes posteriores.

El sistema detectó automáticamente todos los *frames* definidos en el archivo *URDF*, asignándoles sus respectivos nombres. Además, gracias al uso del *plugin* de *ROS2*, los componentes *ROS2 Frame* se añadieron de forma automática a cada parte del robot, lo que facilitó considerablemente el proceso de configuración.

Para evitar conflictos en los tópicos *TF* generados por cada *frame*, se configuró el parámetro *Namespace* con el valor *Default*, haciendo coincidir su nombre con el del *frame* correspondiente. Esta decisión permitió asegurar que cada transformación se publicara de forma independiente, con nombres claros y sin solapamientos derivados de la jerarquía estructural del modelo.

5.2.3 Configuración de sensores en *O3DE*

En *O3DE*, se implementaron los sensores utilizando el *plugin* *ROS2*, que ofreció todos los componentes necesarios para integrar robots dentro del motor. Gracias a este *plugin*, los sensores pudieron publicar sus datos directamente en *ROS2*, lo que facilitó la comunicación entre la simulación y el sistema robótico.

La incorporación de cada sensor se realizó desde la jerarquía de componentes, donde se seleccionaba el componente deseado y se añadía el sensor correspondiente. Tras esto, se configuraron sus propiedades para asegurar que funcionaran correctamente dentro del entorno de *ROS2*.

La distribución y cantidad de sensores fueron las mismas que en los robots de *CoppeliaSim*, tal como se explicó en la subsección 5.1.3, con la diferencia de que, en este caso, se añadieron cámaras de profundidad (*Depth*) junto a las cámaras *RGB* en los mismos componentes, ya que este simulador sí cuenta con soporte para ese tipo de sensores.

Entre las configuraciones comunes que se aplicaron a los sensores, se activó la opción de *Publishing Enable* para que el sensor pudiera publicar datos en *ROS2*. También se estableció la frecuencia (*Frequency*), que definió la tasa a la que el sensor enviaba los datos en hertzios. Esto fue clave para mantener un equilibrio entre rendimiento y precisión, controlando la frecuencia de actualización sin saturar el sistema.

Por otro lado, se especificó el *Topic Name*, que determinó el nombre del tópico en *ROS2* donde el sensor publicaba la información. Esto permitió que otros nodos se suscribieran y recibieran los datos generados por el sensor.

En cuanto a la *Política de Calidad de Servicio (QoS)*, se configuraron los parámetros que regulan la comunicación entre publicadores y suscriptores, afectando aspectos como la fiabilidad, el rendimiento y la latencia. Se emplearon perfiles como *Reliable*, que garantizaba la entrega de todos los mensajes, ideal para datos críticos, y *Best Effort*, que no aseguraba la entrega completa pero era más eficiente para datos con alta frecuencia donde se toleraban pérdidas.

También se ajustaron las políticas *Keep Last* y *Keep All* para controlar cuántos mensajes se almacenaban en el búfer, facilitando que nuevos suscriptores recibieran mensajes recientes o todos los mensajes disponibles. Además, la opción *Durability* se configuró para decidir si los mensajes debían mantenerse disponibles para suscriptores que se conectaran después de la publicación.

Más allá de estas configuraciones generales, cada sensor fue adaptado según sus características particulares, ajustando parámetros específicos para optimizar su rendimiento dentro del sistema y asegurar un correcto funcionamiento, estas configuraciones específicas se explicarán a continuación para cada sensor.

Cámaras RGBD

Para añadir cámaras *RGB* o *RGBD (Depth + RGB)*, se utilizó el componente del *plugin ROS2* denominado *Camera Sensor*. Este componente, mostrado en la figura 5.2 incluye los parámetros *Depth camera* y *Color camera*, que, al activarse, permiten configurar las propiedades específicas de cada tipo de cámara. Cuando estos parámetros se desactivan, las cámaras correspondientes no se emplean, aunque no existe una señal visual clara, como un oscurecimiento, que indique esta desactivación.

Al activar simultáneamente las cámaras *Depth* y *Color* dentro de un mismo componente, se constató que ambas compartían configuraciones como la resolución y los valores de *near clip distance* y *far clip distance*, que definen el rango de visión efectivo. Esto implica que no es posible asignar resoluciones diferentes a cada cámara en un solo componente.

Por esta razón, cuando se requería que la cámara *Depth* y la cámara *Color* tuvieran resoluciones distintas, se optó por crear dos componentes de cámara separados. De este modo, cada sensor podía configurarse de manera independiente, ajustando resoluciones y otros parámetros específicos para cada tipo de cámara.

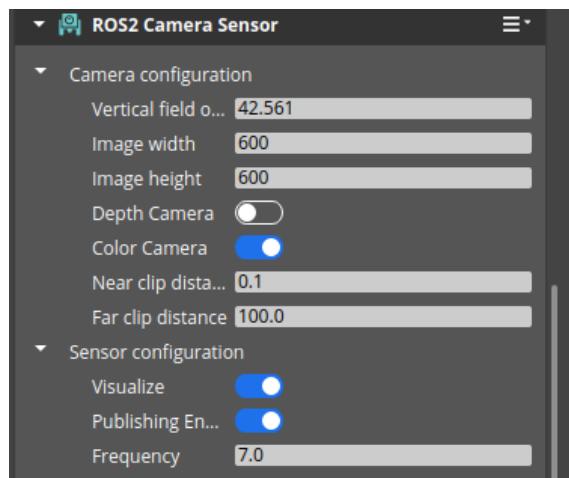


Figura 5.2: Componente de *ROS2 GEM Camera Sensor*.

LIDAR 2D

Para implementar el *LIDAR 2D* se usó el componente *Lidar 2D Sensor* de la figura 5.3 ya que el importador *URDF* no reconoció este tipo de sensor automáticamente y se debió añadir manualmente al objeto correspondiente.

Además de configurar el tópico de publicación, fue necesario definir varios parámetros clave:

- *Points per layer*: número de puntos que el sensor utiliza para detectar objetos en su única capa.
- *Min/Max horizontal range*: ángulo de apertura del sensor; el ángulo 0 se corresponde con la intersección entre los planos X e Y.
- *Min/Max range*: rango de distancia en el que el sensor puede detectar objetos.
- *Lidar Model*: se estableció en *CustomLidar2D* para habilitar la modificación de los parámetros anteriores.

Durante el proceso de configuración, se observó que la colocación del sensor dentro de una malla puede provocar colisiones, incluso si dicha malla no posee un colisionador activo. Para mitigar este comportamiento, se optó por posicionar el sensor ligeramente fuera del modelo tridimensional o, alternativamente, incrementar el valor de min range para evitar interferencias con la superficie. Otra solución explorada fue la asignación de capas de colisión específicas a los objetos que deben ser ignorados por el sensor. No obstante, la documentación disponible no proporciona información detallada sobre la implementación de esta última estrategia.

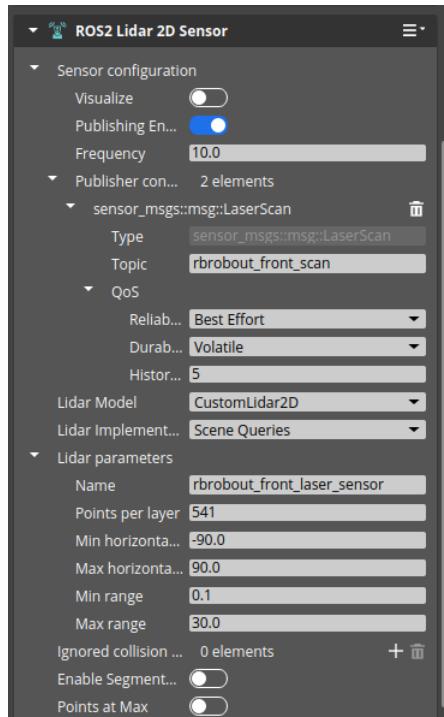


Figura 5.3: Componente de *ROS2 GEM LIDAR 2D Sensor*.

IMU

Por defecto, el componente *IMU* viene configurado correctamente para su funcionamiento al usar el *Importer*, pero se puede configurar para que tenga un comportamiento similar a uno real

modificando las variaciones de su aceleración lineal, velocidad angular y de la orientación. La variación se refiere a la variabilidad aleatoria que se aplica a esos datos, simulando de esta forma el ruido que pueda tener el sensor.

LIDAR 3D

Para añadir el *LIDAR 3D* en *O3DE* se usó el componente *LIDAR Sensor* de la figura 5.4. Este componente se configuró de forma similar al *LIDAR 2D*. En sus campos de configuración se activó el tipo *CustomLidar* para poder personalizar las características del sensor según los requerimientos del proyecto. Este campo permite ajustar con precisión varios parámetros clave del sensor.

Entre los campos configurables más importantes se encuentran:

- *Layers (capas)*: define la cantidad de capas verticales que el *LIDAR* emite para obtener un escaneo tridimensional del entorno.
- *Points per layer*: especifica el número de puntos de medición que se emiten en cada capa, lo que afecta la resolución del sensor.
- *Horizontal Field of View (FOV)*: determina el ángulo de apertura horizontal del sensor, es decir, el rango angular que cubre durante la rotación.
- *Vertical Field of View (FOV)*: establece el ángulo de apertura vertical, correspondiente a la extensión de las capas desde el ángulo más bajo hasta el más alto.
- *Min Range* y *Max Range*: indican las distancias mínima y máxima a las que el *LIDAR* puede detectar objetos.

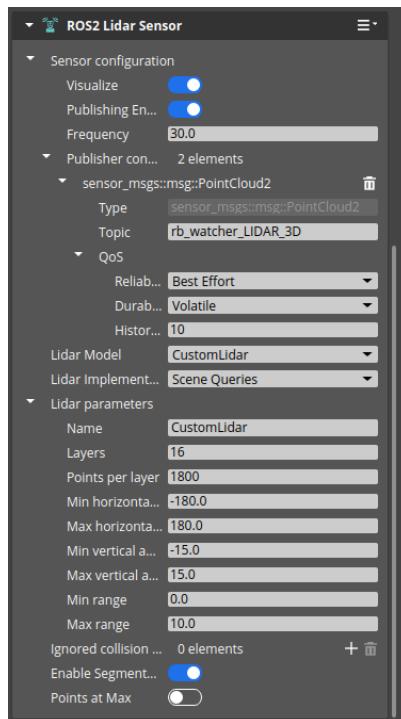


Figura 5.4: Componente de *ROS2 GEM LIDAR Sensor*.

GPS

Para implementar el GPS, se usó el componente *GNSS Sensor*, en el cual se configuraron únicamente los campos principales comunes a todos los sensores.

5.2.4 Implementación del control

Para implementar el control, fue necesario situar varios componentes del *plugin* de ROS2 en diferentes partes del robot.

En el componente raíz del robot, que habitualmente corresponde a *robot_base_footprint* (la proyección del chasis del robot sobre el plano del suelo), se añadió el componente *Robot Control* de la figura 5.5 . En este componente, se especificó el tópico a través del cual se controlará el movimiento del robot y se cambió el valor del campo *Steering* a *Twist*.

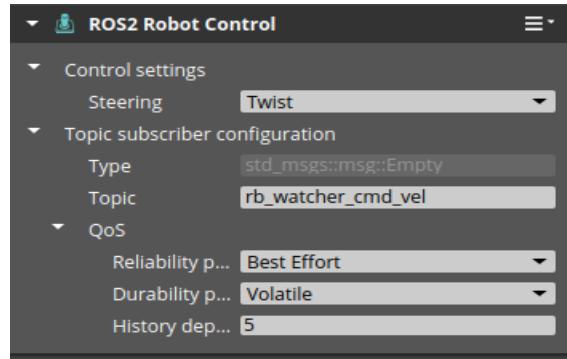


Figura 5.5: Componente de ROS2 GEM Robot Control.

Además del componente mencionado, se deben añadir en *robot_base_footprint* los componentes *Skid Steering Vehicle Model* de la figura 5.6 y *Skid Steering Twist Control* (sin parámetros configurables) .

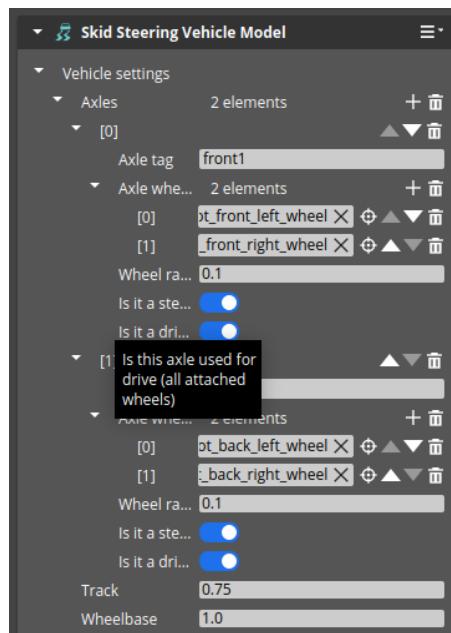


Figura 5.6: Componente de ROS2 GEM Skid Steering Vehicle Model.

En el componente *Skid Steering Vehicle Model*, se agregaron dos ejes de transmisión y se asignó un nombre para cada uno de ellos en el campo *Axle tag*.

En cada eje fue necesario especificar los objetos correspondientes a las ruedas izquierda "[0]" y derecha "[1]" buscando en la jerarquía de componentes cada rueda. También se definieron las longitudes, en metros, de:

- **Track:** distancia entre las ruedas del mismo eje, medida de lado a lado.
- **Wheelbase:** distancia entre el eje delantero y el eje trasero del vehículo.

Estas distancias pueden consultarse en las figuras 4.4 y 4.2. Adicionalmente, se especificaron las velocidades y aceleraciones máximas. Aceleraciones angulares con valores demasiado bajos resultaron insuficientes para permitir que el robot girara adecuadamente. Por último, en cada rueda fue necesario modificar el objeto *PhysX articulation link*, el cual es reconocido automáticamente por el *URDF Importer* a partir del archivo *URDF* original.

Las modificaciones realizadas a este componente, con la finalidad de obtener la mejor respuesta posible del controlador son las siguientes:

- **Local Rotation:** se ajustó para que la rueda gire en la dirección correcta, ya que por defecto suele estar desalineada.
- **Use Motor:** se activó esta opción para habilitar el control mediante motor.
- **Force Limit Value:** especifica el límite de torque que puede aplicar el motor, para cada robot se tuvo que probar mediante iteraciones el mejor valor para este campo.
- **Stiffness Value:** se configuró este valor a cero para evitar un comportamiento elástico no deseado.
- **Damping Value:** se le asignó un valor elevado de 40 ya que mediante la iteración de valores se estableció este como uno de los valores que mejor desempeño daba en ambos robots.

Los demás parámetros del componente se importaron correctamente desde el archivo *URDF* original, por lo que no se requirieron modificaciones adicionales.

5.2.5 Implementación de la odometría

La odometría en *ROS2* consiste en estimar la posición y orientación de un robot a partir de datos provenientes de sensores, como *encoders* (sensor de posición) o unidades de medición inercial (*IMU*). Esta información se publica habitualmente en el tópico */odom* utilizando mensajes del tipo *nav_msgs/Odometry*, y resulta fundamental para tareas de navegación y localización.

Para implementar esta funcionalidad en *O3DE*, se empleó el componente *Odometry Sensor*, de la figura 5.7, el cual se colocó en el componente raíz de la jerarquía de componentes del robot.

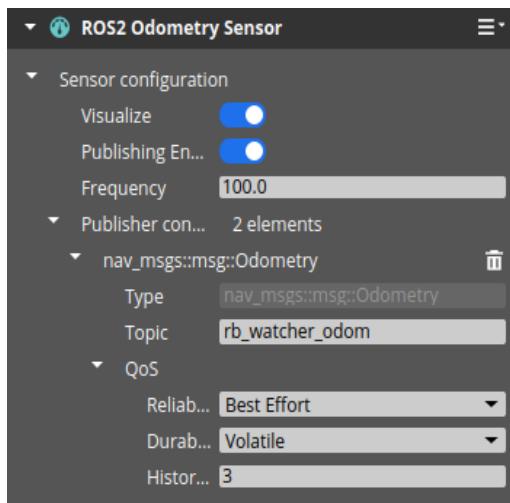


Figura 5.7: Componente de *ROS2 GEM Odometry Sensor*.

5.2.6 Implementación del *spawn* dinámico

Para habilitar el *spawn* dinámico de robots en la escena, conocida como nivel en *O3DE*, se añadió un elemento *ROS2 Spawner*, al componente raíz del nivel.

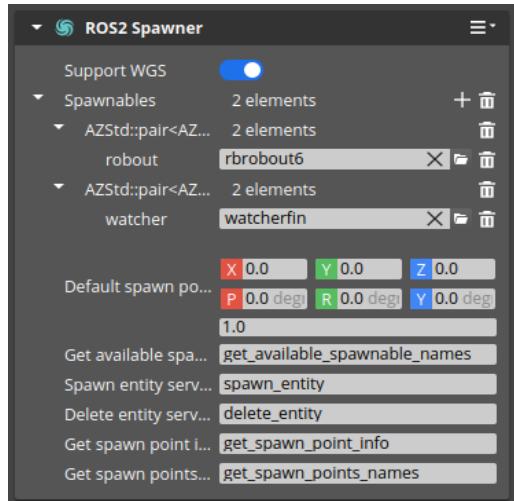


Figura 5.8: Componente de *ROS2 Spawner*.

En el componente mostrado en la figura 5.8, se agregaron los modelos de los robots previamente guardados, y se les asignó un nombre para su correcta invocación durante el proceso de *spawn*, en este caso, "robout" y "watcher". Finalmente se pudo *spawnear* dinámicamente cada robot en la escena mediante el servicio de *spawn* que usa *Gazebo* con el siguiente comando:

```
$ ros2 service call /spawn_entity gazebo_msgs/srv/SpawnEntity "{name: 'watcher', initial_pose: {position: {x: 5.0, y: -12.0, z: 0.0}, orientation: {x: 0.0, y: 0.0, z: 0.0, w: 0.1}}}"
```

5.3 Configuración y uso de paquetes de *SLAM 3D* en robots simulados

En esta sección se describen los archivos que se editaron para utilizar cada paquete de *SLAM 3D* en los diferentes simuladores usando el robot *Rb_Watcher*. Primero se explica la configuración general común a todos los simuladores y, al final de cada explicación, se detallan las particularidades específicas de cada uno.

El objetivo es emplear una configuración uniforme en todos los simuladores para asegurar la comparabilidad de los resultados. Para ello, se utilizó una configuración *SLAM 3D* que emplea únicamente los datos del *LIDAR 3D*. Aunque estos paquetes también permiten compensación de mediciones por movimientos bruscos como *lidar_deskewing* mediante *IMU*, o mapeado 3D mediante cámaras *RGBD*. Dichas características no se usaron en el análisis de resultados debido a que no son compartidas por todos los simuladores.

Todos los paquetes utilizados requirieron la configuración de tres elementos principales: el tópico del *LIDAR 3D*, el *frame* del *base_link* del robot y el *frame* global sobre el cual se genera el mapa 3D (este último no siempre es necesario).

Para visualizar el árbol completo de *frames* publicados en *ROS2*, se empleó el siguiente comando:

```
$ ros2 run tf2_tools view_frames
```

Se identificó en el resultado devuelto el nombre del *frame_global* y del *base_link* del robot.

Para listar todos los tópicos disponibles, se empleó el comando:

```
$ ros2 topic list
```

En este listado se localizó el nombre del tópico correspondiente al *LIDAR 3D*.

5.3.1 RTAB-MAP

La instalación del paquete *RTAB-MAP* [25] se realizó mediante la instalación de binarios específicos para ROS2 Humble, lo que evitó la necesidad de compilar el código manualmente.

Para configurar *RTAB-MAP*, se partió de un archivo de lanzamiento base llamado *rtabmap.launch.py*, ubicado en el directorio `/opt/ros/humble/share/rtabmap_examples/launch/`. Este archivo se adaptó para cada robot y simulador mediante modificaciones en dos aspectos principales. Primero, se actualizaron las secciones de remapeo de tópicos para que el nombre del tópico del *LIDAR 3D* coincida con el que publica el robot en el simulador correspondiente. Segundo, se ajustaron los argumentos por defecto del lanzamiento, como el nombre del *frame* del *base_link* (el *frame* del robot sobre el cual se colocan todos los demás componentes) mediante la variable (*frame_id*) y la activación del tiempo simulado añadiendo (*use_sim_time*).

El lanzamiento del paquete se realizó con el comando:

```
$ ros2 launch rtabmap_examples rtabmapNombreSimulador.launch.py
```

Una vez finalizado el mapeado, el mapa 3D se exportó desde la interfaz del visualizador de *RTAB-MAP* en formato *.ply*. Para su visualización, se utilizó un *script* externo en *Python*, que carga el archivo mediante la librería *Open3D*, permitiendo examinar el mapa generado.

5.3.2 Lidarslam_ros2

La instalación de *Lidarslam_ros2* [26] requirió clonar el repositorio oficial en un espacio de trabajo de ROS2, dentro de `~/ros2_ws/src/lidarslam_ros2`, y su posterior compilación utilizando `colcon build`.

La configuración se centró en dos archivos principales: el archivo de parámetros *lidarslam.yaml* y el archivo de lanzamiento *lidarslam.launch.py*.

En *lidarslam.yaml* se activaron opciones como el uso del tiempo simulado (*use_sim_time*) y se actualizaron los nombres del *frame* global y del *frame* base del robot en secciones clave como *scan_matcher* y *graph_based_slam*.

Por su parte, el archivo *lidarslam.launch.py* se modificó para apuntar al archivo de parámetros anterior modificado, remapear el tópico correspondiente al *LIDAR 3D* y configurar los parámetros de los nodos, incluyendo la transformación estática necesaria entre el *frame map* (sobre el cual crea el mapa el algoritmo de *RTAB-MAP*) y el *frame* global del robot simulado *odom*.

Al probar el paquete con *Webots*, se detectaron errores causados por la ausencia del campo de intensidad en los mensajes del *LIDAR 3D*. Al revisar el código fuente del nodo *scanmatcher*, se identificó una condición que verifica si falta dicho campo y que genera un error en la consola. Para solucionar este problema, se desarrolló un nodo adicional que suscribe a los mensajes del tópico del *LIDAR 3D* y les añade un valor fijo de intensidad en todos los mensajes. De esta manera, configurando el paquete para suscribirse a este nuevo tópico con intensidad, el mapeado 3D funciona correctamente. No obstante, en *RViz* (el visualizador de ROS2 que usa *Lidarslam_ros2*), la nube resultante aparece toda del mismo color debido a la intensidad constante.

En cuanto a *O3DE*, al intentar aplicar un procedimiento similar, no se logró que *Lidarslam_ros2* funcionara correctamente. Se presentó un error de desincronización relacionado con la obtención de *frames*, específicamente una extrapolación hacia el futuro: el algoritmo de *SLAM 3D* requiere un *frame* que aún no está disponible en el momento solicitado. El retardo entre el *frame* requerido y el momento de la solicitud se mantiene constante. Se probó a desactivar el tiempo simulado con

use_sim_time en falso, sin que se resolviera el problema. Al investigar la ejecución del nodo que integra el *plugin* de *ROS2* en *O3DE*, se confirmó que utiliza *use_sim_time* en falso, tal como se puede observar en la figura 5.9. También se intentó activarlo con *use_sim_time* en verdadero sin éxito.

```
/o3de_ros2_node:
  ros_parameters:
    qos_overrides:
      /parameter_events:
        publisher:
          depth: 1000
          durability: volatile
          history: keep_last
          reliability: reliable
      /tf:
        publisher:
          depth: 100
          durability: volatile
          history: keep_last
          reliability: reliable
      /tf_static:
        publisher:
          depth: 1
          history: keep_last
          reliability: reliable
  use_sim_time: false
```

Figura 5.9: Parámetros del nodo que crea *O3DE* para usar el *plugin* de *ROS2*.

El lanzamiento del sistema se realizó con el siguiente comando:

```
$ ros2 launch lidarslam lidarslam.launch.py
```

Para guardar el mapa, se invocó un servicio *ROS2* que genera un archivo en formato *.pcd*. Este archivo puede visualizarse utilizando un *script* en *Python* externo que emplea *Open3D*.

5.3.3 MOLA

El paquete *MOLA* [24] se instaló mediante paquetes binarios disponibles en el repositorio oficial de *ROS2*. A diferencia de los otros paquetes, *MOLA* no requiere modificar archivos *YAML* o archivos de lanzamiento para su configuración. En su lugar, la configuración se realiza mediante variables de entorno que especifican el *frame* base (*MOLA_TF_BASE_LINK*) y el *frame* de la odometría (*MOLA_TF_ESTIMATED_ODOMETRY*).

El paquete se lanzó con un comando que incluye argumentos para indicar el tópico del *LIDAR 3D* y el *frame* global:

```
$ export MOLA_TF_BASE_LINK=nombre_del_base_link
$ export MOLA_TF_ESTIMATED_ODOMETRY=nombre_del_frame_de_la_odometria

$ ros2 launch mola_lidar_odometry ros2-lidar-odometry.launch.py \
  lidar_topic_name:=/nombre_del_topico_del_lidar3d \
  mola_lo_reference_frame:=nombre_del_frame_global \
  forward_ros_tf_odom_to_mola:=True
```

La generación del mapa 3D se realizó a través de la interfaz gráfica que se abre al iniciar el nodo, donde el mapa se guardó inicialmente en formato *.simplemap*. Para visualizar este mapa, fue necesario convertirlo al formato *.mm* (metric map) utilizando la herramienta *sm2mm*. Esta herramienta permite además aplicar una secuencia de filtros, definidos en un archivo *.yaml*, que procesan el mapa original para mejorar su calidad y corregir posibles errores.

Durante el proceso, se detectó que los mapas generados por el robot del simulador *Gazebo Ignition* contenían valores infinitos que provocaban fallos en la visualización y un consumo excesivo de recursos, llegando incluso a detener el proceso. Para solucionar este problema, se investigaron los filtros disponibles en `sm2mm` y se aplicó un filtro que limita los puntos del mapa a un radio de 300 metros desde el origen del mapeo. Esto evitó la inclusión de valores infinitos y garantizó una visualización estable y eficiente del mapa en formato `.mm`.

6. Resultados

Este capítulo expone los resultados obtenidos tras la integración y evaluación de los robots simulados en los entornos *CoppeliaSim* y *O3DE*, empleando *ROS2* como *middleware* de comunicación. En primer lugar, se analizan los resultados de la integración de los robots con *ROS2* en ambos simuladores, destacando las funcionalidades alcanzadas, los desafíos encontrados durante el proceso y el nivel de compatibilidad logrado entre los robots de *Robotnik* y *ROS2*, lo que a su vez refleja el grado de integración de cada simulador con esta tecnología.

Posteriormente, se examina el rendimiento de los robots en términos de ejecución, capacidad de respuesta y estabilidad, tanto en escenarios con un solo robot como en configuraciones con múltiples instancias operando simultáneamente. Finalmente, se presentan los resultados relativos al rendimiento de los paquetes de *SLAM 3D*, centrados en el robot *Rb_Watcher*, que incorpora un sensor *LIDAR 3D* y permite evaluar la eficacia de estos algoritmos en entornos simulados.

6.1 Resultados de la integración de los robots con *ROS2*

En esta sección se presentan los resultados obtenidos tras la implementación de los robots móviles *Rb_Robout* y *Rb_Watcher* en los entornos de simulación *CoppeliaSim* y *O3DE*. Con el objetivo de evaluar el grado de integración y funcionalidad alcanzado en ambos entornos, se analizan y comparan diversos aspectos clave: la complejidad de la implementación, los sensores que se han podido integrar en los robots, así como las funcionalidades esenciales implementadas mediante *ROS2*, incluyendo odometría, *spawn* dinámico de robots, transformaciones (*TFs*) y control de movimiento.

Con el fin de verificar el correcto funcionamiento de los sensores, se incorpora una imagen de cada robot en su respectivo entorno de simulación, acompañada de la visualización en *RViz* de las mediciones obtenidas. En dicha visualización también se representan las transformaciones *TF*, lo que permite corroborar la coherencia espacial de los datos generados por los sensores. Esta evaluación permite identificar las fortalezas y limitaciones de cada simulador en cuanto a su compatibilidad en aplicaciones de robótica móvil con *ROS2*.

Los resultados se exponen en detalle en los apartados correspondientes a cada simulador. Para una visualización resumida y comparativa de los mismos, se presenta la Tabla 6.1.

6.1.1 Resultados en *CoppeliaSim*

En *CoppeliaSim* se ha logrado implementar correctamente los sensores principales de ambos robots, utilizando los *plugins* disponibles para simular sensores y empleando *scripts* en *Lua* para establecer la comunicación con *ROS2* desde cero. Entre los sensores implementados se incluyen: *IMU*, *GPS*, *LIDAR 2D*, *LIDAR 3D* y cámaras *RGB*. Sin embargo, no se ha podido integrar cámaras *Depth*, ya que el único *plugin* disponible correspondía a cámaras 360°, cuya utilización incrementaría significativamente la latencia del sistema. Por esta razón, se optó por prescindir de ellas, delegando el análisis de profundidad exclusivamente al sensor *LIDAR 3D*. Asimismo, no fue posible incorporar cámaras de infrarrojos, las cuales estaban presentes en el modelo original de cada robot en *Gazebo Classic* desde sus archivos *URDF*.

A continuación, en las Figuras 6.1 y 6.4 se presentan los robots *Rb_Watcher* y *Rb_Robout*, respectivamente, en la escena ligera de *CoppeliaSim*. En correspondencia con la posición de cada robot en la escena, la Figura 6.2 muestra la visualización en *RViz* de las mediciones del sensor *LIDAR 3D* del *Rb_Watcher*, mientras que la Figura 6.5 presenta las mediciones obtenidas por los dos sensores *LIDAR 2D* integrados en el *Rb_Robout*. En ambas visualizaciones también se incluyen las transformaciones *TF*, lo que permite verificar la correcta publicación de las referencias espaciales de cada robot.

Asimismo, para evidenciar el correcto funcionamiento de las cámaras *RGB*, en la Figura 6.3 se muestra la visualización captada por las cámaras del *Rb_Watcher*, y en la Figura 6.6 la correspondiente al *Rb_Robout*.

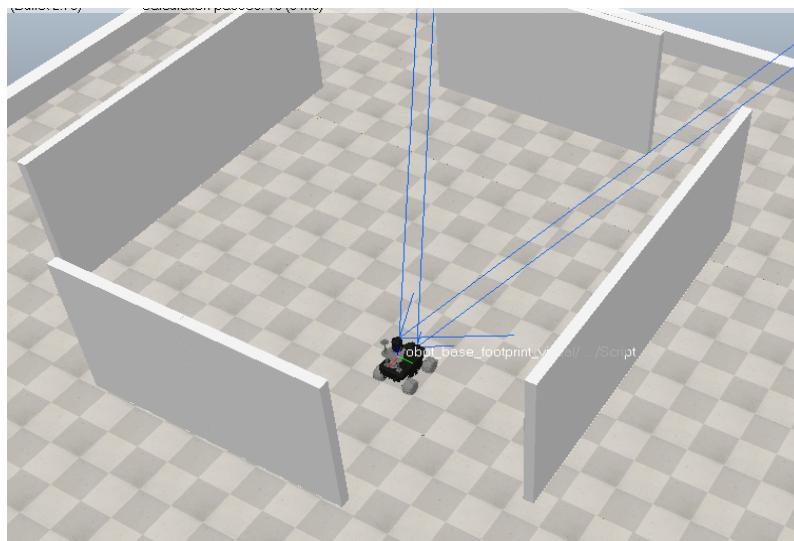


Figura 6.1: Robot *Rb_Watcher* posicionado en el entorno de simulación *CoppeliaSim*. La visualización de los rayos del sensor *LIDAR 3D* ha sido desactivada intencionadamente para mejorar la eficiencia computacional del sistema durante la simulación.

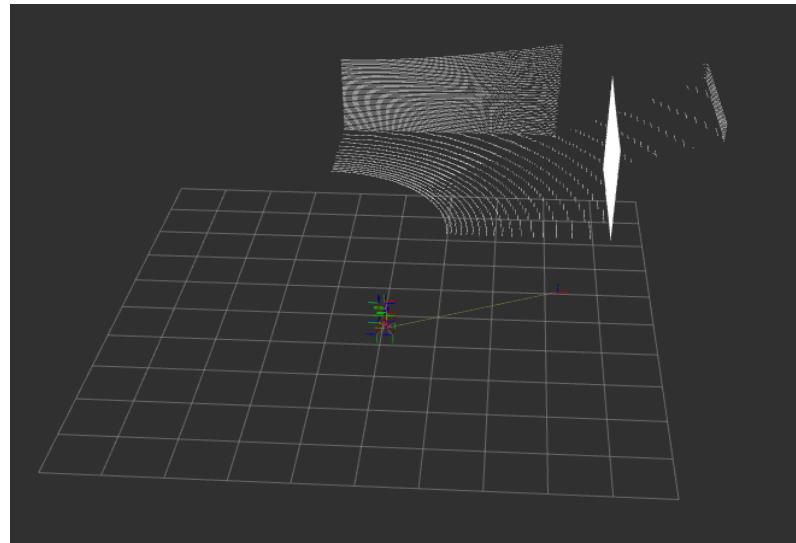


Figura 6.2: Visualización en *RViz* de las mediciones del sensor *LIDAR 3D* del robot *Rb_Watcher* en el entorno de simulación *CoppeliaSim*. Se muestra la nube de puntos generada por el sensor, junto con las transformaciones *TF* que definen el marco de referencia del robot y sus componentes.

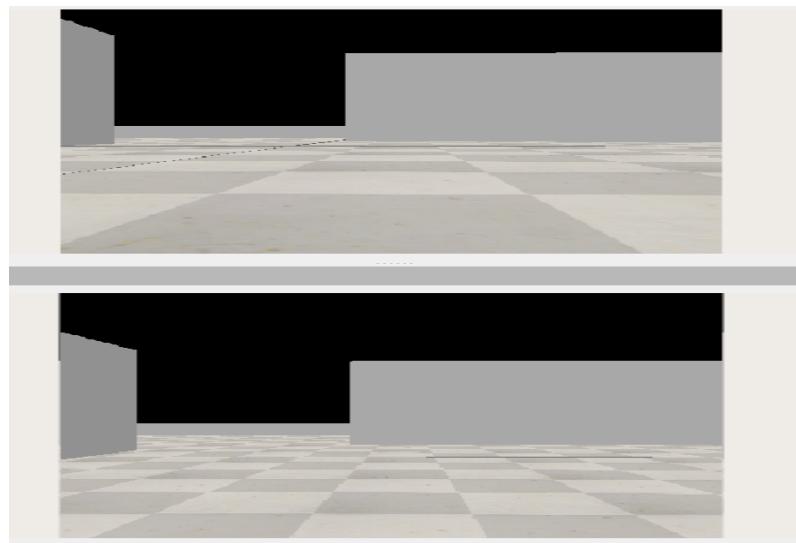


Figura 6.3: Capturas de las cámaras *RGB* del robot *Rb_Watcher* en el entorno de simulación *CoppeliaSim*. Arriba: cámara *RGB* ubicada en la parte delantera del robot. Abajo: cámara *RGB* ubicada en la parte superior del robot.

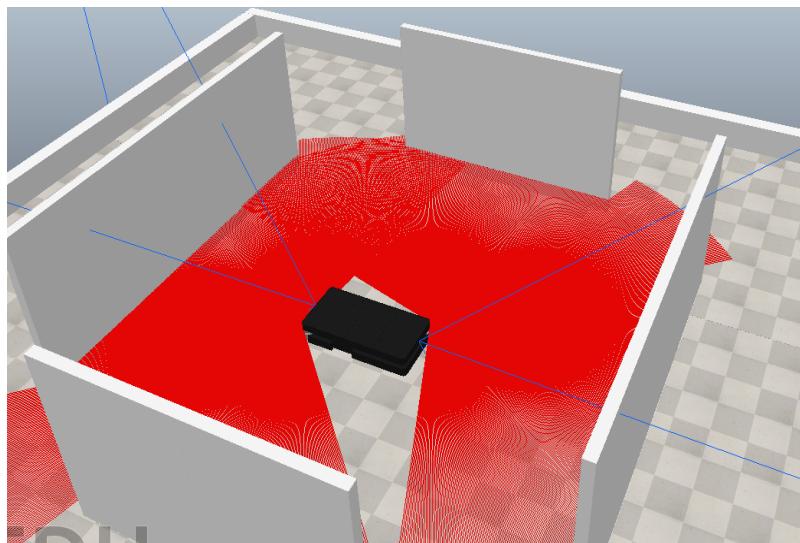


Figura 6.4: Robot *Rb_Robout* posicionado en el entorno de simulación *CoppeliaSim*.

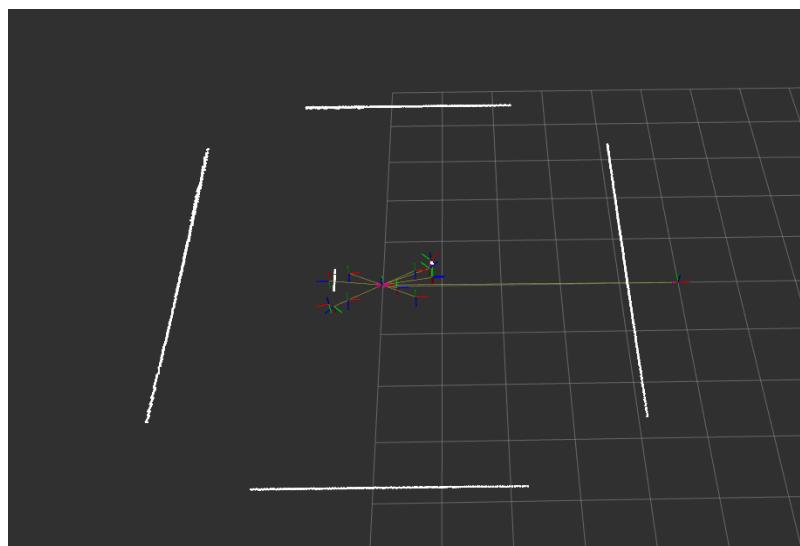


Figura 6.5: Visualización en *RViz* de las mediciones de los sensores *LIDAR 2D* del robot *Rb_Robout* en el entorno de simulación *CoppeliaSim*. Se muestran los haces láser , junto con las transformaciones *TF* que definen el marco de referencia del robot y sus componentes.

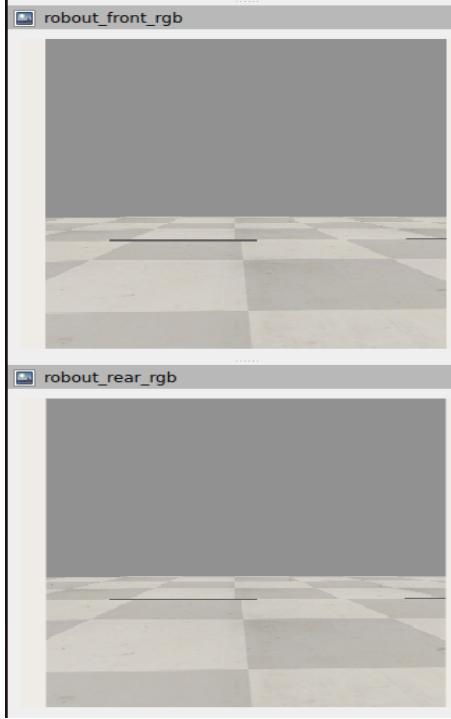


Figura 6.6: Capturas de las cámaras *RGB* del robot *Rb_Robout* en el entorno de simulación *CoppeliaSim*. Arriba: cámara *RGB* ubicada en la parte delantera del robot. Abajo: cámara *RGB* ubicada en la parte trasera del robot.

Respecto a las funcionalidades de *ROS2*, la odometría no se pudo implementar a través de un *plugin* que utilizara el giro de ruedas, debido a la inexistencia de uno compatible. En su lugar, se empleó una solución basada en la extracción de la posición del robot en la escena de *CoppeliaSim*, lo cual se asemeja a un sistema de localización tipo *GPS*, pero sin la capacidad de simular ruido en las mediciones.

La publicación de transformaciones (*TFs*) entre los distintos componentes del robot tampoco se pudo realizar mediante un *plugin* automatizado. En su lugar, fue necesario generar manualmente estas transformaciones utilizando *scripts* en *Lua*, calculando las posiciones y orientaciones relativas entre los distintos elementos de la escena.

En cuanto al *spawn* dinámico de robots, aunque el simulador dispone de una API que permite la inserción dinámica de modelos, se debió desarrollar, además del *script* para *spawn*, un sistema adicional que asignase a cada robot un *namespace* distinto. Esto fue necesario para evitar conflictos en los nombres de los tópicos y de los marcos de referencia (*frames*). El sistema resultante se implementó correctamente y demostró ser funcional.

Respecto al control de movimiento, no se encontró ningún *plugin* compatible con *ROS2*, ya que el único disponible era para *ROS 1* y no ha sido actualizado desde 2019. Por tanto, se desarrolló un sistema de control desde cero, utilizando las ecuaciones cinemáticas del robot, implementadas mediante *scripts* en *Lua*. Este sistema permite un control funcional a través del tópico estándar */cmd_vel*.

En términos generales, el grado de integración con *ROS2* en *CoppeliaSim* se considera bajo, dado que fue necesario implementar desde cero la mayoría de las funcionalidades, partiendo únicamente de la capacidad básica de comunicación por tópicos. No obstante, a nivel de funcionalidad del robot, se consiguió implementar la mayoría de las capacidades previstas, aunque algunos sensores no pudieron añadirse. Cabe destacar que el sensor *LIDAR 3D* presenta una latencia elevada, lo que hace inviable su uso en tareas prácticas en *ROS2*, como se detallará en la siguiente sección.

6.1.2 Resultados en O3DE

En *O3DE* también se logró satisfactoriamente la implementación de los sensores básicos del robot. A diferencia de *CoppeliaSim*, en este entorno sí fue posible integrar las cámaras *Depth*, mejorando así la percepción tridimensional del entorno. Al igual que en *CoppeliaSim*, no fue posible integrar las cámaras de infrarrojos.

A continuación, en las Figuras 6.7 y 6.10, se presentan los robots *Rb_Watcher* y *Rb_Robout*, respectivamente, posicionados en la escena ligera del entorno de simulación *O3DE*. En correspondencia con la configuración sensorial de cada plataforma, la Figura 6.8 muestra la visualización en *RViz* de las mediciones obtenidas mediante el sensor *LIDAR 3D* del *Rb_Watcher*, mientras que la Figura 6.11 presenta los datos captados por los dos sensores *LIDAR 2D* del *Rb_Robout*. En ambas visualizaciones se incluyen las transformaciones *TF*, que permiten verificar la correcta publicación de los marcos de referencia de cada componente robótico.

Por otro lado, para evaluar el sistema de percepción visual, se capturaron las salidas de las cámaras integradas en cada robot. La Figura 6.9 muestra las imágenes captadas por las cámaras del *Rb_Watcher*, que incluyen una cámara de profundidad y dos cámaras RGB, distribuidas en posiciones estratégicas del robot. En cuanto al *Rb_Robout*, las Figuras 6.12 y 6.13 ilustran las vistas captadas por sus cámaras frontales y traseras, las cuales también combinan sensores RGB y de profundidad.

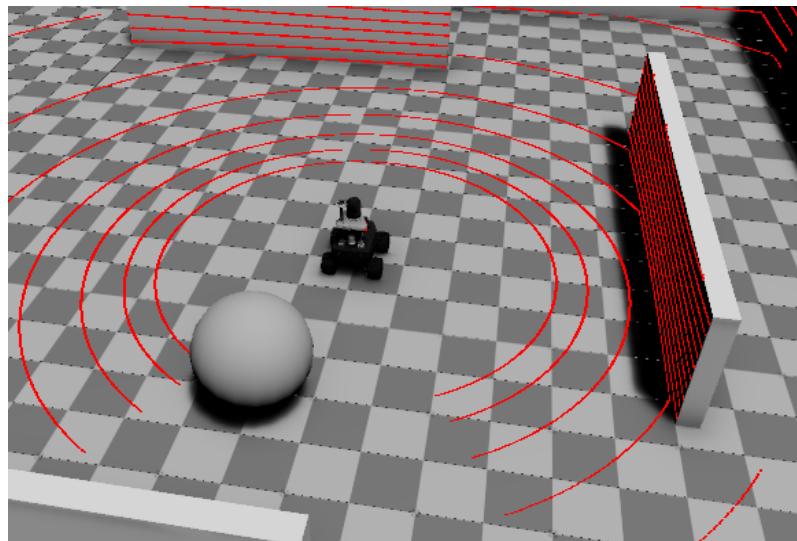


Figura 6.7: Robot *Rb_Watcher* posicionado en el entorno de simulación *O3DE*.

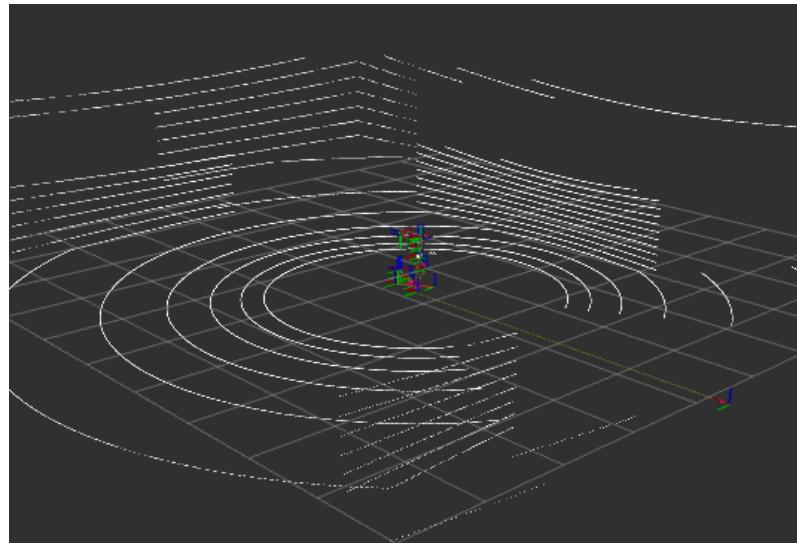


Figura 6.8: Visualización en *RViz* de las mediciones del sensor *LIDAR 3D* del robot *Rb_Watcher* en el entorno de simulación *O3DE*. Se muestra la nube de puntos generada por el sensor, junto con las transformaciones *TF* que definen el marco de referencia del robot y sus componentes.



Figura 6.9: Capturas de las cámaras *RGBD* del robot *Rb_Watcher* en el entorno de simulación *O3DE*. Arriba: cámara de profundidad ubicada en la parte delantera del robot. En medio: cámara *RGB* ubicada en la parte superior del robot. Abajo:cámara *RGB* ubicada en la parte delantera del robot.

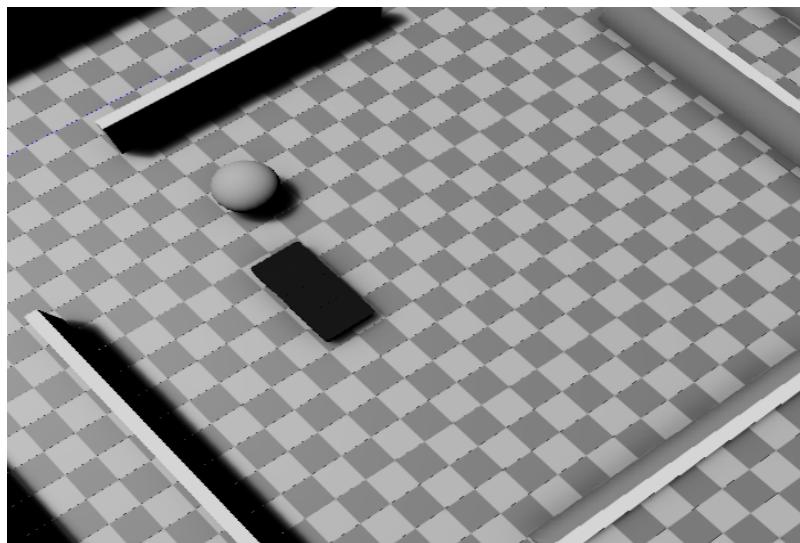


Figura 6.10: Robot *Rb_Robout* posicionado en el entorno de simulación *O3DE*.

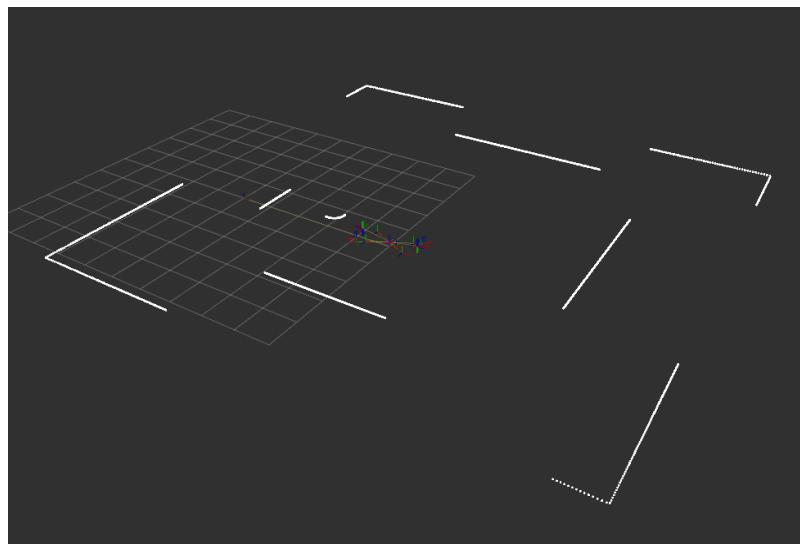


Figura 6.11: Visualización en *RViz* de las mediciones de los sensores *LIDAR 2D* del robot *Rb_Robout* en el entorno de simulación *O3DE*. Se muestran los haces láser , junto con las transformaciones *TF* que definen el marco de referencia del robot y sus componentes.

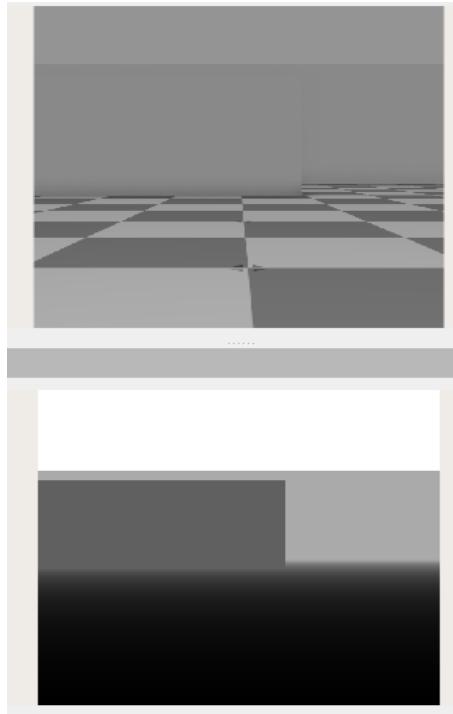


Figura 6.12: Capturas de las cámaras frontales RGB y de profundidad del robot *Rb_Robout* en el entorno de simulación *O3DE*. Arriba: cámara RGB ubicada en la parte delantera del robot. Abajo: cámara de profundidad ubicada en la parte delantera del robot.

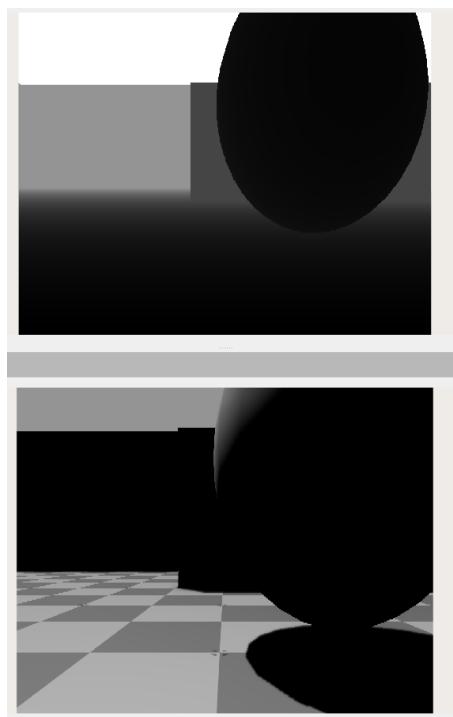


Figura 6.13: Capturas de las cámaras traseras RGB y de profundidad del robot *Rb_Robout* en el entorno de simulación *O3DE*. Arriba: cámara de profundidad ubicada en la parte trasera del robot. Abajo: cámara RGB ubicada en la parte trasera del robot.

En cuanto a las funcionalidades de *ROS2*, la odometría fue implementada utilizando el *plugin ROS2 Odometry Sensor*, el cual funciona correctamente estimando la posición del robot mediante el giro de sus ruedas, lo que representa una implementación más realista que en *CoppeliaSim*.

Las transformaciones (*TFs*) de todos los componentes definidos en el archivo *URDF* se reconocen automáticamente y se transmiten a *ROS2* sin necesidad de intervención adicional, permitiendo modificar sus nombres si se desea.

Respecto al *spawn* dinámico, se utilizó el *plugin ROS2 Spawner*, que, una vez configurado, permite insertar robots dinámicamente en la escena, asignándoles automáticamente un *namespace* único mediante el servicio de *spawn* compatible con *Gazebo*.

Para el control de movimiento, se dispone de dos tipos de controladores dentro del *plugin* de *ROS2*: *skid steering* y *Ackermann*. No obstante, ambos presentan problemas de estabilidad, ya que tienden a saturarse y dejar de funcionar correctamente tras un corto periodo de tiempo. Se intentó mejorar su rendimiento ajustando diversos parámetros del controlador, modificando las fricciones de las ruedas y probando diferentes colisionadores (mallas reales y cilindros simplificados), pero estos esfuerzos no produjeron mejoras significativas.

Finalmente, el grado de integración con *ROS2* en O3DE se considera alto, ya que la mayoría de sensores y funcionalidades *ROS2* están incluidas en un único *plugin*, y la mayor parte del trabajo se reduce a la configuración de los componentes. No obstante, el controlador de movimiento representa un punto débil importante, limitando la utilidad práctica del robot en entornos de simulación prolongados o exigentes.

Cuadro 6.1: Comparativa de funcionalidades implementadas en *CoppeliaSim* y *O3DE*.

Aspecto evaluado	CoppeliaSim	O3DE (Open 3D Engine)
Sensores RGB	Implementados correctamente	Implementados correctamente
Sensores <i>Depth</i>	No implementados	Implementados correctamente
LIDAR 2D y 3D	Implementados (latencia alta en <i>LIDAR 3D</i>)	Implementados correctamente
IMU y GPS	Implementados	Implementados
Cámaras infrarrojas	No implementadas	No implementadas
Odometría	Estimada a partir de la posición en la escena (no realista)	<i>Plugin</i> de odometría con estimación basada en ruedas
TFs	Generadas manualmente mediante <i>scripts</i> en <i>Lua</i>	Generadas automáticamente desde <i>URDF</i>
<i>Spawn</i> dinámico	Implementado manualmente con sistema de <i>namespaces</i>	Implementado con <i>plugin ROS Spawner</i> (<i>namespace</i> automático)
Control de movimiento	Implementado mediante cinemática inversa con <i>Lua</i> (funcional)	<i>Plugins</i> disponibles (<i>Skid Steering</i> y <i>Ackermann</i>), pero se saturan y fallan
Grado de integración con ROS2	Bajo (gran parte implementado desde cero)	Alto (integración directa con <i>plugins</i> de <i>ROS2</i>)

6.2 Resultados del rendimiento de los robots

Para obtener los resultados de rendimiento de los robots que se presentan en las tablas 6.2 y 6.3, se siguió un procedimiento sistemático en cada simulador. En primer lugar, se crearon dos escenas

diferentes: una escena vacía y otra con paredes, diseñadas para que los sensores pudieran detectar obstáculos y que a su vez fueran fácilmente replicables en todos los simuladores. A continuación, se cargó la escena correspondiente en cada simulador y se procedió a lanzar los robots de forma secuencial, uno por uno. Con cada robot *spawnreado*, se registraron las estadísticas relevantes mediante el comando `top`, para medir el consumo de *CPU* y *RAM*.

En lo que respecta al uso de *GPU*, en *O3DE* esta información es proporcionada directamente por el simulador en pantalla, junto con los *FPS* obtenidos. En el caso de *CoppeliaSim*, el uso de *GPU* se obtuvo empleando el comando de *Nvidia* correspondiente.

El factor de tiempo real (*RTF*), que indica cuán rápido se ejecuta la simulación respecto al tiempo real, y la latencia sensorial fueron suministrados por la interfaz de *CoppeliaSim*. No obstante, en *O3DE* la latencia se calculó de forma indirecta. Para ello, se midió la frecuencia real del sensor más significativo del robot, que en el caso del robot *Rb_Robout* correspondió a uno de sus *LIDAR 2D*, mientras que para el robot *Rb_Watcher* la medición se realizó sobre el *LIDAR 3D*. Esta frecuencia real fue comparada con la frecuencia teórica asignada a dicho sensor dentro del simulador. Aplicando la fórmula (6.1), se determinó la latencia en milisegundos. Esta medición se realizó cada vez que se *spawnneaba* un nuevo robot, registrando así la latencia específica del sensor para cada configuración.

$$\text{Latencia (ms)} = \left(\frac{1}{\text{Frecuencia real}} - \frac{1}{\text{Frecuencia teórica}} \right) \times 1000 \quad (6.1)$$

6.2.1 Rendimiento de los robots en *O3DE*

Cuadro 6.2: Rendimiento de los robots en *O3DE*.

Escenario	Robot	N.º Robots	Startup Time (s)	RTF (%)	FPS	Latencia (ms)	RAM (GB)	CPU (%)	GPU (MB)
Sin escenario	Rb_Watcher	1	0.4	-	61.2	17	1.6	203	616
		2	1.30	-	60.3	28	1.7	194	708
		3	1.92	-	43	56	1.9	206	900
	Rb_Robout	1	0.6	-	59.2	20	2	208	1272
		2	1.54	-	37.2	24	2.1	231	1831
		3	2.2	-	10.2	32	2.4	319	2780
Escenario ligero	Rb_Watcher	1	0.42	-	60.5	17	2.1	172	580
		2	1.27	-	57.8	23	2.1	200	670
		3	1.9	-	53.2	70	2.2	201	828
	Rb_Robout	1	0.43	-	61.5	16	1.8	210	1305
		2	1.23	-	53	33	1.9	247	2004
		3	2.1	-	9.5	28	2.1	301	2877

Cuadro 6.3: Rendimiento de los robots en *CoppeliaSim*.

Escenario	Robot	N.º Robots	Startup Time (s)	RTF (%)	FPS	Latencia (ms)	RAM (GB)	CPU (%)	GPU (GB)
Sin escenario	Rb_Watcher	1	1.3	58	-	126	0.93	100.3	0
		2	6.8	7.6	-	620	0.92	100	0
		3	9.8	7.3	-	763	1	100.3	0
	Rb_Robout	1	5.3	52	-	25	0.97	112	0
		2	7.7	41	-	63	0.99	103	0
		3	11.2	7.2	-	104	1.1	110	0
Escenario ligero	Rb_Watcher	1	1.5	46	-	403	1	103	0
		2	8.4	25	-	1190	1	103	0
		3	16	14	-	1526	1	100	0
	Rb_Robout	1	2.41	77	-	25	0.72	102	0
		2	8	48	-	64	0.9	100	0
		3	14.4	44	-	111	1.4	125	0

Como se observa en la tabla 6.2, el campo correspondiente al *RTF* está vacío, dado que esta funcionalidad no está implementada en *O3DE*. En cuanto al *Startup Time* —es decir, el tiempo que tarda en cargarse un robot dentro del simulador— este proceso es prácticamente inmediato,

dependiendo principalmente del tiempo que el usuario invierte en introducir manualmente el comando necesario para añadir cada robot.

Respecto a los *FPS*, con un solo robot, independientemente del modelo, se mantienen cercanos al máximo, alrededor de 60 *FPS*, con pequeñas fluctuaciones de aproximadamente 1 *FPS*. Sin embargo, a medida que aumenta el número de robots, esta tasa disminuye notablemente, siendo los valores más bajos observados al simular tres unidades del modelo *Rb_Robout*.

En relación a la latencia, tal como se muestra en la figura 6.14, con dos robots, sin importar el modelo, esta se mantiene por debajo de los 35 ms, un valor aceptable para la mayoría de aplicaciones robóticas que requieren colaboración entre dos unidades. Sin embargo, se ha identificado que al simular tres unidades de *Rb_Watcher*, la latencia aumenta de manera significativa, llegando a duplicar los valores previos con la adición de un solo robot extra.

Por último, en cuanto al uso de *CPU*, se observa que el simulador aprovecha múltiples núcleos, alcanzando un consumo equivalente a tres núcleos cuando se simulan tres robots del tipo *Rb_Robout*.

Además de estos aspectos, se han descubierto otros hallazgos relevantes que aportan información valiosa para optimizar el uso de los robots y los simuladores, contribuyendo a una mejor planificación y desempeño en aplicaciones robóticas futuras.

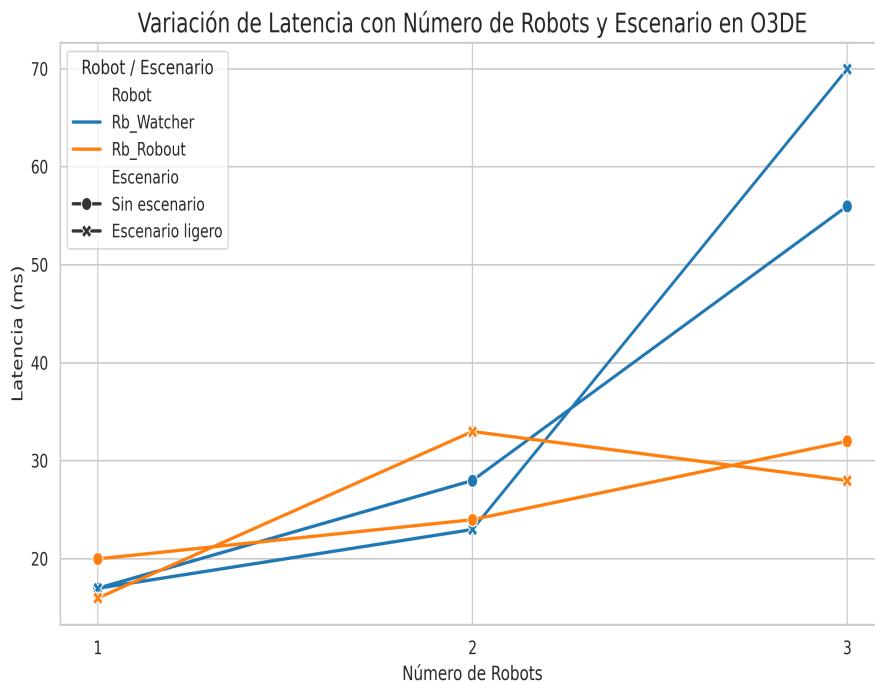


Figura 6.14: Variación de la latencia (en milisegundos) en función del número de robots para los modelos *Rb_Watcher* y *Rb_Robout* en el motor *O3DE*, comparando dos escenarios: sin escenario y escenario ligero.

En primer lugar, se ha observado que no existe una correlación evidente entre los valores de latencia y los *FPS*, tal como se muestra en la figura 6.15. Esta falta de relación directa podría atribuirse a la arquitectura interna del motor *O3DE*, donde el *plugin* de comunicación con *ROS 2* se encuentra integrado de manera nativa. A diferencia de soluciones basadas en puentes intermedios, esta integración directa podría favorecer una separación más clara entre los ciclos gráficos y las operaciones de comunicación con *ROS2*, permitiendo así que ambos procesos se desarrollen de forma más independiente y con menor interferencia mutua. En lo relativo al uso de la GPU, se han

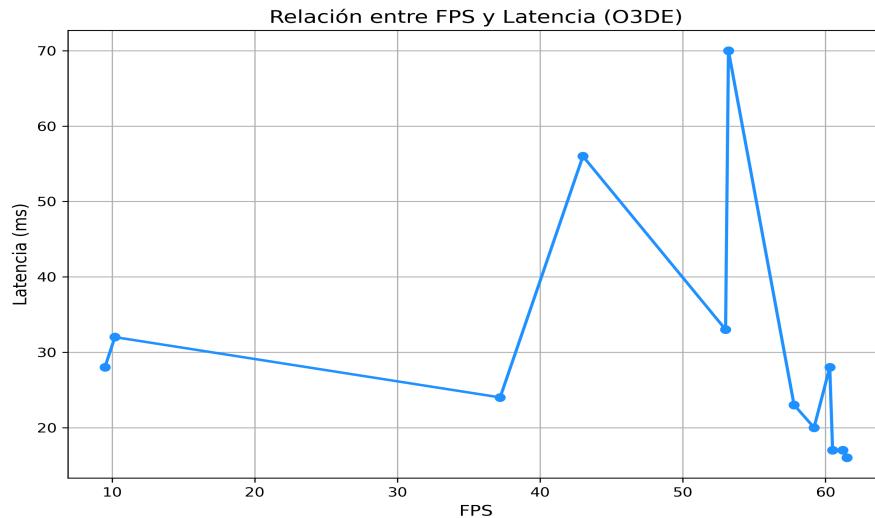


Figura 6.15: Distribución de los valores de latencia en función de la frecuencia de fotogramas por segundo (*FPS*) en el motor *O3DE*.

identificado comportamientos contrastantes entre ambos modelos de robot tal como se refleja en la figura 6.16. Por un lado, el *Rb_Watcher* presenta un consumo menor de GPU en el escenario ligero en comparación con el escenario vacío. Este resultado resulta, a priori, contraintuitivo, dado que un entorno más complejo suele implicar un mayor procesamiento gráfico. Por otro lado, el *Rb_Robout* exhibe un comportamiento coherente con lo esperado, incrementando su consumo de GPU al ejecutarse en un entorno más cargado visualmente.

Además, se ha constatado que el *Rb_Robout* utiliza significativamente más recursos de *GPU* que el *Rb_Watcher*, con un consumo medio aproximadamente un 174 % superior.

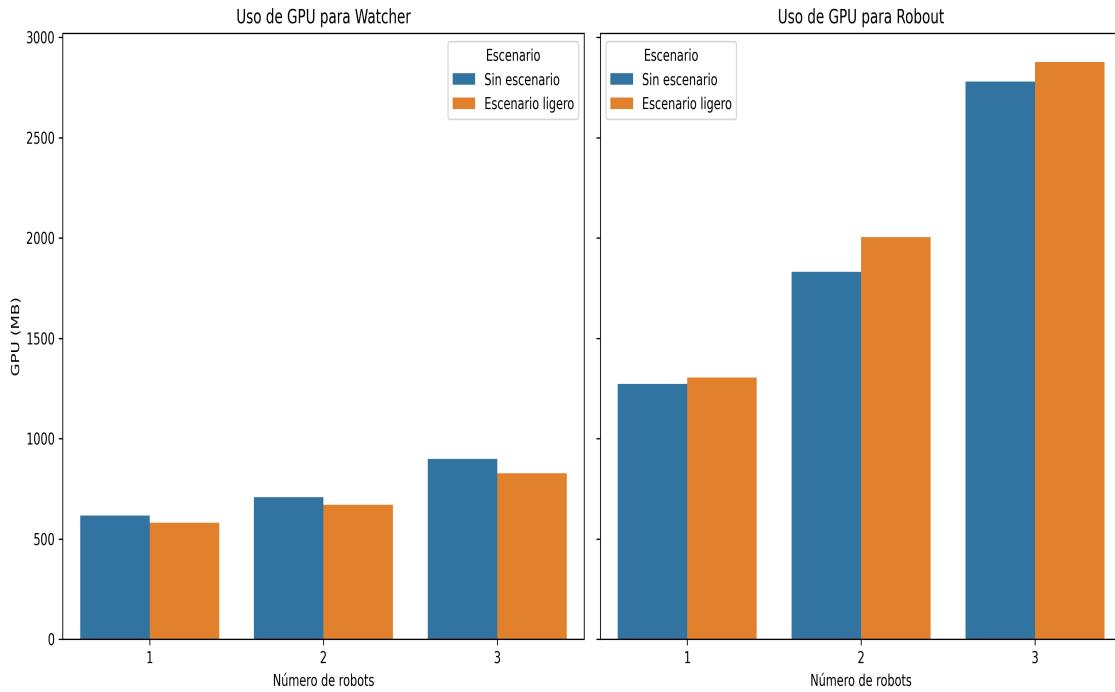


Figura 6.16: Comparación del uso de *GPU* (en *MB*) entre los robots *Rb_Watcher* y *Rb_Robout* en dos configuraciones del motor *O3DE*: escenario vacío y escenario ligero.

Otro hallazgo destacable es la capacidad de carga que el motor *O3DE* es capaz de manejar en términos de número de instancias de robot sin comprometer el rendimiento gráfico. En particular, se ha comprobado, tal como muestra la figura 6.17, que es posible simular hasta tres unidades del *Rb_Watcher* en ambos escenarios manteniendo una tasa de actualización por encima del umbral mínimo aceptable de 30 *FPS*. En el caso del *Rb_Robout*, esta capacidad se reduce a dos unidades, superando también dicho umbral.

6.2.2 Rendimiento de los robots en *CoppeliaSim*

En cuanto al rendimiento de los robots *Robotnik* en *CoppeliaSim*, se observa en la figura 6.18 que la latencia tiende a incrementarse conforme aumenta el número de robots, siendo especialmente notable en el caso del *Rb_Watcher*, donde la latencia presenta un aumento abrupto al añadir un segundo robot. Además, la latencia de un único *Rb_Watcher* en el escenario ligero es más del doble que en el escenario vacío. De hecho, la latencia de un solo *Rb_Watcher* en el escenario ligero es aproximadamente cuatro veces mayor que la de un *Rb_Robout* en el mismo entorno. Este comportamiento indica que la implementación de un *LIDAR 3D* con comunicación a *ROS2* en el *Rb_Watcher*, y por tanto en el propio simulador, no resulta factible debido al elevado retardo en la transmisión de datos.

En cuanto al *Startup Time*, a medida que disminuye el *RTF*, aumenta el tiempo para generar los robots, ya que la implementación interna de los mismos presenta un pequeño retardo para establecer comunicación con *ROS2* tras cierto tiempo. Esto se debe a que, durante las primeras iteraciones de su bucle infinito, no todos los sensores están disponibles.

Respecto al *RTF*, los robots generados individualmente muestran una media de aproximadamente 58.25 %, lo que significa que la simulación se ejecuta a poco más de la mitad del tiempo real. Sin embargo, al generar un segundo robot, el *RTF* cae drásticamente a valores por debajo del 20 %, salvo en el caso del *Rb_Robout*, como se observa en la Figura 6.5. Por tanto, *CoppeliaSim* no

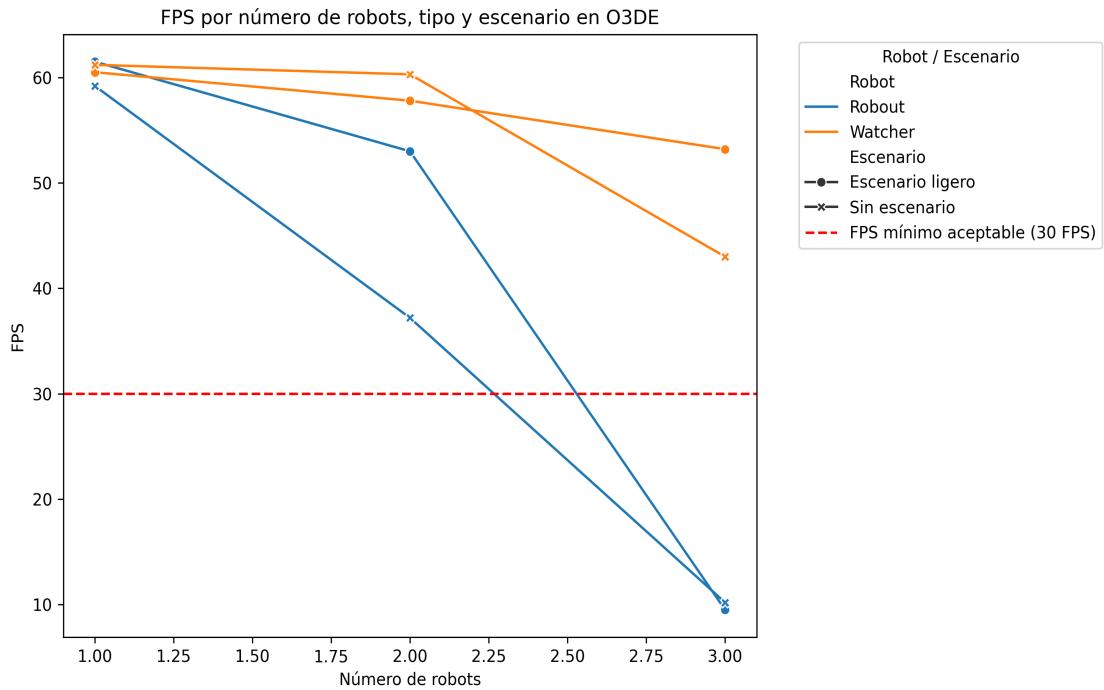


Figura 6.17: Evolución de la tasa de fotogramas por segundo (*FPS*) en función del número de robots instanciados para los modelos *Rb_Watcher* y *Rb_Robout*. Se comparan los resultados en dos configuraciones de entorno: sin escenario y con escenario ligero. La línea discontinua roja indica el umbral mínimo de rendimiento aceptable (30 *FPS*).

posee capacidad efectiva para simulación multirobot.

En cuanto al uso de la *CPU*, se emplea únicamente un núcleo, lo cual, junto con la ausencia de soporte para *GPU*, puede explicar el bajo rendimiento observado. Finalmente, el campo de *FPS* aparece vacío en la tabla, dado que el simulador no proporciona información sobre esta métrica.

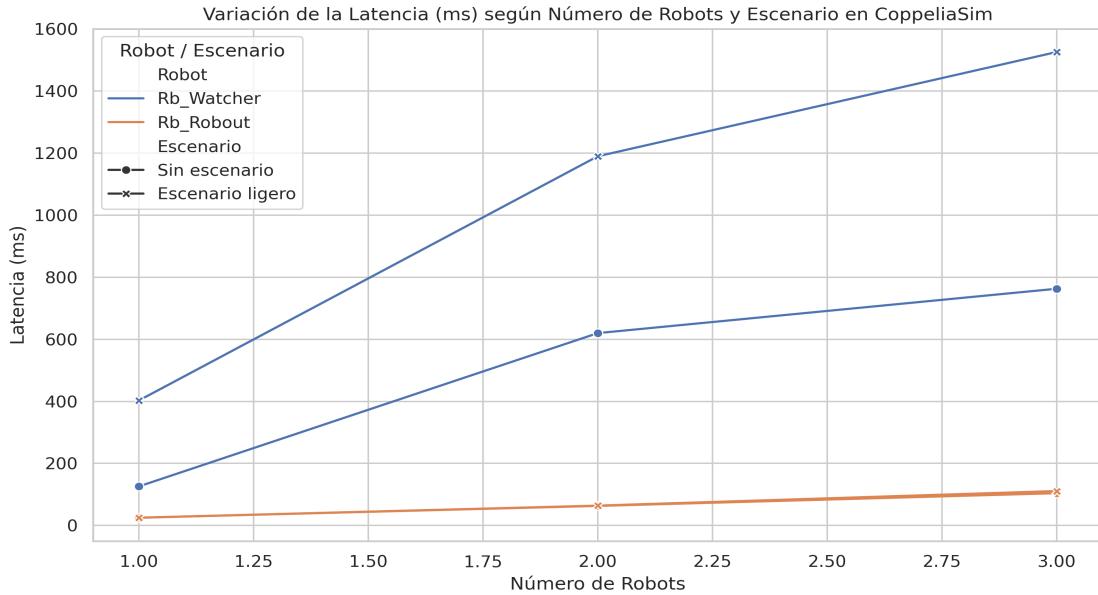


Figura 6.18: Variación de la latencia en milisegundos (ms) en función del número de robots para los modelos *Rb_Watcher* y *Rb_Robout* en dos escenarios distintos en *CoppeliaSim*.

Con el objetivo de determinar el origen del elevado retardo observado en el robot *Rb_Watcher* durante las simulaciones en *CoppeliaSim*, se ha llevado a cabo una comparación entre dos configuraciones distintas: una con el *LIDAR 3D* conectado a *ROS2* mediante el *script* de comunicación con *ROS2*, y otra con el mismo sensor pero sin establecer dicha comunicación.

La figura 6.19 muestra, para ambas configuraciones, la evolución del *RTF* y de la latencia a medida que aumenta el número de robots desplegados en el entorno vacío. El propósito de este análisis es aislar el impacto que la comunicación con *ROS2* puede tener sobre el rendimiento del simulador y sobre la latencia percibida en la transmisión de datos sensoriales.

Los resultados de la figura 6.19 revelan que, en ausencia de comunicación con *ROS2*, el robot mantiene un *RTF* elevado (por encima del 30% incluso con tres robots) y una latencia considerablemente menor en todos los casos. En particular, con un solo robot, la latencia se reduce de 126 ms a tan solo 40 ms, lo cual representa una mejora sustancial. Este patrón se mantiene al aumentar el número de robots, indicando que el uso del *script* de comunicación con *ROS2* es el principal responsable del incremento de latencia observado en la configuración original.

Este hallazgo es relevante, ya que permite concluir que el cuello de botella en la transmisión de datos no se debe únicamente al procesamiento del *LIDAR 3D*, sino que está estrechamente relacionado con la arquitectura de comunicación *ROS2* implementada en el simulador. Dicho de otro modo, el sistema de publicación y suscripción utilizado para enviar los datos del sensor desde el simulador a *ROS2* introduce un retardo adicional significativo.

En consecuencia, esta información resulta esencial para futuras decisiones de diseño.

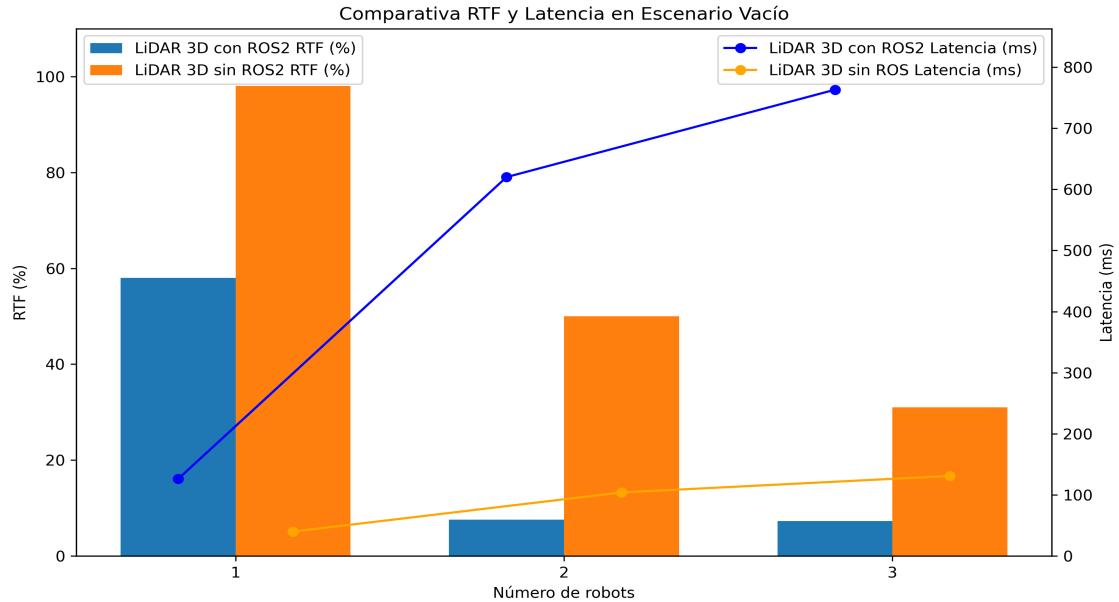


Figura 6.19: Comparativa del *RTF* y la latencia del *Rb_Watcher* en el escenario vacío, con y sin comunicación a *ROS2*.

La tabla 6.4 presentada a continuación muestra una evaluación exhaustiva del rendimiento del robot *Rb_Watcher* en diversos simuladores robóticos, incluyendo *Gazebo Ignition*, *O3DE*, *CoppeliaSim*, *Isaac Sim*, *Unity* y *Webots*. Esta evaluación se llevó a cabo en dos tipos de escenarios —vacío y ligero— y con configuraciones de uno, dos y tres robots, para analizar cómo se comportan las diferentes plataformas bajo condiciones crecientes de complejidad.

El análisis se centra en métricas clave como el tiempo de arranque, factor de tiempo real, *frames* por segundo, uso de recursos del sistema (*RAM*, *CPU*, *GPU*) y especialmente la latencia sensorial. Esta última es fundamental para aplicaciones que requieren una rápida respuesta y sincronización precisa entre sensores y controladores en simulaciones robóticas.

Esta tabla fue elaborada en el marco de las prácticas profesionales en la empresa *Robotnik Automation* y es fruto del trabajo conjunto entre los estudiantes *Joel Ramos*, *Lucía Esteve* y el autor de este informe.

Cuadro 6.4: Comparativa de simuladores con robot *Rb_Watcher*.

Simulador	Startup time (s)	RealTimeFactor(%)	FPS	RAM (GB)	CPU (%)	GPU (GB)	Sensor latency (ms)
Sin escenario							
Gazebo Ignition (1)	110	90	59	1.995	290	0.76	92.97
Gazebo Ignition (2)	222	85	59	2.182	280	1.00	135.85
Gazebo Ignition (3)	300	60	59	2.400	298	1.31	275.57
O3DE (1)	1.35	—	60	3.10	128.9	0.62	16.2
O3DE (2)	1.35	—	59.7	3.20	125.1	0.70	17.6
O3DE (3)	1.35	—	52.2	3.40	137.2	0.71	100
Coppelia (1)	0.7	55	—	0.86	100.2	0.03	130
Coppelia (2)	1.5	23	—	0.87	100.0	0.04	350
Coppelia (3)	3	13	—	0.88	100.1	0.06	1050
Isaac Sim (1)	17.47 + 2.01	—	44.93	5.22	1317	1.66	19.86
Isaac Sim (2)	16.45 + 2.04	—	35.71	5.40	1418	2.15	26.25
Isaac Sim (3)	17.11 + 2.28	—	34.28	5.69	1532	2.63	31.58
Unity (1)	1.01 + 7.04	90.1	60	4.75	47.8	0.74	—
Unity (2)	1.23 + 8.3	90.32	60	4.84	240.2	0.73	—
Unity (3)	1.15 + 8.4	90.10	60	5.03	247.1	0.71	—
Unity (4)	1.02 + 9.16	89.90	60	5.25	123.9	0.63	—

Simulador	Startup time (s)	RealTimeFactor(%)	FPS	RAM (GB)	CPU (%)	GPU (GB)	Sensor latency (ms)
Webots (1)	2.35	99	—	0.94	26.6	0.43	29.74
Webots (2)	3.2	98	—	1.20	43.7	0.47	30.23
Webots (3)	2.8	98	—	1.40	56.2	0.36	30.11
Escenario ligero							
Gazebo Ignition (1)	150 + 3.39	90	59	1.993	270	0.76	85.45
Gazebo Ignition (2)	280 + prev	60	59	2.189	270	1.00	172.74
Gazebo Ignition (3)	360 + prev	50	59	2.489	278	1.31	325.51
O3DE (1)	0.98	—	61.37	2.40	143	0.62	17.6
O3DE (2)	0.98	—	59.9	2.60	128.9	0.66	23.4
O3DE (3)	0.98	—	52.6	2.80	138.6	0.76	88.6
Coppelia (1)	1	20	—	0.87	100.1	0.03	700
Coppelia (2)	2.5	9	—	0.92	100.2	0.04	1800
Coppelia (3)	4.5	4	—	0.97	100.3	0.06	4800
Isaac Sim (1)	16.18 + 1.16	—	39.03	6.18	1322	2.25	19.92
Isaac Sim (2)	16.09 + 2.15	—	23.96	6.34	1432	2.40	25.32
Isaac Sim (3)	16.24 + 1.32	—	15.14	7.09	1634	3.00	29.01
Unity (1)	1.49 + 10.29	89.56	60	2.89	223.9	0.46	—
Unity (2)	1.36 + 11.01	90.63	60	2.90	238.4	0.42	—
Unity (3)	1.04 + 13.18	90.52	60	3.09	245.2	0.41	—
Unity (4)	1.18 + 7.23	88.89	60	5.40	265.22	0.36	—
Webots (1)	2.74 + 2.14	99	—	1.00	27.6	0.47	29.45
Webots (2)	3.25 + prev	98	—	1.30	43.7	0.49	29.90
Webots (3)	4 + prev	96	—	1.50	47.1	0.37	30.16
Webots (4)	3*10 + 2.14	96	—	2.71	141.6	0.83	30.10

La figura 6.20 ofrece una representación visual de la latencia sensorial del robot *Rb_Watcher* en cada simulador evaluado, con configuraciones de uno, dos y tres robots. Este gráfico facilita la comparación del rendimiento temporal de los simuladores y permite observar cómo escala la latencia con el aumento del número de robots, un factor crítico en simulaciones multirrobot y aplicaciones en tiempo real.

Los datos muestran que *O3DE* es uno de los simuladores más consistentes en términos de latencia, manteniéndola muy baja y estable, entre 16 y 18 ms con uno y dos robots, y aumentando significativamente solo al simular tres robots, alcanzando 100 ms. Además, *O3DE* destaca como uno de los simuladores que logra mantener los *FPS* más altos, superando los 60 *FPS* en escenarios con hasta dos robots, lo que garantiza una simulación visual fluida y precisa.

En contraste, *CoppeliaSim* presenta latencias sensoriales elevadas que crecen de forma pronunciada con el número de robots, desde 130 ms con un robot hasta valores superiores a 1000 ms con tres robots en escenarios vacíos. Este incremento es aún más acusado en escenarios con carga ligera, llegando a valores cercanos a 4800 ms, lo que limita considerablemente su aplicabilidad en simulaciones multirrobot de alta frecuencia.

Otros simuladores como *Isaac Sim* y *Webots* mantienen latencias relativamente bajas y constantes, posicionándose también como opciones viables para simulaciones en tiempo real.

Dado el destacado rendimiento de *O3DE*, se ha decidido explorar y evaluar sus prestaciones en la aplicación de *SLAM 3D*. Por otro lado, debido a las latencias extremadamente altas de *CoppeliaSim*, se ha descartado su uso para esta evaluación.

Para una comparación más detallada y práctica de las latencias, se seleccionarán simuladores que presentan latencias bajas y estables como *Webots* y *Gazebo Ignition*. Esta selección se basa en dos razones principales. Primero, *Isaac Sim* requiere una tarjeta gráfica *Nvidia* para un rendimiento óptimo, un *hardware* que no está disponible en el equipo de trabajo. Segundo, tanto *Webots* como *Gazebo Ignition* cuentan con soporte nativo y consolidado para *ROS2*, lo que facilita su integración y uso en entornos robóticos avanzados, especialmente en aplicaciones que demandan interoperabilidad y modularidad.

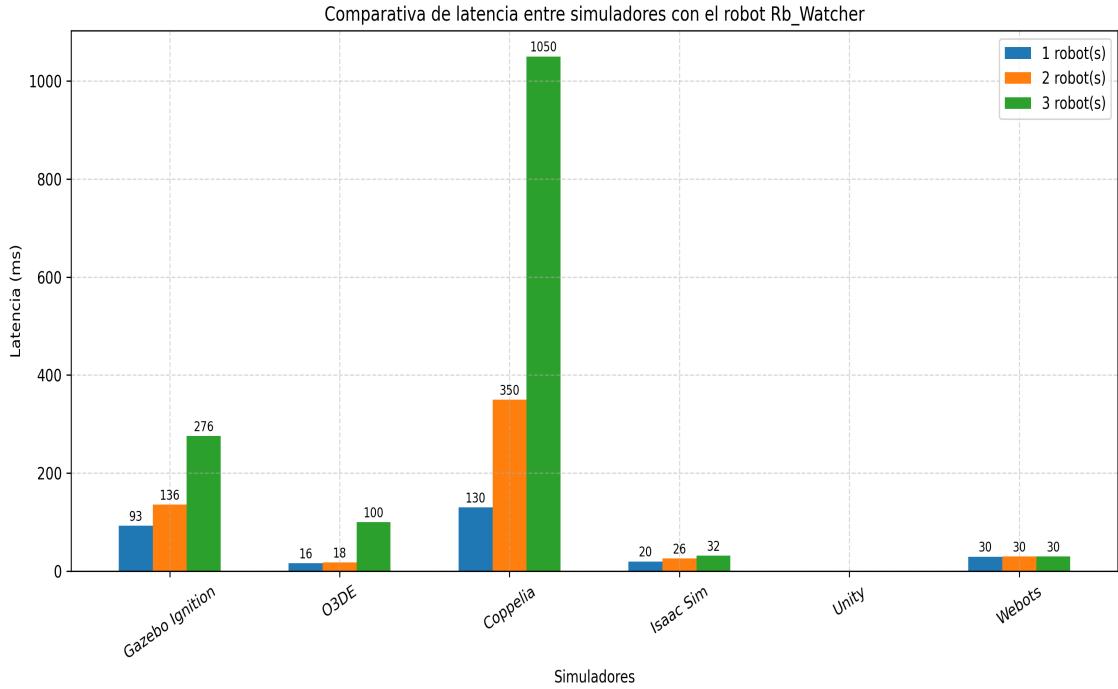


Figura 6.20: Comparativa de la latencia sensorial (en ms) del robot *Rb_Watcher* en distintos simuladores (*Gazebo Ignition*, *O3DE*, *CoppeliaSim*, *Isaac Sim*, *Unity* y *Webots*) con configuraciones de 1, 2 y 3 robots en una escena ligera común.

6.3 Resultados del rendimiento de los paquetes de *SLAM 3D*

En esta sección se presenta un análisis exhaustivo del rendimiento de distintos paquetes de *SLAM 3D* al ser ejecutados en tres simuladores con soporte para entornos robóticos: *O3DE*, *Webots* y *Gazebo Ignition*. Para ello se usó adicionalmente los robots *Rb_Watcher* implementados por otro estudiante en prácticas, Joel Ramos[27]. El objetivo principal del estudio es evaluar la viabilidad de aplicar técnicas de *SLAM 3D* en el simulador *O3DE*, el cual no está diseñado específicamente para aplicaciones robóticas, y comparar su comportamiento con simuladores más consolidados y compatibles con *ROS2*.

Para asegurar una comparación justa y objetiva, se ha definido un escenario de pruebas común y una trayectoria idéntica que el robot recorre en los tres simuladores. Esta trayectoria, representada en la Figura 6.21, permite analizar el desempeño de cada combinación de simulador y paquete *SLAM* bajo las mismas condiciones espaciales, dinámicas y sensoriales. Gracias a esta estandarización, los resultados obtenidos son directamente comparables entre sí.

Durante la ejecución de las pruebas, se han recopilado múltiples métricas con el fin de caracterizar el rendimiento de cada paquete. Entre ellas se incluyen el consumo de *CPU* (en porcentaje) y el uso de memoria *RAM* (en megabytes), indicadores fundamentales de la eficiencia computacional. Además, se ha definido una métrica de compatibilidad con el simulador, cuantificada en una escala de 0 a 4, donde 0 denota una incompatibilidad total y 4 representa una integración completa y funcional. Esta puntuación puede verse penalizada si se requieren herramientas adicionales para establecer comunicación con *ROS2* o para visualizar correctamente los mapas generados.

Otra dimensión relevante del análisis es la calidad del mapeado 3D producido por cada combinación simulador–paquete. Esta se ha evaluado de manera cualitativa mediante una escala ordinal que va desde “*muy baja*” hasta “*muy alta*”, considerando el nivel de detalle, la precisión en la reconstrucción del entorno, la coherencia estructural y la ausencia de errores. La mejor combinación

recibe la calificación de "*muy alta*", mientras que la peor se categoriza como "*muy baja*".

Finalmente, se ha tenido en cuenta el tamaño del mapa generado, medido en *MB*, ya que este parámetro impacta directamente en los requisitos de almacenamiento y en la escalabilidad del sistema al ser implementado en escenarios reales o de gran escala.

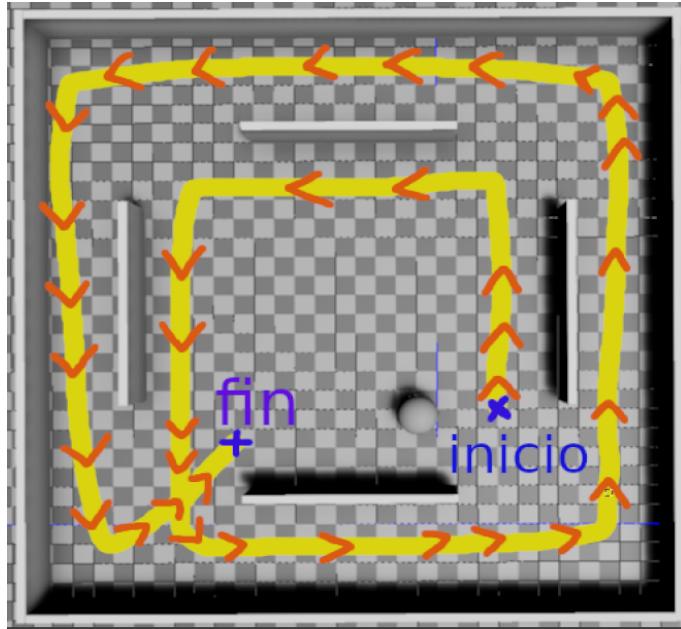


Figura 6.21: Ruta seguida por el robot *Rb_Watcher* mientras se ejecuta cada *SLAM 3D*.

Cuadro 6.5: Resultados de combinaciones simulador–paquete.

Simulador	CPU (%)	RAM (GB)	Compat. (0–4)	Calidad Mapeado	Tamaño (MB)	Imagen
RTAB-MAP						
O3DE	26	1.53	4	Muy alta	25.3	Fig. 6.22
Webots	108.9	2.15	4	Media-Alta	66.9	Fig. 6.24
Gazebo Ignition	59	0.35	1	-	58.4	Fig. 6.23
Lidarslamros2						
O3DE	-	-	0	-	-	-
Webots	230.9	0.15	3	Media	8.1	Fig. 6.28
Gazebo Ignition	431	0.04	4	Media-Baja	5.5	Fig. 6.29
MOLA						
O3DE	19.3	0.32	4	Alta	38.4	Fig. 6.25
Webots	46	0.3	4	Media-Alta	22.7	Fig. 6.26
Gazebo Ignition	59	0.39	3	Media	4.7	Fig. 6.27

El análisis comparativo, basado en la información recogida en la tabla 6.5 , revela comportamientos diferenciados según la combinación de simulador y paquete de *SLAM 3D* utilizado, destacando el caso particular del simulador *O3DE*, cuyo propósito inicial no estaba orientado a la integración directa con sistemas de percepción y control en ROS 2. A pesar de ello, se han obtenido resultados muy destacables que merecen especial atención.

En primer lugar, se ha detectado una incompatibilidad total (nivel 0) entre *O3DE* y el paquete *Lidarslam_ros2*. A pesar de haber replicado las configuraciones empleadas con éxito en los otros simuladores, no ha sido posible ejecutar el paquete en *O3DE*. El principal obstáculo ha sido un error de desincronización de *frames* entre *ROS2* y el simulador, para el cual no se ha encontrado una solución funcional hasta el momento. Esto limita completamente la viabilidad de esta combinación.

Por el contrario, *O3DE* ha mostrado un comportamiento sorprendentemente bueno con el paquete *RTAB-MAP*, el cual ha obtenido la mejor calidad de mapeado 3D entre todas las combinaciones evaluadas. Las reconstrucciones generadas presentan un alto grado de fidelidad geométrica: las

formas de los objetos se conservan con gran precisión, la esfera del entorno se representa prácticamente sin huecos, las paredes detectadas muestran grosor y continuidad, y las zonas escaneadas por el *LIDAR 3D* no presentan vacíos significativos. Estos resultados son especialmente relevantes teniendo en cuenta que *O3DE* no está concebido originalmente para soportar sensores *ROS* nativos, lo cual demuestra que, con la configuración adecuada, puede ofrecer una experiencia de simulación de alta calidad para mapeado 3D.

Además, el paquete *MOLA* ha demostrado una calidad elevada en el mapeado dentro de *O3DE*, aunque ligeramente inferior a la obtenida con *RTAB-MAP*. La reconstrucción del entorno es estable y detallada, lo que refuerza la viabilidad de *O3DE* como simulador para ciertas tareas de *SLAM 3D*, siempre que el paquete empleado sea compatible. Este buen desempeño se correlaciona con la baja latencia observada en los resultados de la subsección anterior, figura 6.20 , similar a lo registrado en *Webots*. En contraste, *Gazebo* presenta mayores latencias y genera mapas cuya calidad no alcanza niveles altos en ninguna de las pruebas realizadas.

Un aspecto a tener en cuenta, sin embargo, es que el controlador del robot comienza a fallar pasados aproximadamente 20 segundos de ejecución en *O3DE*. En particular, se observa un comportamiento anómalo durante maniobras de giro, donde el robot se desplaza de forma muy lenta o no realiza correctamente la rotación sobre su eje. Este efecto puede afectar la navegabilidad o el rendimiento del mapeado en trayectorias prolongadas, por lo que debe considerarse en futuros ajustes o mejoras del entorno.

En el caso de *Gazebo Ignition* con *RTAB-MAP*, si bien se ha logrado realizar una configuración funcional que permite iniciar el proceso de mapeado, el sistema comienza a presentar distorsiones significativas tras unos segundos de ejecución. Se ha observado que, pasado un corto periodo de tiempo, el mapa generado comienza a deformarse debido a errores acumulados, y además aparecen puntos proyectados a distancias infinitas, lo que compromete gravemente la visualización y la utilidad del mapa. Aunque se ha conseguido generar un entorno parcial antes de que se degraden los resultados, estos problemas han reducido su puntuación de compatibilidad hasta un nivel 1, ya que si bien se genera información, esta no es sostenible ni confiable a lo largo del tiempo.

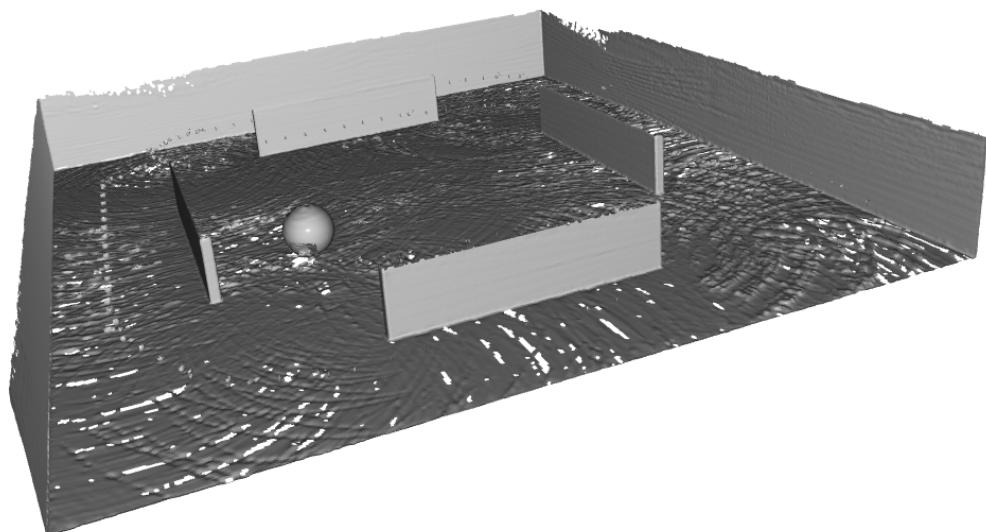


Figura 6.22: Imagen del mapeado 3D realizado con *RTAB-MAP* en *O3DE*.

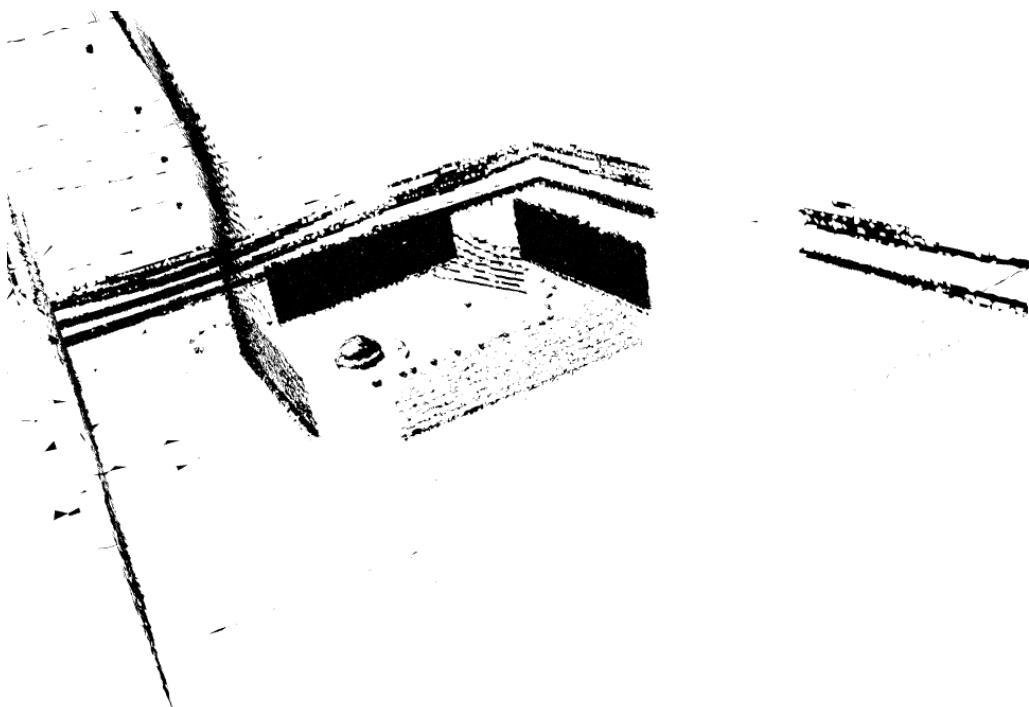


Figura 6.23: Imagen del mapeado 3D realizado con *RTAB-MAP* en *Gazebo*.

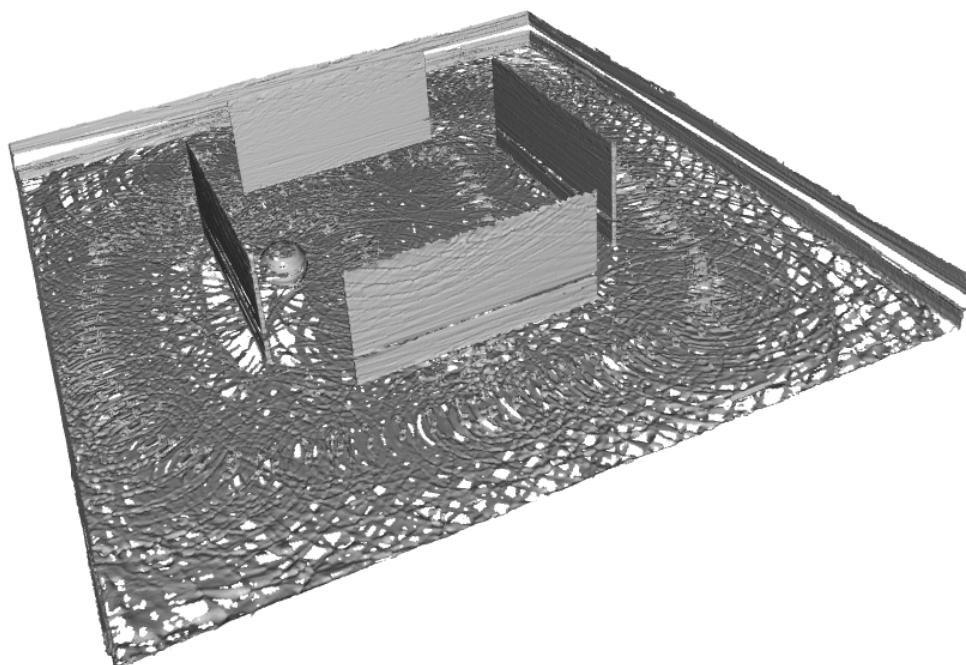


Figura 6.24: Imagen del mapeado 3D realizado con *RTAB-MAP* en *Webots*.

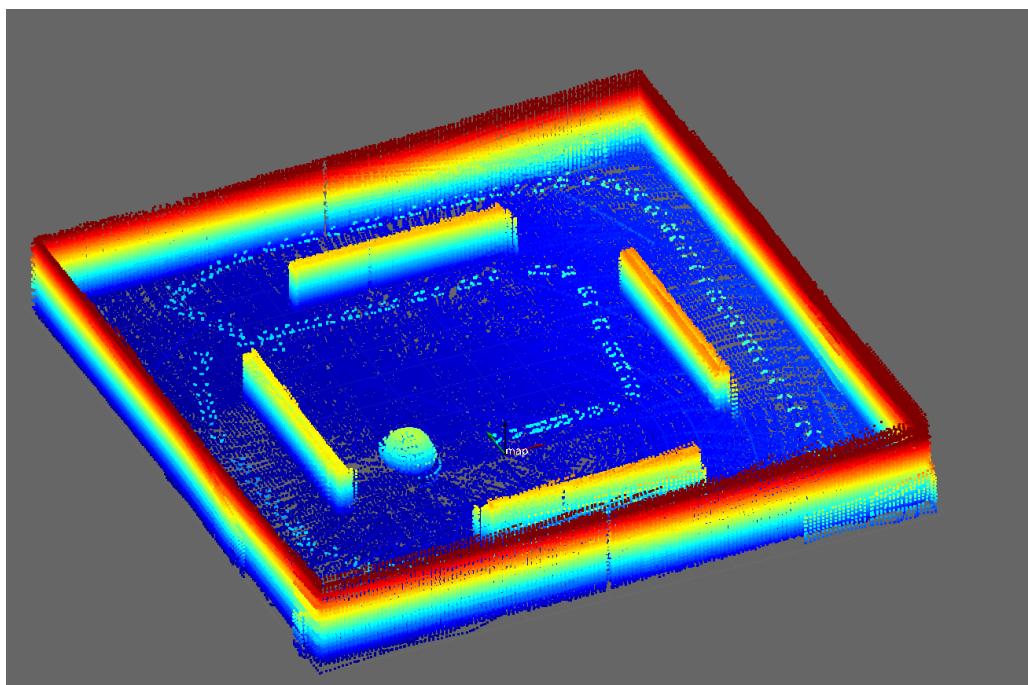


Figura 6.25: Imagen del mapeado 3D realizado con *MOLA* en *O3DE*.

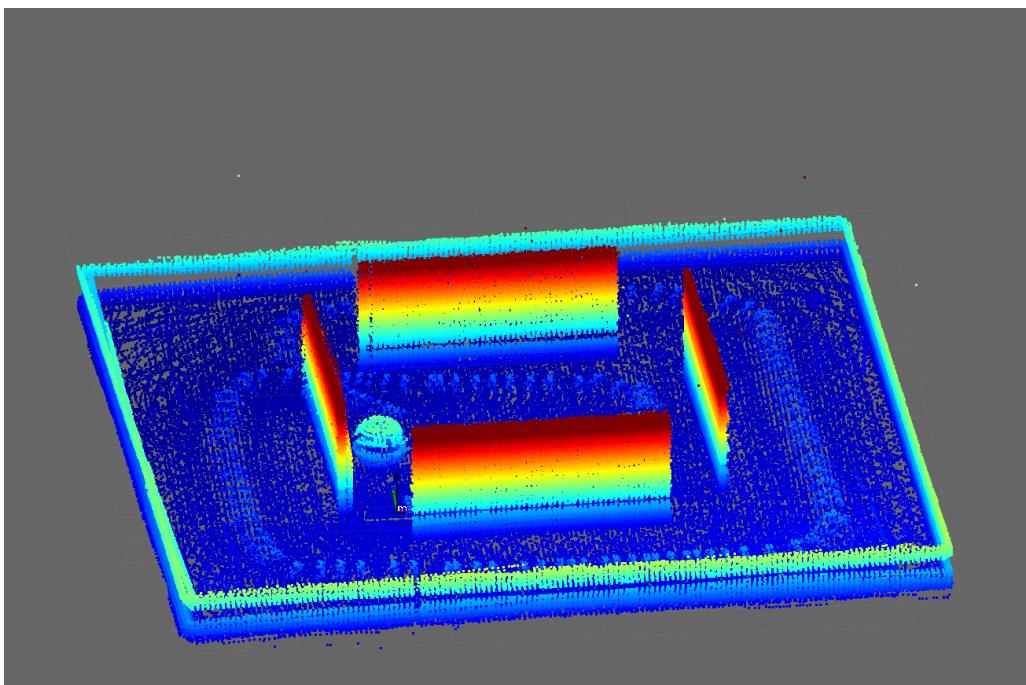


Figura 6.26: Imagen del mapeado 3D realizado con *MOLA* en *Webots*.

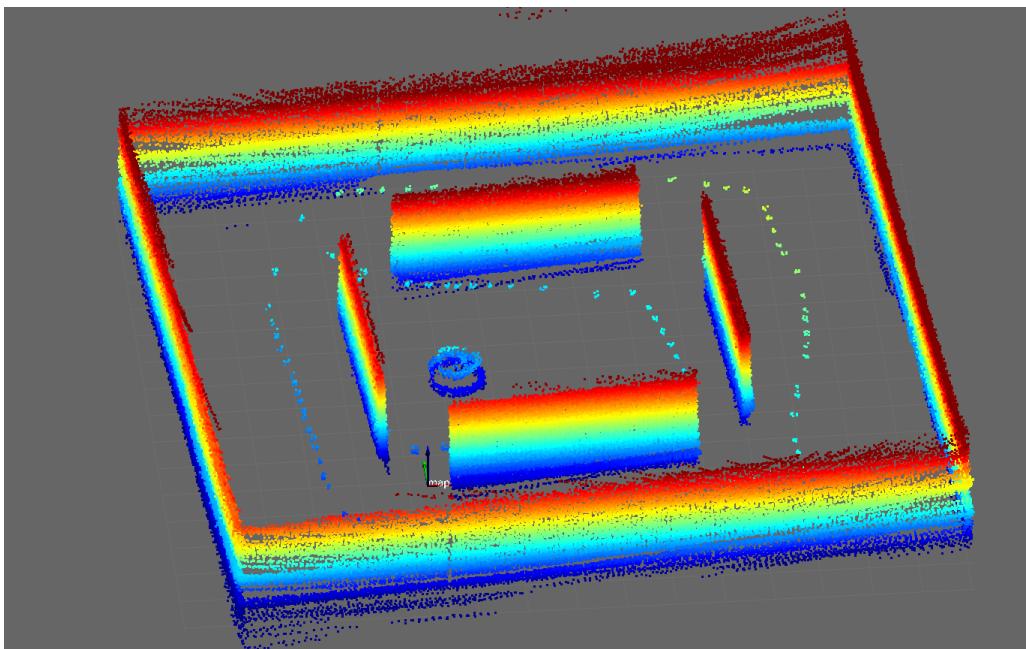


Figura 6.27: Imagen del mapeado 3D realizado con *MOLA* en *Gazebo*.

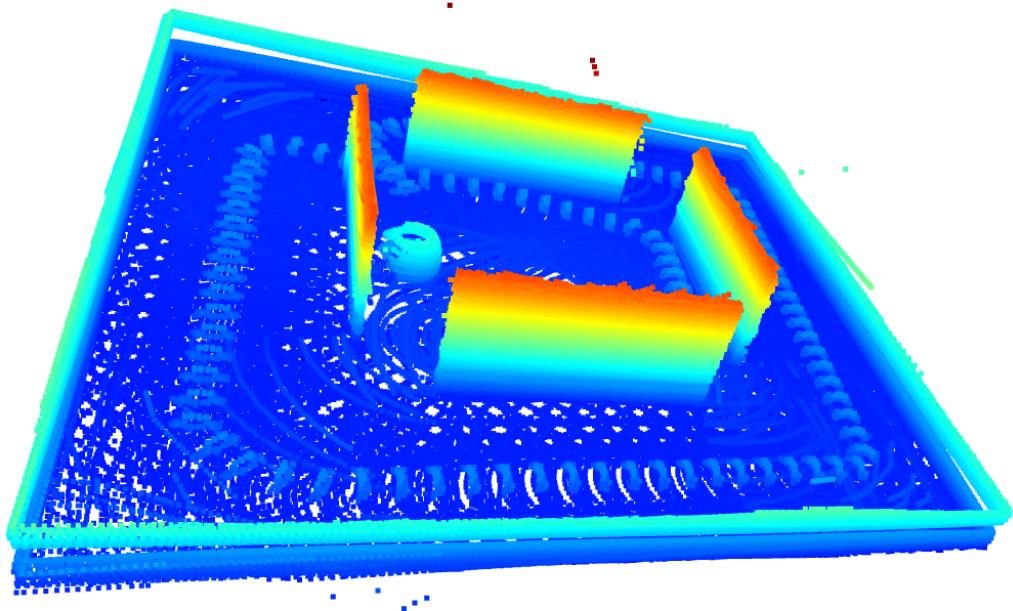


Figura 6.28: Imagen del mapeado 3D realizado con *Lidarslam_ros2* en *Webots*.

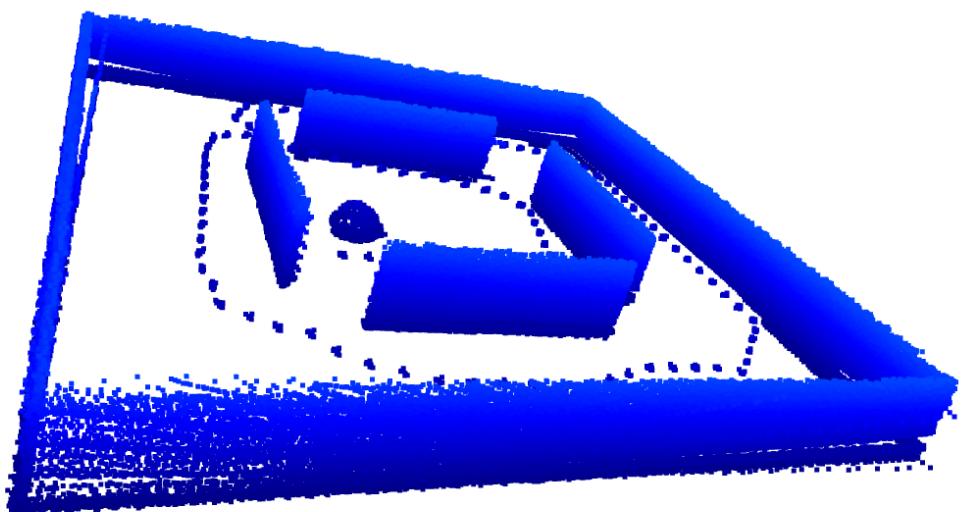


Figura 6.29: Imagen del mapeado 3D realizado con *Lidarslam_ros2* en *Gazebo*.

7. Conclusiones y posible trabajo futuro

En este trabajo se ha conseguido implementar las funcionalidades básicas necesarias para el uso de paquetes de *ROS2* con los robots de *Robotnik*. Se ha demostrado que es posible realizar una correcta integración de dichos robots en dos simuladores diferentes, *CoppeliaSim* y *O3DE*.

Aunque se han presentado ciertas limitaciones, como la estabilidad no óptima del controlador de los robots en *O3DE* y la falta de *plugins* específicos, así como la ausencia de facilidades automatizadas en *CoppeliaSim* (por ejemplo, la comunicación de sensores con *ROS2* ya implementada), el estudio confirma que los robots son funcionales en ambos entornos.

Sin embargo, debido a la arquitectura interna de los simuladores, algunas capacidades resultaron limitadas, particularmente en el manejo multirrobot. En *CoppeliaSim*, se ha demostrado que el número máximo de robots *Rb_Robout* que pueden operar simultáneamente es tres. Además, el robot *Rb_Watcher* no pudo ser utilizado en aplicaciones robóticas con *ROS2* debido a la alta latencia de su sensor *LIDAR 3D*, causada por el *script* de comunicación con *ROS2*. Esta elevada latencia puede estar relacionada con el hecho de que *CoppeliaSim* utiliza únicamente un núcleo de la *CPU* para la simulación y no aprovecha la *GPU*, lo que limita el rendimiento en tiempo real. Esta limitación impidió su uso en tareas de *SLAM 3D*.

Por otro lado, aunque el controlador en *O3DE* presenta cierta inestabilidad, las mediciones de sus sensores y la comunicación con *ROS2* son estables, logrando latencias bajas y una capacidad multirrobot de tres robots *Rb_Robout* y dos *Rb_Watcher*. Además, el uso de un *Rb_Watcher* en este simulador ha producido resultados altamente satisfactorios en el mapeado 3D, incluso mejores que los obtenidos en simuladores más compatibles con *ROS2*, como *Webots* y *Gazebo Ignition*.

En resumen, el trabajo confirma que, pese a algunas restricciones inherentes a cada simulador, la integración y funcionamiento de los robots de *Robotnik* es viable y funcional, ofreciendo un valioso punto de partida para futuras mejoras y aplicaciones en simulación robótica con *ROS2*.

Como continuación natural de este trabajo, se plantean varias líneas de desarrollo orientadas a optimizar el rendimiento de los simuladores, reducir la latencia sensorial y mejorar la calidad general de la simulación en entornos multirrobot.

Una de las principales propuestas se centra en el simulador *CoppeliaSim*, donde se prevé desarrollar un *plugin* en lenguaje *Lua* específico para el sensor 3D. El objetivo de este *plugin* es

que el sensor devuelva directamente los datos en el formato requerido por los mensajes de *ROS2*, evitando así el paso intermedio de transformación y empaquetado que actualmente se realiza en el *script* de comunicación con *ROS2*. La implementación del *plugin* mencionado busca precisamente eliminar este cuello de botella y reducir de forma significativa el tiempo de respuesta sensorial.

Además, se contempla el desarrollo de un *plugin* de control basado en el código fuente del controlador de *ROS1*, con el fin de integrar el sistema de control de forma más directa y eficiente en el simulador. Esto permitiría un mayor grado de sincronización entre el simulador y el *middleware*, y reduciría la carga de procesos externos que actualmente pueden afectar negativamente al rendimiento global del sistema.

En el caso del simulador *O3DE*, se plantea investigar con mayor profundidad cómo está implementado el sistema de control de robots dentro del motor del simulador. Durante los experimentos realizados se detectaron problemas de saturación en el controlador predeterminado al aumentar el número de robots activos, lo que limita la escalabilidad del entorno. Una posible línea de trabajo consiste en estudiar la viabilidad de integrar el controlador desarrollado por *Robotnik*, mediante la creación de un *plugin* personalizado y la recompilación del proyecto con dicha funcionalidad añadida. Esta integración permitiría mejorar el rendimiento del sistema de control y aumentar la capacidad del simulador para escenarios multirrobot exigentes.

Otra mejora relevante consistirá en la automatización del proceso de evaluación mediante el desarrollo de *scripts* en *bash*. Estos permitirán cargar escenas, lanzar robots, medir métricas clave y almacenar los resultados de forma automática. Este procedimiento no solo reducirá la intervención manual requerida, sino que también garantizará la repetibilidad y consistencia de los experimentos, elementos fundamentales en evaluaciones comparativas rigurosas.

En cuanto a la evaluación de los mapas generados mediante algoritmos de *SLAM 3D*, en este trabajo dicha evaluación se ha realizado de forma cualitativa debido a la falta de un formato común entre los distintos resultados. Como parte del trabajo futuro, se plantea convertir todos los mapas generados a un formato unificado, lo que permitirá aplicar métricas cuantitativas basadas en similitud estructural para medir el error entre el entorno real y el reconstruido. Además, se estudiará la aplicación de algoritmos de agrupamiento (*clustering*) con el fin de mejorar la coherencia espacial de los mapas, eliminando puntos aislados o ruido.

Finalmente, una de las tareas previstas en la planificación inicial del trabajo, pero que no llegó a desarrollarse, fue el diseño de entornos industriales simulados. Estos escenarios, que podrían incluir elementos característicos de entornos logísticos o de fabricación, permitirían validar algoritmos de percepción, navegación y control en contextos más cercanos a la robótica aplicada en la industria. Esta línea de trabajo representa una continuación lógica del presente estudio y podría aportar una base valiosa para investigaciones orientadas a la transferencia de soluciones de simulación al ámbito industrial real.

Bibliografía

Bibliografía

- [1] Robotnik Automation Sl - Teléfono y dirección | Empresite. (s. f.). *Empresite España - Buscador de Empresas y Negocios de España*. <https://empresite.economista.es/ROBOTNIK-AUTOMATION.html>
- [2] Ed. (2021, 16 diciembre). *ROS1 vs ROS2, Practical Overview For ROS Developers. The Robotics Back-End.* <https://roboticsbackend.com/ros1-vs-ros2-practical-overview/>
- [3] Open Dynamics Engine. (s.f.). *Open Dynamics Engine*. <https://www.ode.org/>
- [4] Colaboradores de Wikipedia. (2024, 14 noviembre). *Bullet (software)*. Wikipedia, la Enciclopedia Libre. [https://es.wikipedia.org/wiki/Bullet_\(software\)](https://es.wikipedia.org/wiki/Bullet_(software))
- [5] Configuración PhysX de NVIDIA. (s. f.). *NVIDIA*. https://www.nvidia.com/content/control-panel-help/vlatest/es-es/mergedprojects/nv3desn/NVIDIA_PhysX_Configuration.htm
- [6] DART: Dynamic Animation and Robotics Toolkit. (s. f.). *DART*. <https://dartsim.github.io/>
- [7] MasterD. (2023, 10 octubre). *¿Qué es y cómo funciona un motor gráfico?* MasterD. <https://www.masterd.es/blog/que-es-como-funciona-motor-grafico>
- [8] GeeksforGeeks. (2025, 23 abril). *Socket Programming in C*. GeeksforGeeks. <https://www.geeksforgeeks.org/c/socket-programming-cc/>
- [9] Colaboradores de Wikipedia. (2025, 10 mayo). *Middleware*. Wikipedia, la Enciclopedia Libre. <https://es.wikipedia.org/wiki/Middleware>
- [10] Información Ranking de Robotnik Automation sl | Ranking Empresas. (s. f.). *Directorio Ranking Empresas - Ranking de las Principales Empresas Españolas*. <https://ranking-empresas.economista.es/ROBOTNIK-AUTOMATION.html>

- [11] Informe Empleo Informática Julio 2025 - tecnoempleo. (s. f.). <https://www.tecnoempleo.com/informe-empleo-informatica.php>
- [12] Robotnik. (2025, 10 junio). *RB-WATCHER: Robot para seguridad y vigilancia | Robotnik®*. <https://robotnik.eu/es/productos/robots-moviles/rb-watcher/>
- [13] Robotnik. (2025a, febrero 27). *RB-ROBOUT - Robot móvil autónomo industrial | Robotnik®*. <https://robotnik.eu/es/productos/robots-moviles/rb-robout/>
- [14] Robot simulator CoppeliaSim: create, compose, simulate, any robot - Coppelia Robotics. (s. f.). <https://www.coppeliarobotics.com/>
- [15] Documentación ros2 Humble: ROS2 Documentation — ROS2 Documentation: Humble documentation. (s. f.). <https://docs.ros.org/en/humble>
- [16] Automaticaddison. (2021, 18 septiembre). *How to Convert a Xacro File to URDF and Then to SDF*. <https://automaticaddison.com/how-to-convert-a-xacro-file-to-urdf-and-then-to-sdf/>
- [17] sim.getInt32Param. (s. f.). *CoppeliaSim API Documentation*. <https://manual.coppeliarobotics.com/en/regularApi/simGetInt32Param.htm>
- [18] Tablas en Lua | dbtutoriales. (s. f.). *DBTutorial.es*. <https://dbtutoriales.com/tutorial/introduccion-a-la-programacion-en-lua/page/tablas-en-lua>
- [19] Interface between Velodyne VPL16 and ROS - CoppeliaSim forums. (2018, 15 octubre). *CoppeliaSim forums*. <https://forum.coppeliarobotics.com/viewtopic.php?t=7544>
- [20] Davó, A. D. (2024, 4 agosto). *Using ROS2 in Coppelia*. David Davó. <https://blog.ddavo.me/posts/tutorials/ros2-coppelia-lidar/>
- [21] Martínez, B.S., Góngora, J.P., Isaza, L.M. & Jiménez, S.D. (2021). *Control de plataforma móvil de cuatro ruedas mecanum con modelo cinemático*. Universidad EIA. <https://repository.eia.edu.co/bitstreams/3be1ba60-6c69-4d95-b564-ec9c4f1df804>
- [22] O3DE: Installing O3DE for Linux. (s. f.). *O3DE Documentation*. <https://docs.o3de.org/docs/welcome-guide/setup/installing-linux/>
- [23] O3DE: ROS2 gem. (s. f.). *O3DE Documentation*. <https://docs.o3de.org/docs/user-guide/gems/reference/robotics/ros2/>
- [24] Blanco-Claraco, J. L. (2025). A flexible framework for accurate LiDAR odometry, map manipulation, and localization. *The International Journal Of Robotics Research*. <https://doi.org/10.1177/02783649251316881>
- [25] Labb  , M., & Michaud, F. (2019). RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2), 416–446. <https://doi.org/10.1002/rob.21831>
- [26] Rsasaki. (s. f.). GitHub - rsasaki0109/lidarslam_ros2: ROS2 package of 3D lidar slam using ndt/gicp registration and pose-optimization. GitHub. https://github.com/rsasaki0109/lidarslam_ros2?tab=readme-ov-file

- [27] Ramos Beltrán, J. (s. f.). *GitHub - JoelRamosBeltran/Robotnik_practicas: Trabajo que realicé en la empresa de Robotnik durante las prácticas.* GitHub. https://github.com/JoelRamosBeltran/Robotnik_practicas