UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Ingeniería de Sistemas y Automática

Teleoperacion de robots móviles basada en Unity

Trabajo Fin de Máster

Máster Universitario en Automática e Informática Industrial

AUTOR/A: Guerrero de la Casa, Francisco

Tutor/a: Gracia Calandin, Luis Ignacio

Cotutor/a: Solanes Galbis, Juan Ernesto

CURSO ACADÉMICO: 2023/2024

Índice

Índice de figuras

Resumen

Este proyecto de fin de máster se centra en el desarrollo de un sistema integral para el control remoto y la extracción de información de sensores en el robot Turtlebot3, utilizando el entorno ROS (Robot Operating System). La iniciativa surge en colaboración con el Departamento de Ingeniería de Sistemas y Automática y el respaldo del Instituto de Diseño y Fabricación (IDF), que proporciona las herramientas esenciales para llevar a cabo este trabajo.

La primera fase del proyecto se enfoca en la configuración de la conexión en ROS, estableciendo una comunicación efectiva entre el sistema y el Turtlebot3. Esto sienta las bases para el control remoto, donde se implementa la capacidad de manejar el robot de forma remota mediante un teclado y un mando de control, permitiendo movimientos fluidos y seguros.

En paralelo, se aborda la extracción de información de sensores del Turtlebot3. Se identifican y utilizan los diversos sensores disponibles en el robot, desarrollando nodos específicos en ROS para la lectura y procesamiento de esta información. La integración de estos datos en el sistema de control remoto proporciona una comprensión más completa y detallada del entorno circundante del robot.

Opcionalmente, se implementa una interfaz de usuario que facilita la visualización de datos de sensores y estados del robot, mejorando la experiencia de control remoto y permitiendo una supervisión más eficiente.

La metodología adoptada incluye una fase inicial de investigación para comprender las capacidades del Turtlebot3 y el entorno ROS, seguida por el desarrollo de los componentes clave. La integración se realiza cuidadosamente para garantizar una operación sin problemas del sistema. Se documenta de manera exhaustiva el código, la arquitectura y la configuración, y se realizan pruebas rigurosas tanto en entornos simulados como en entornos reales para validar la estabilidad, la precisión del control remoto y la efectividad de la extracción de información de sensores.

Abstract

This master's thesis project focuses on the development of a comprehensive system for remote control and sensor information extraction in the Turtlebot3 robot, using the ROS (Robot Operating System) environment. The initiative arises in collaboration with the Department of Systems and Automation Engineering and the support of the Institute of Design and Manufacturing (IDF), which provides the essential tools to carry out this work.

The first phase of the project focuses on configuring the connection in ROS, establishing effective communication between the system and the Turtlebot3. This lays the foundation for remote control, where the ability to remotely operate the robot using a keyboard and a control pad is implemented, enabling smooth and secure movements.

Simultaneously, the extraction of sensor information from the Turtlebot3 is addressed. The various sensors available on the robot are identified and utilized, with specific ROS nodes developed for the reading and processing of this information. The integration of these data into the remote control system provides a more comprehensive and detailed understanding of the robot's surrounding environment.

Optionally, a user interface is implemented to facilitate the visualization of sensor data and robot states, enhancing the remote control experience and enabling more efficient monitoring.

The adopted methodology includes an initial research phase to understand the capabilities of the Turtlebot3 and the ROS environment, followed by the development of key components. Integration is carefully carried out to ensure a smooth operation of the system. The code, architecture, and configuration are extensively documented, and rigorous testing is conducted in both simulated and real environments to validate stability, remote control accuracy, and the effectiveness of sensor information extraction.

1. Introducción

A medida que la informática y las tecnologías de la nueva industria han evolucionado, con ellas se han desarrollado nuevos métodos para poder realizar las operaciones de antaño de una manera distinta, más eficiente, menos costosa. Cuestiones como el control de estados y situación económica, el crecimiento de las empresas, las gestiones de la producción que antes requerían de complejas operaciones con equipos enteros que fuesen capaces de realizar este análisis. Hoy día estos procesos que a priori pudieran parecer más una traba que una ventaja, no solo se han simplificado y acelerado hasta casi llevar su estudio al día, sino también como esto y su abaratamiento han permitido una democratización del estudio para un gestor de empresas con cualquier cantidad de capital, que sepa buscar y entender cómo usar semejante herramienta, por supuesto.

La tendencia en las empresas es a implementar soluciones de estudio de sus clientes o proveedores. Es así que el análisis de los datos ha llegado a jugar un papel crucial en lo que se refiere a la planificación empresarial. En este proyecto se podrían haber diseccionado diversos campos como pueda ser el performance de los trabajadores, la optimización de los medios de producción y gastos, etc. En este caso se ha decantado por el estudio de la interacción con el medio exterior, es decir, la forma en que los flujos económicos afectan a cada empresa en cuestión y, a ser posible, herramientas de interpretación de tendencias.

Uno de los rasgos fundamentales del estudio de los datos sin importar su naturaleza es el denominador común respecto de la metodología de trabajo:

1. Aunque pareciera obvio, el primer paso para el estudio de datos es en si mismo la obtención de los datos. Y es que los estudios de mercado tienen unos períodos extremadamente largos de años e incluso generaciones de estudio. Sin ir más lejos, los datos de los dos proyectos se remontan hasta el año 2010. Esto puede resultar un hándicap, pues los datos de que se disponen no son numerosos o peor, pueden ser estadísticamente poco explicativos.

2. La transmisión de los datos, aunque en este proyecto no se tratará tal tema, es necesaria junto con un almacenaje en formatos genéricos como hojas de cálculos y bases de datos. En este proyecto todo se hará en formato Excel para mayor simplicidad.

3. La limpieza de los datos es junto con el análisis, el cuerpo y la razón de ser de tales proyectos. El tratamiento de los datos hace posible cualquier modelización o representación que se le desee dar.

4. La analítica de datos corresponde al juicio del ingeniero. No es una mera cuestión de aplicar un modelo predictivo o hacer un panel que enseñe una gráfica, sino crear una historia. Construir una narrativa

que pueda explicar la realidad e intentar que coincida con la creada por un experto en el campo. En definitiva, que el modelo represente una extensión más de la mente del intérprete.

5. La visualización es la meta definitiva del proyecto, pues todo lo que se ha realizado giraba en torno a la construcción de la imagen que justifica la narrativa.

1.1. Motivación del Proyecto

La creciente demanda de soluciones autónomas en entornos diversos ha impulsado el desarrollo de tecnologías robóticas avanzadas. El presente proyecto surge como respuesta a la necesidad de exploración autónoma de espacios, mediante la implementación de un robot móvil con características de control asistido. Este innovador enfoque busca combinar la eficiencia del diseño compacto de un robot doméstico con la capacidad de exploración y mapeo de entornos desconocidos, empleando tecnologías como el LIDAR (Light Detection and Ranging).

El núcleo de la propuesta radica en la integración de un LIDAR en el robot móvil, permitiendo la obtención de datos precisos sobre el entorno circundante. Este sensor, junto con el empleo de la plataforma de comunicación ROS TCP-IP, posibilita la transmisión en tiempo real de información detallada sobre el posicionamiento del robot y datos de obstáculos. La interacción remota se convierte así en una herramienta esencial para la supervisión y control eficaz del robot, proporcionando una experiencia de usuario fluida desde cualquier dispositivo remoto. Este proyecto también se centra en la utilidad práctica de la información recopilada con el mapeo en tiempo real de los datos del LIDAR. Este enfoque combinado de exploración y presentación de datos sienta las bases para aplicaciones prácticas en diversos campos, desde la vigilancia autónoma hasta la cartografía de interiores en entornos complejos.

Con este proyecto, se pretende avanzar en la frontera de la robótica móvil autónoma, proporcionando una solución integral que no solo demuestre la viabilidad técnica, sino que también abra nuevas posibilidades en la exploración autónoma y la representación visual de entornos en tiempo real.

1.2. Objetivos del Proyecto

Los objetivos del siguiente proyecto son el análisis de la viabilidad del empleo del control manual para el mapeo y detección de entornos con las herramientas actuales en Unity y ROS junto con los recursos ya provistos por el fabricante del Turtlebot3:

- Producir una herramienta viable para un usuario no familiarizado. Control simple, dinámico e intuitivo.
- Familiarización con las herramientas provistas por el entorno de programación y motor gráfico junto con la comunicación por ROS con sus paquetes.
- Crear un producto visual y explicativo que permita al usuario poco cualificado construir una historia entorno a lo que se está visualizando.
- Añadir herramientas para optimizar y exportar el modelo con el fin de que puedan ser empleadas por futuros programadores.
- Propuestas a futuro. Sugerencias de líneas de investigación nuevas y propuestas de mejora del control y resultados.

1.3. Estructura del Documento

Este proyecto será definido a lo largo de las siguientes páginas y posteriormente se exhibirán los resultados y las conclusiones, se expone cuál es la estructura de capítulos del documento y una breve explicación de cada apartado:

1. Introducción: En este apartado se hace una breve introducción de lo que se desea por parte del proyectista y la empresa directora, de los objetivos de los proyectos y la motivación y justificación del trabajo.
2. Antecedentes: Este apartado se divide en puntos clave de los conceptos tratados en la documentación y se exponen ciertos proyectos en los que se presentó lo último de dichos puntos a tratar, hasta concluir en ciertos razonamientos previos definidos por estudios, de manera que puedan servir de guía para el desarrollo del trabajo.
3. Tecnologías empleadas: Los programas, paquetes y motores empleados para el grueso del desarrollo del proyecto.
4. Soluciones alternativas: Un breve comentario de alternativas y líneas paralelas que podrían desarrollarse a raíz del proyecto.
5. Programación: En el apartado se planteó como un listado y definición de los lenguajes y/o programas empleados, aunque sea una lista corta, junto con las librerías y paquetes con los que se programó.
6. Desarrollo del proyecto: Contiene el corpus del proyecto y como tal consta de apartados importantes tales como:

   a) Instalación de todos los programas (sin contar ciertos paquetes de uso puntual).

   b) ROS-Unity Integration para la comunicación entre el robot y el simulador Unity.

   c) Algunas demos de comprobación.

   d) Una propuesta de indicadores de movimiento para el usuario.

   e) La lectura del Lidar del robot para el descubrimiento del mapa.

   f) Los resultados finales listos para extraer conclusiones y propuestas.

7. Conclusiones: Las conclusiones extraidas de los resultados del desarrollo, junto con alguna justificación de la elección y decisiones.
8. Propuestas a Futuro: Varias propuestas de mejora en diferentes rangos de complejidad y utilidad.

2. Antecedentes

2.1. Robótica

La cronología de la robótica móvil, desde los albores de los años 50 hasta los desarrollos más recientes, revela una trayectoria fascinante en la que la creatividad humana ha dado vida a máquinas programables y versátiles. George Devol, un pionero en la materia, no solo concibió esta idea, sino que también logró un hito clave al vender un modelo de soldadura a General Motors, estableciendo así los cimientos para la incursión de robots en aplicaciones industriales. [Fra21]Robots Modernos

Un momento decisivo en la historia de la robótica móvil fue la creación del robot Shakey durante la década de los 60. Aunque sus capacidades eran limitadas en términos de planificación y actuación ante el entorno, representó uno de los primeros casos de estudio de robots dirigidos de manera remota. Esta etapa inicial allanó el camino para posteriores desarrollos.

La planificación de rutas, esencial en la navegación robótica, experimentó un notable avance gracias al impulso de la tercera revolución industrial. Este periodo facilitó el desarrollo de estrategias tanto de bajo como de alto nivel en planificación, abordando la reacción ante eventos externos e internos, así como la definición de rutas predefinidas. La distinción entre ruta global, donde se posee toda la información del entorno, y camino local, con información limitada al inicio, ha sido vital para optimizar la eficiencia en la navegación en entornos complejos.

En las últimas décadas, los robots móviles se han utilizado con éxito en diversas áreas, como la militar, la industrial y la seguridad, para llevar a cabo misiones cruciales sin necesidad de tripulación. La planificación de rutas es un problema fundamental que debe resolverse antes de que los robots móviles puedan navegar y explorar entornos complejos. Se divide en planificación de rutas globales (donde el robot conoce toda la información del entorno antes de comenzar) y planificación de rutas locales (donde el robot comienza con información limitada sobre el entorno). La planificación de rutas es esencial para ahorrar tiempo y reducir costos. Este artículo presenta una revisión crítica de estas dos categorías de planificación de rutas y resume investigaciones relevantes en el campo. [cita]

El ámbito de los robots móviles omnidireccionales ha experimentado una investigación profunda en cuanto a mecanismos de ruedas especializados.

[yZmLxC18]Han-ye Zhang, Wei-ming Lin y Ai-xia Chen llevaron a cabo una revisión de diversos mecanismos de ruedas especializados, centrándose principalmente en la construcción de robots móviles holonómicos y omnidireccionales. Luego, el enfoque se dirige a un robot móvil omnidireccional específico, cuya plataforma está compuesta por tres conjuntos de ruedas laterales ortogonales. Tras introducir un modelo dinámico para este tipo de robot móvil, se exploran cuatro enfoques de control con el objetivo de desarrollar un sistema de control basado en retroalimentación. [cita]

[Wat98]Watanabe remarcó la importancia de la investigación y desarrollo en el campo de los robots móviles omnidireccionales, los cuales ofrecen capacidades excepcionales de movilidad y maniobrabilidad en una variedad de entornos. Los enfoques de control discutidos representan una contribución valiosa para mejorar el rendimiento y la precisión de estos robots en diversas aplicaciones, consolidando así el interés continuo en este emocionante y en constante evolución campo de la robótica móvil. [cita]

En una vertiente innovadora, se ha explorado la introducción de comportamiento caótico en robots móviles. Un controlador diseñado para garantizar movimientos caóticos ha demostrado eficacia al permitir una exploración completa del espacio de trabajo sin necesidad de un mapa previo o planificación de ruta.

Un robot móvil con estas características podría ser aplicado como un robot de patrulla o limpieza en espacios cerrados, como habitaciones, pisos o edificios. La dependencia sensible de las condiciones iniciales se convierte en una ventaja en el caso de un robot de patrulla, ya que su trayectoria de exploración se torna altamente impredecible.

[NS01]Nakamura y Sekiguchi comparan el comportamiento caótico con el método de paseo aleatorio para explorar un espacio de trabajo desconocido. Se llevó a cabo un experimento con un robot que realiza un paseo aleatorio en el mismo entorno. Se observa que el robot caótico puede explorar de manera más eficiente el espacio de trabajo, ya que no necesita detenerse y girar, como en el paseo aleatorio. La densidad de cobertura del robot en su entorno es uniforme, lo que garantiza una exploración eficaz.

Los resultados muestran que el robot móvil caótico puede cubrir el 90 % del área total en una tercera parte del tiempo requerido por el paseo aleatorio para lograr la misma cobertura. Se sugiere que el enfoque caótico podría ser estocásticamente superior al método aleatorio, lo que lo convierte en una estrategia prometedora para la exploración en robótica en entornos desconocidos.

En síntesis, la robótica móvil ha evolucionado desde sus inicios hasta convertirse en un campo diverso y vibrante, donde la planificación de rutas, el desarrollo de robots omnidireccionales y la introducción de comportamientos caóticos continúan impulsando la innovación y el progreso. Estos avances representan solo una fracción de las posibilidades que el futuro promete en la intersección entre la tecnología y la movilidad robótica.

2.2. ROS

[CDFS+14]Codd-Downey, Mojiri, Speers, Wang y Jenkin hablaron de como la construcción de interfaces de teleoperación basadas en realidad virtual para sistemas autónomos implica la integración de dos tecnologías de software muy diferentes. Los robots habilitados para ROS utilizan una arquitectura de software basada en el paso de mensajes, mientras que los sistemas de realidad virtual están típicamente vinculados a un bucle de representa-

ción. Este artículo describe un sistema y un enfoque general para vincular estas dos tecnologías dispares. En lugar de desarrollar un nodo de realidad virtual dentro de ROS, se utilizan rosbridge y websockets para proporcionar un mecanismo mediante el cual la infraestructura de realidad virtual puede escuchar e inyectar mensajes de ROS en el entorno de ROS.

El uso de websockets como protocolo de comunicación subyacente permite que los entornos de ROS y realidad virtual operen en hardware completamente separado, solo requiriendo que los dos sistemas estén conectados por una conexión de red confiable. Los experimentos informados aquí se realizaron en interiores y utilizando sistemas de visualización relativamente simples. El trabajo en curso está investigando la implementación del robot en exteriores y el uso de una plataforma de movimiento para indicar la orientación de balanceo/inclinación del vehículo con respecto a la gravedad a un operador que controla el robot desde la plataforma de movimiento.

Este trabajo aborda la integración de tecnologías de manera novedosa, superando las diferencias fundamentales en la arquitectura de software de ROS y los sistemas de realidad virtual. El enfoque de utilizar websockets para la comunicación facilita la operación en hardware separado, lo que podría tener aplicaciones más amplias en entornos de teleoperación para robots autónomos en diversos escenarios.

En este otro caso de uso específico, [LNS+20] se exponía que las acciones de cada robot son determinadas por su propio programa en lugar de un algoritmo de control general. Esto significa que la información de transformación de los robots no se comparte completamente entre todos los robots, y cada robot solo puede detectar el objetivo frente a él basándose en sus propios sensores.

En muchos casos del mundo real, los robots son controlados por un sistema central. Para verificar el rendimiento de esta implementación para este sistema central, presentamos un segundo caso de uso en la siguiente subsección.

Si bien este proyecto es simple y solo consistirá en la teleoperación de un robot con el objetivo de al menos emplear la función Lidar, por lo menos en interesante mostrar el alcance de la comunicación y control mediante ROS. Y es que el artículo describe un caso de uso específico de simulación híbrida multirobot, utilizando ocho robots Turtlebot3, dos de los cuales son físicos y seis virtuales. El objetivo principal fue evaluar el rendimiento de esta simulación en un escenario de tráfico complejo.

El proyecto en cuestión fue el del Comportamiento de Entrada y Salida de Multirobot en una Glorieta [cita] y consistió en la simulación híbrida multirobot que implicó la combinación de robots físicos y virtuales en un entorno simulado. Este enfoque permite evaluar y validar algoritmos de control y comportamientos en escenarios realistas sin el riesgo asociado con robots físicos.

Turtlebot3 es una plataforma de robot móvil muy popular y por lo tanto es común encontrarla empleada en la investigación y desarrollo. Su versatilidad

y capacidad para integrar diferentes sensores y actuadores tales como giroscopios, Lidar y acelerómetros lo hacen adecuado para simulaciones realistas de entorno y mapas.

Se utilizaron dos algoritmos para controlar la velocidad de los robots, uno para seguir el carril en la glorieta utilizando un controlador PID, y otro para ajustar la velocidad y evitar colisiones mediante el uso de sensores simulados de LIDAR.

2.3. Unity Game Engine

Unity Technologies lanzó la primera versión de Unity a mediados de 2005, inicialmente enfocada en el desarrollo para OS X. Desde entonces, Unity ha evolucionado y ampliado su soporte a más de 25 plataformas, incluyendo realidad virtual y aumentada. Unity es un motor de juego independiente de la plataforma, donde los juegos se crean manipulando objetos 2D o 3D y adjuntándoles varios componentes. Posee un potente motor de física PhysX, motor de renderización, motor de detección de colisiones, entre otros. Además, admite entornos 3D de alta fidelidad y permite modelar sensores. Utiliza un mesh estándar para los objetos del juego, lo que incluye la ubicación de todos los vértices de la forma del objeto, mejorando así la autenticidad y realismo del comportamiento de los objetos dinámicos en el simulador. [LNS+20]

La comunidad de Unity cuenta con una amplia variedad de tutoriales en línea, lo que facilita el inicio del desarrollo desde el primer día. Unity tiene una API bien documentada junto con una comunidad activa. Permite modificar el entorno virtual mediante un editor o manipularlo directamente en la ventana de juego. También proporciona un lenguaje de secuencias de comandos para que los desarrolladores definan comportamientos y controladores.

Comparado con otros motores de juego, Unity se destaca por ser un editor rico en funciones y altamente flexible. Ofrece soporte tanto para desarrollo 2D como 3D, con herramientas y funcionalidades necesarias para diferentes géneros. Además, cuenta con herramientas avanzadas de inteligencia artificial, como un sistema de navegación que permite crear personajes no jugadores (NPC) capaces de moverse inteligentemente en el entorno virtual. La iteración rápida es otra característica destacada, ya que el modo de reproducción se utiliza para realizar ediciones iterativas rápidas y ver las alteraciones en los scripts instantáneamente. Por último, Unity cuenta con una comunidad vibrante de desarrolladores que comparten ideas, sugerencias, información y consultas.

2.3.1. Teleoperación con Unity

[HGOM18] Antiguo sistema de conexión. Biblioteca de enlace ROS a través de rosbridge expuesto por Hussein, García y Olaverri-Monreal en la ICVES.

Es necesario utilizar bibliotecas para vincular las dos arquitecturas, ROS y Unity, que deben implementarse en ambos extremos. Tiene que haber una estructura para ROS1 específicamente con todos los nodos que se podrían tener en el sistema, además de un nodo de Unity que actúa como el enlace para suscribirse y publicar mensajes a través del nodo de rosbridge de y hacia Unity. Por otro lado, en el lado derecho se encuentra el motor de juego Unity con todos los scripts que uno podría tener en el sistema, además de un script ROS que actúa como el enlace para suscribirse y publicar mensajes a través de la biblioteca rosbridgelib de y hacia ROS. Todos los mensajes transmitidos entre las dos arquitecturas están formateados como JSON.

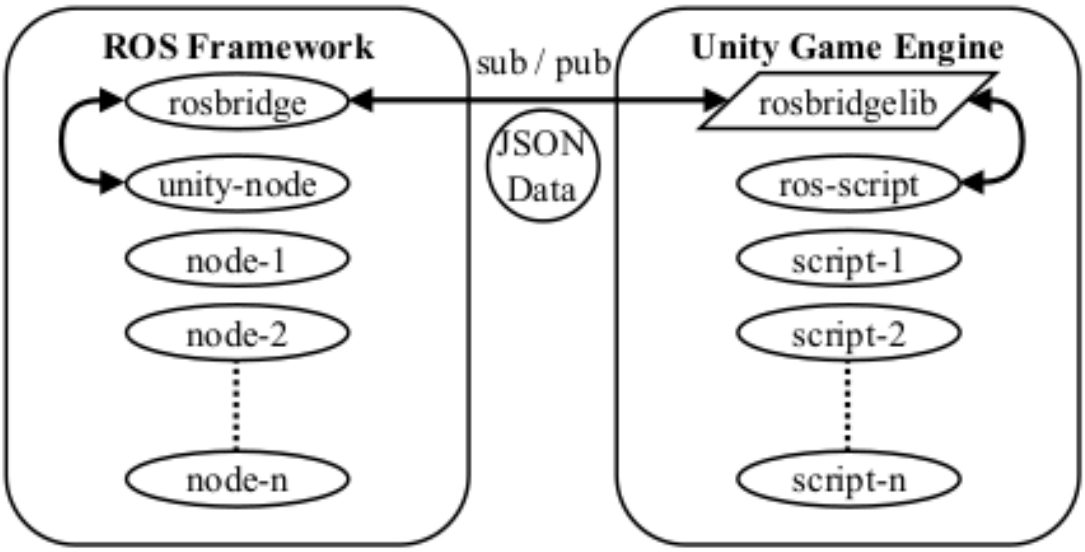


Figura 1: Vista general de la conexión entre ROS y Unity.

Para la estructura ROS: se podría utilizar la biblioteca rosbridge que, si bien es más fácil de emplear, está cayendo en desuso por su falta de versatilidad, con lo que en este proyecto se montará todo a partir de los sockets tradicionales de ROS. Esta biblioteca proporciona una API JSON para la funcionalidad de ROS para programas no ROS. Es una biblioteca de Python responsable de tomar cadenas JSON y convertirlas en mensajes ROS, y viceversa. En este enfoque, se seleccionó un servidor WebSocket como interfaz, donde el servidor de rosbridge acepta conexiones WebSocket e implementa el protocolo de rosbridge. [...] se inicia el nodo websocket de rosbridge desde el paquete del servidor de rosbridge, después de ajustar la dirección IP y el número de puerto del WebSocket. Luego, en el nodo de Unity, se suscribe a los mensajes desde Unity y publica mensajes a Unity según las necesidades de la aplicación.

Para el motor Unity: se utiliza en el ejemplo la biblioteca rosbridgelib. Esta biblioteca debe incluirse en la carpeta de activos del proyecto Unity, donde

debe importarse en el archivo de script ROS. Este es el archivo principal que configura la conexión WebSocket para conectarse a la misma dirección IP y el puerto ajustados por el nodo WebSocket de rosbridge. [...] La biblioteca no admite todos los mensajes de ROS; actualmente, los disponibles son aquellos de los paquetes estándar y personalizados definidos en las dependencias del paquete.

3. Tecnologías empleadas

Para definir correctamente las tecnologías producto del desarrollo de las empresas previamente citadas no hay mejor explicación que la de dichas empresas involucradas en su mismo desarrollo.

3.1. Ubuntu/Linux

Según Canonical, la empresa que lo ideó inicialmente y la propia Ubuntu Comunity, Debian es un proyecto voluntario que ha desarrollado y mantenido un sistema operativo GNU/Linux durante más de una década. Y es que se trata de la roca sobre la que se forjó el sistema operativo Unix del que parte Ubuntu.

Ubuntu desarrolla y mantiene un sistema operativo de código abierto multiplataforma basado en Debian, con un enfoque en la calidad de lanzamiento, actualizaciones de seguridad empresariales y liderazgo en capacidades clave de plataforma para integración, seguridad y usabilidad. [...] Canonical garantiza el mantenimiento y soporte empresarial con la opción de obtener mantenimiento de Seguridad y soporte comercial para implementaciones de Ubuntu en el escritorio, el servidor y la nube.

Canonical lidera el ecosistema de Ubuntu, colaborando con proveedores de servicios en la nube y hardware público para habilitar una plataforma de alta calidad que puede utilizarse de forma gratuita en cualquier lugar.

3.2. Unity

Unity es un motor de juego multiplataforma desarrollado por Unity Technologies. Inicialmente diseñado para Mac OS X, ha evolucionado para admitir diversas plataformas, desde escritorio hasta realidad virtual. Destaca en el desarrollo de juegos móviles para iOS y Android, siendo accesible para desarrolladores principiantes e ideal para proyectos independientes. Además de su uso en videojuegos, Unity ha encontrado aplicación en industrias como el cine, automotriz, arquitectura, ingeniería, construcción y las Fuerzas Armadas de los Estados Unidos.

Unity se trata de una plataforma mundial para crear y operar contenido interactivo en 3D en tiempo real (RT3D), capacitando a creadores en diversas industrias e industrias de diversa naturaleza y alrededor del mundo.

3.3. ROS

Robot Operating System (Sistema Operativo de Robots), es un marco de trabajo de código abierto diseñado para el desarrollo de software para robots. En realidad no se trata de un sistema operativo, sino más bien un conjunto de paquetes y bibliotecas que facilitan el desarrollo de software para robots (mayormente facilitados para Unix, aunque tambien funcional para otros sistemas operativos).

1. Se proporciona una arquitectura distribuida que permite la comunicación entre diferentes componentes de software llamados nodos a través de un núcleo o core. Estos nodos pueden ejecutarse en diferentes computadoras y dispositivos y comunicarse entre sí a través de un sistema de mensajería IP.

2. Controladores y abstracciones para varios dispositivos y hardware comúnmente utilizados en robótica, facilitando la interoperabilidad y reutilización de código como es el caso del mando se Xbox o el robot Turtlebot3.

3. El software en ROS se organiza en paquetes, que contienen bibliotecas, ejecutables, conjuntos de datos y archivos de configuración. Esto simplifica la gestión de dependencias y la distribución de software. Por ejemplo, la simulación del entorno de trabajo del propio Turtlebot3.

4. Este entorno incluye herramientas de línea de comandos y visualización y manipulación de datos en tiempo real tales como el posicionamiento, la rotación de los ejes o valores del lidar del Turtlebot3.

ROS se utiliza en una variedad de aplicaciones robóticas, desde robots industriales y de servicio hasta drones y vehículos autónomos. Proporciona una base sólida para el desarrollo de software robótico al ofrecer una infraestructura común y herramientas para desarrolladores de robótica en todo el mundo.

La versión empleada para este proyecto en específico será una antigua ROS melodic que encaja en el Ubuntu 18.04.

3.3.1. Unity/ROS-TCP-Connector

Se trata de un repositorio generado por el propio equipo de ROS con el apoyo del equipo Siemens ROS# Project Team para su empleo en Unity.

Este repositorio contiene dos paquetes de Unity: el ROS-TCP-Connector, para enviar y recibir mensajes desde ROS, y el Visualizations Package, para agregar visualizaciones de mensajes entrantes y salientes en la Unity scene.

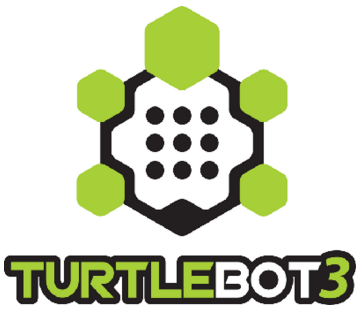
3.3.2. Unity/ROS-TCP-Endpoint

Paquete ROS Unity utilizado para crear un endpoint que acepta mensajes ROS enviados desde una escena de Unity. Este paquete ROS funciona en conjunto con el paquete Unity ROS-TCP-Connector.

Las instrucciones y ejemplos sobre cómo utilizar este paquete ROS se pueden encontrar en el repositorio Unity Robotics Hub.

3.3.3. Turtlebot3 + Burger

TurtleBot es un robot de plataforma estándar para ROS. El nombre Turtle proviene del robot Turtle, impulsado por el lenguaje de programación educativo Logo en 1967. Además, el nodo turtlesim, que aparece por primera vez en el tutorial básico de ROS, es un programa que imita el sistema de comandos del programa de la tortuga Logo. También se utiliza para crear el ícono de Turtle como símbolo de ROS. Los nueve puntos utilizados en el logo de ROS provienen del caparazón trasero de la tortuga. TurtleBot, que se originó a partir de Turtle de Logo, está diseñado para enseñar fácilmente a personas nuevas en ROS a través de TurtleBot, así como para enseñar el lenguaje de programación Logo. Desde entonces, TurtleBot se ha convertido en la plataforma estándar de ROS, siendo la plataforma más popular entre desarrolladores y estudiantes.



La tecnología central de TurtleBot3 es SLAM, navegación y manipulación, lo que lo hace adecuado para robots de servicio doméstico. TurtleBot puede ejecutar algoritmos SLAM (localización y mapeo simultáneos) para

construir un mapa y moverse por una habitación. Además, se puede controlar de forma remota desde una computadora portátil, un joystick o un teléfono inteligente basado en Android (o un mando de Xbox, siendo este el caso del proyecto). [...] TurtleBot3 se puede utilizar como un manipulador móvil capaz de manipular un objeto mediante la conexión de un manipulador como OpenMANIPULATOR. OpenMANIPULATOR tiene la ventaja de ser compatible con TurtleBot3 Waffle y Waffle Pi, lo que permite compensar la falta de libertad y lograr una mayor integridad como robot de servicio con las capacidades de SLAM y navegación que tiene TurtleBot3.

3.3.4. Gazebo

Gazebo es una colección de bibliotecas de software de código abierto diseñadas para simplificar el desarrollo de aplicaciones de alto rendimiento. La audiencia principal de Gazebo son los desarrolladores, diseñadores y educadores de robots. [...] Cada biblioteca dentro de Gazebo tiene dependencias mínimas, lo que permite su uso en tareas que van desde la resolución de transformaciones matemáticas hasta la codificación de video y la simulación, así como la gestión de procesos. Simplemente elige las bibliotecas que necesitas para tu aplicación sin comprometerte con todo un ecosistema.

La confianza y la fiabilidad se han establecido a través de un proceso de curación y mantenimiento liderado por Open Robotics en colaboración con una comunidad de desarrolladores. Cada biblioteca dentro de Gazebo ha sido diseñada para cumplir un propósito específico. [...] El desarrollo y el mantenimiento siguen un protocolo que incluye revisiones múltiples, verificadores de código e integración continua. [cita]

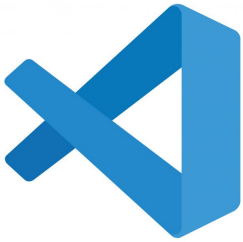
3.4. Visual Studio

IDE de Visual Studio es una plataforma de lanzamiento creativa que puede utilizar para editar, depurar y compilar código y, finalmente, publicar una aplicación. Además del editor y depurador estándar que ofrecen la mayoría de IDE, Visual Studio incluye compiladores, herramientas de completado de código, diseñadores gráficos y muchas más funciones para mejorar el proceso de desarrollo de software. [cita]

Para el caso de este proyecto bastó con Visual Studio únicamente como plataforma de codificación, pues Unity cuenta con su propia compilación cada vez que se crea un ejecutable o se prueba un escenario, poniendo a punto cada uno de los objetos del proyecto.

3.4.1. Visual Studio Code

Visual Studio Code es un editor de código fuente ligero pero eficaz que se ejecuta en el escritorio y está disponible para Windows, macOS y Linux (siendo essta última la preferencia de sistema operativo del proyecto por su facilidad de manejo del ROS). Incluye compatibilidad integrada con JavaScript, TypeScript y Node.js, y cuenta con un amplio ecosistema de extensiones para otros lenguajes y entorno de ejecución. [cita]



4. Programación

Los lenguajes, paquetes y librerías empleados en el desarrollo del proyecto.

4.1. CSharp/C#

Se trata de un lenguaje de programación desarrollado por Microsoft que forma parte de la familia de lenguajes de programación C y se diseñó con el objetivo de combinar la eficiencia y la potencia de lenguajes como C++ con la facilidad de uso y la productividad de lenguajes como Java. La principal ventaja de este lenguaje es la capacidad de mantener procesos paralelos con facilidad y eficiencia.

1. Se trata de un lenguaje de programación orientado a objetos, lo que significa que se organiza alrededor de objetos que pueden contener datos y código que opera en esos datos.

2. Su tipado estático, aunque menos cómodo por hacer que los tipos de datos deben declararse antes de su uso, hace que se verifique su corrección en tiempo de compilación para asegurarse de que se utilicen de acuerdo con las reglas establecidas por el lenguaje.

3. A diferencia de lenguajes como C++, C# tiene un sistema de recolección de basura que gestiona automáticamente la liberación de memoria, simplificando el desarrollo y reduciendo los errores relacionados con la gestión manual de la memoria.

4. Incluye características de seguridad, como la verificación de límites de los arrays y la gestión automática de la memoria, que ayudan a prevenir errores comunes que podrían conducir a vulnerabilidades de seguridad. Aunque tal funcionalidad no será de utilidad para una aplicación tan simple.

5. Fué diseñado para facilitar el desarrollo rápido de aplicaciones mediante el uso de bibliotecas y herramientas integradas en el entorno de desarrollo de Microsoft, como el Visual Studio para este proyecto.

Con una gran variedad de aplicaciones, desde desarrollo de software para Windows hasta desarrollo de aplicaciones web y servicios en la nube a través de la plataforma .NET. También es un lenguaje de elección para el desarrollo de juegos con el motor Unity. Para el caso de este proyecto será precisamente su capacidad de aprovechar los objetos preparados por los paquetes ROS para la manipulación de los telegramas producidos por la simulación y posteriormente el mismo Turtlebot3.

4.2. Paquete ROS–Unity Communication

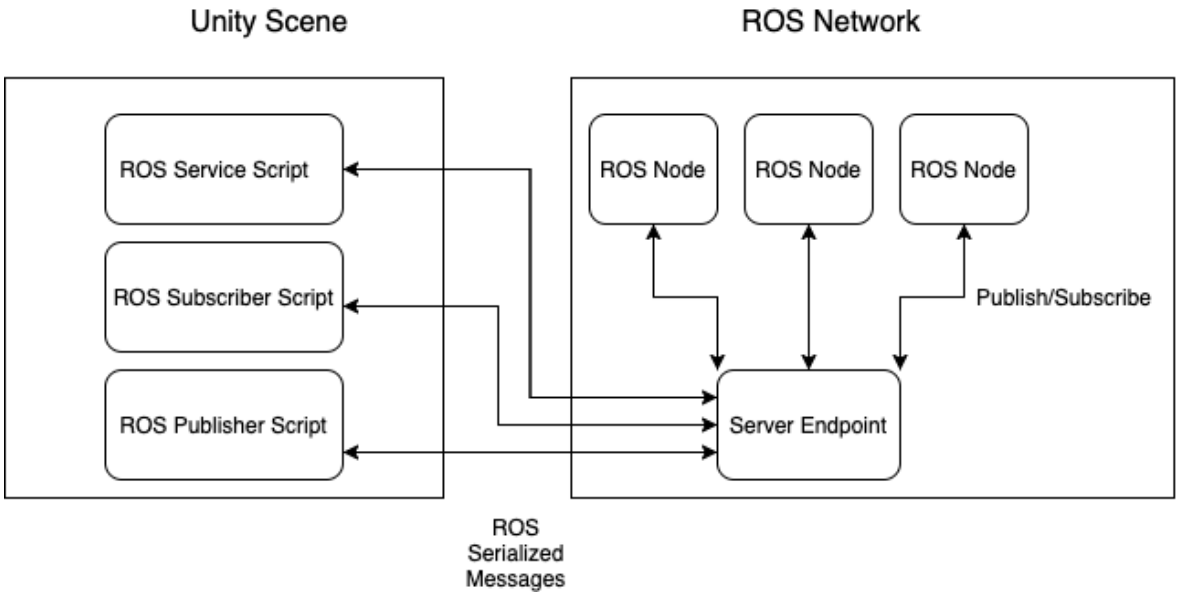


Figura 2: Esquema de ejemplo de estructura de comunicación de ROS.

Un ROS-TCP-endpoint que funciona como un nodo ROS, facilitando el intercambio de mensajes entre Unity y ROS. Se espera que los mensajes que se pasan entre Unity y ROS estén serializados de la misma manera que ROS los serializaría internamente. Para lograr esto, el complemento MessageGeneration (del repositorio ROS-TCP-Connector) puede generar clases en C#, que incluyen funciones de serialización y deserialización, a partir de archivos .msg de ROS.

El complemento ROSConnection (también del ROS TCP Connector) proporciona los scripts necesarios en Unity para publicar, suscribirse o llamar a un servicio.

4.3. Paquetes Unity extra

Paquetes y namespaces son establecidos por el propio Unity o bien por Siemens para el desarrollo de ROS-Unity Integrationpara la comunicación

con robots y mensajes relacionados con sensores y geometría, así como el sistema de entrada de inputs actual de Unity.

A continuación una breve explicación de los mismos.

1. System.Collections: Este paquete proporciona interfaces y clases que definen varias colecciones de objetos, como listas, colas, pilas, etc. Se utiliza comúnmente para manipular y trabajar con colecciones de datos.

2. System.Collections.Generic: Similar a System.Collections, este paquete contiene interfaces y clases para colecciones genéricas en .NET. Ofrece implementaciones más eficientes y seguras que las colecciones no genéricas.

3. UnityEngine: Este es el paquete fundamental de Unity y abre acceso a la API principal de Unity. Contiene clases y funciones esenciales para el desarrollo de juegos, como manipulación de objetos, física, renderización, entrada del usuario, etc.

4. UnityEngine.InputSystem: Introduce el sistema de entrada unificado de Unity. Proporciona una forma más moderna y flexible de manejar la entrada del usuario en comparación con el antiguo sistema de entrada.

5. UnityEngine.InputSystem.Utilities: Contiene utilidades adicionales relacionadas con el sistema de entrada, como funciones para mapear entradas, procesar eventos, etc.

6. System.Linq: Este paquete no es específico de Unity, sino parte del Namespace System. LINQ (Language Integrated Query) es una característica de C# que permite consultas integradas directamente en el código. Para el caso de este proyecto, únicamente se empleó para la realización de consultas en listas y selección componentes específicos.

7. Unity.Robotics.ROSTCPConnector: Este paquete pertenece a Unity Robotics y proporciona funcionalidades relacionadas con la conexión TCP para el Robot Operating System (ROS). Facilita la comunicación entre Unity y sistemas robóticos basados en ROS.

8. Unity.Robotics.ROSTCPConnector.MessageGeneration: Ofrece funciones relacionadas con la generación de mensajes para la comunicación en el entorno ROS.

9. RosMessageTypes.Sensor: Este paquete contiene tipos de mensajes específicos para sensores en ROS. Puede incluir definiciones de mensajes para datos de sensores utilizados en la aplicación. Para la estructura LaserScanMsg que maneja el topic /scan del Lidar.

10. RosMessageTypes.Geometry: Contiene tipos de mensajes específicos para geometría en ROS. Puede incluir definiciones de mensajes para representar información geométrica en la aplicación. Para la estructura TwistMsg que maneja el topic /cmd_vel de la parte de control cinemático del Turtlebot3.

11. RosMessageTypes.Tf2: Contiene tipos de mensajes relacionados con transformaciones (tf2) en ROS. Puede incluir definiciones de mensajes para representar transformaciones espaciales utilizadas en la aplicación. Para la estructura TFMessageMsg que maneja el topic /tf de la parte de control de posición del Turtlebot3.

5. Desarrollo del proyecto

A continuación se indicará el procedimientpo que fue llevado a cabo para la realización del proyecto, incluyendo la instalación de todo el sistema.

5.1. Instalación

5.1.1. Ubuntu/Linux

Con el objeto de instalar la versión 18.04 de Ubuntu, en uno ya instalado se abrió un terminal de PowerShell y se pueden buscar las versiones de Linux disponibles, para posteriormente instalar dicha versión correcta junto con el usuario y contraseña típico del SO basado en Unix escribiendo:

```
wsl -l -o
wsl --install Ubuntu-18.04
```

Ya que WSL no tiene acceso al entorno gráfico, se debe utilizar un software adicional para poder tener disponibilidad de ventanas gráficas cuando se trabaje en Linux.

Para la configuración de la GUI puede ser empleado el programa VcXsrc Windows X Server de libre acceso. Al ser una cuestión tan específica y no estar relacionado con la naturaleza del proyecto se considera de poca importancia como para ser mencionado en el apartado [Tecnologías empleadas].

Una vez descargado se procede a emplear el ejecutable, asegurándose de que las siguientes opciones estén configuradas correctamente:

1. Native OpenGL >Desactivado

2. Disable access control >Activado

Para posteriormente configurarlo tal como se muestra en las siguientes imágenes:

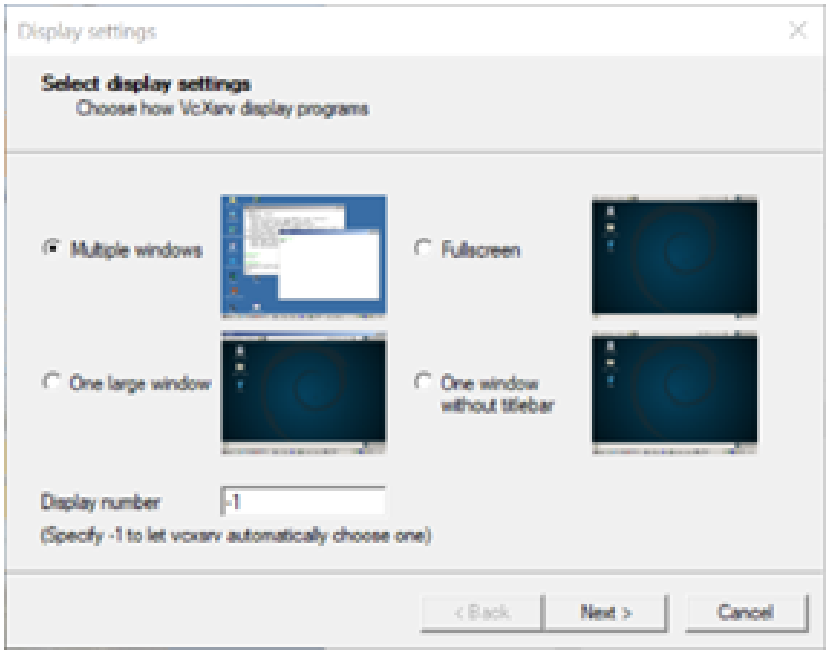


Figura 3: Display client.

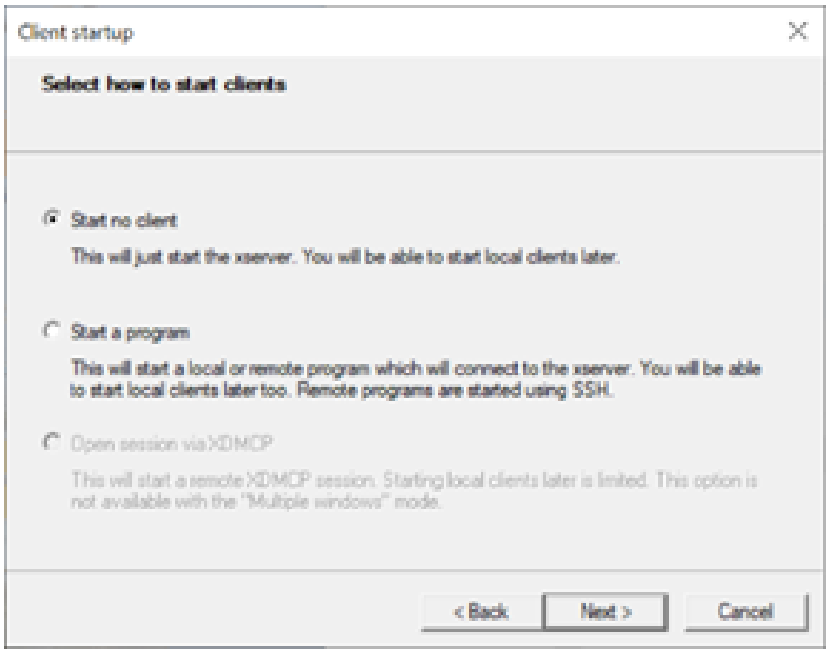


Figura 4: How to start client.

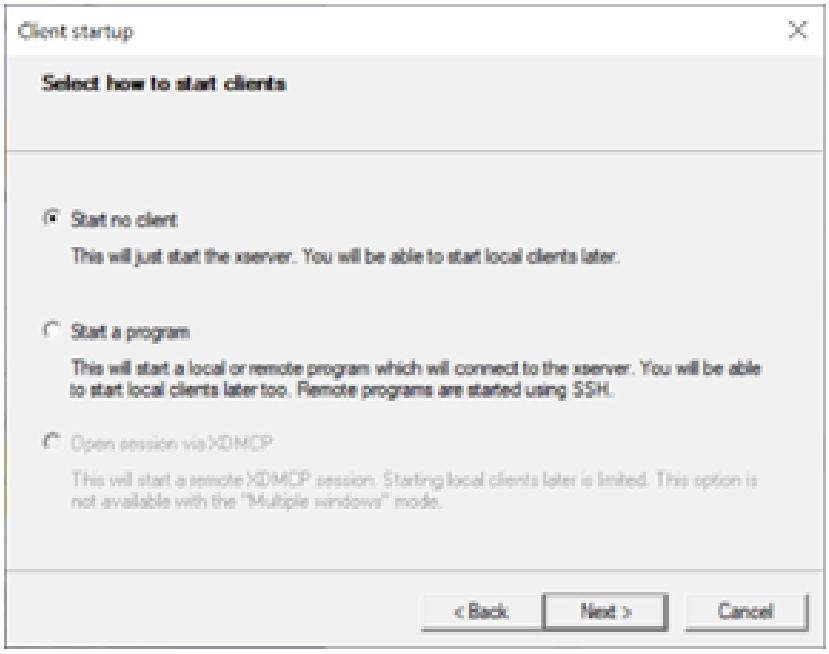


Figura 5: Extra settings.

Finalmente, sigueindo esta guía se debe exportar la dirección del XServer a través de la ventana de comandos y realizar la comprovación de que todo marche correctamente:

```
export DISPLAY=$(cat /etc/resolv.conf | grep nameserver | awk '{
    print $2}'):0
sudo apt update
sudo apt install x11-apps
xcalc
```

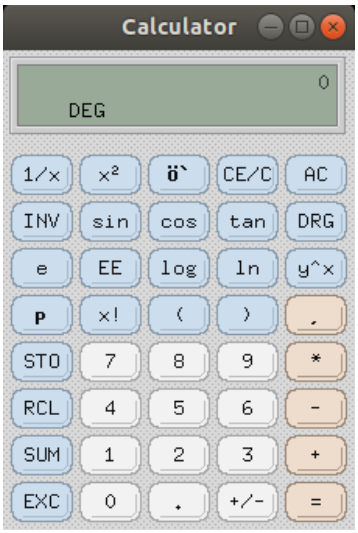


Figura 6: Calculadora producto del comando xcalc.

5.1.2. Unity

Para la instalación del motor Unity, el propio sistema tiene para la descarga el paquete. Basta con buscar e instalar el paquete que aparezca como "Unity Desktop";

```
sudo apt install ubuntu-unity-desktop -y
```

Con ello comenzará la descarga automática de todos los paquetes necesarios. Durante el proceso de configuración postinstalación, se le preguntará al administrador que gestor de inicio de sesión prefiere; Gnome (gdm) o Unity (Lightdm).

Tras esta instalación será conveniente el reinicio del sistema.

5.1.3. ROS Melodic Desktop Full

La siguiente instalación contiene tanto los paquetes de la versión ROS adecuada para Ubuntu 18.04 como los del Gazebo apropiado junto con el Rviz para la lectura de las funcionalidades slam. Y es que Ubuntu 18.04 requiere específicamente los paquetes de ROS Melodic.

Así pues en un terminal de Ubuntu se procederá a escribir la siguiente lista de comandos:

```
sudo sh -c 'echo deb http://packages.ros.org/ros/ubuntu $(
    lsb_release -sc) main > /etc/apt/sources.list.d/ros-latest.
    list'
sudo apt install curl # if you haven't already installed curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/
    ros.asc | sudo apt-key add -
sudo apt update
sudo apt install ros-melodic-desktop-full
echo source /opt/ros/melodic/setup.bash >> ~/.bashrc
source ~/.bashrc
sudo apt install python-rosdep python-rosinstall python-
    rosinstall-generator python-wstool build-essential
sudo apt install python-rosdep
sudo rosdep init
rosdep update
```

También es posible instalarlos uno por uno, pero el proceso es tedioso y puede dar lugar a incoherencias entre versiones de los programas y paquetes instalados para el desarrollo. Por tanto es de esperar el empleo específicamente el paquete provisto por la propia ROS para la instalación.

5.1.4. Turtlebot3 Burger

Si ROS Melodic Desktop Full fue instalado correctamente, Gazebo debe estar incluido entre sus paquetes. Ahora bien, para poder ejecutar el software de control provisto por la empresa para el Turtlebot3 en específico, los paquetes son los siguientes:

```
sudo apt-get install ros-melodic-joy ros-melodic-teleop-twist-joy \ ros-melodic-teleop-twist-keyboard ros-melodic-laser-proc \ ros-melodic-rgbd-launch ros-melodic-depthimage-to-laserscan \ ros-melodic-rosserial-arduino ros-melodic-rosserial-python \ ros-melodic-rosserial-server ros-melodic-rosserial-client \ ros-melodic-rosserial-msgs ros-melodic-amcl ros-melodic-map-server \ ros-melodic-move-base ros-melodic-urdf ros-melodic-xacro \ ros-melodic-compressed-image-transport ros-melodic-rqt* \ ros-melodic-gmapping ros-melodic-navigation ros-melodic-interactive-markers
sudo apt-get install ros-melodic-dynamixel-sdk
sudo apt-get install ros-melodic-turtlebot3-msgs
sudo apt-get install ros-melodic-turtlebot3
```

- Además, hay que indicar específicamente en el archivo .bashrc del directorio home de Linux una serie de configuraciones finales clave para ROS. Y es que este archivo contiene la sucesión de comandos que se producirá al abrir cualquier terminal. En primer lugar, mediante el comando en bash ifconfig ha de extraerse la IP pública (la inet del tercer párrafo);

```
f@f-NC-F5-573G-702H:~$ ifconfig
enp4s0f1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 54:ab:3a:99:fc:f0  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Bucle local)
        RX packets 21889  bytes 2385304 (2.3 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21889  bytes 2385304 (2.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

wlp3s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.131  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 2a0c:5a85:d20c:bf00:73f7:2cb0:841d:e2d3  prefixlen 64  scopeid 0x0<global>
        inet6 2a0c:5a85:d20c:bf00:d938:d69b:f304:5b35  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::e7a0:68fc:3b0:2c83  prefixlen 64  scopeid 0x20<link>
        ether c8:ff:28:41:fd:fb  txqueuelen 1000  (Ethernet)
```

```
        RX packets 1387687  bytes 1803532711 (1.8 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 327589  bytes 50955484 (50.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions
    0
```

Una vez en el archivo .bashrc, es importante asegurarse de que se realizó bien el proceso de establecimiento del workspace de catkin y los paquetes de ROS encontrando una ruta al directorio de ROS Melodic y a catkin.

Posteriormente declarar como se aprecia en el código la IP de la red del dispositivo obtenida con anterioridad y establecer específicamente el puerto 11311. NÓTESE que cada vez que se cambie de red, la IP cambiará, teniendo así que modificar estas dos líneas de código.

Finalmente, junto la configuración que especifica el uso del Turtlebot Burger y no su variante alternativa Waffle escribiendo en el mismo archivo al final:

```
source /opt/ros/melodic/setup.bash
source ~/catkin_ws/devel/setup.bash

export ROS_MASTER_URI=http://192.168.43.138:11311
export ROS_HOSTNAME=192.168.43.138

export TURTLEBOT3_MODEL=burger
```

Tras el proceso de instalación sería conveniente la realización una prueba en caso de fallo en la misma. Por ejemplo, empleando el programa de la prueba de teleoperación que de por si abrirá un core de ROS y peparará los topics de posición y movimiento que ya están listos para la simulación;

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

- En ocasiones procesos anteriormente iniciados pueden sufrir algún error y quedar activos, es por eso que de darse algún fallo, este se solucione simplemente ejecutando los siguientes dos comandos;

```
killall gzserver
killall gzclient
```

- Y a partir de aquí para simular un entorno simple y ver el robot desplazarse se podría abrir un mundo genérico;

```
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

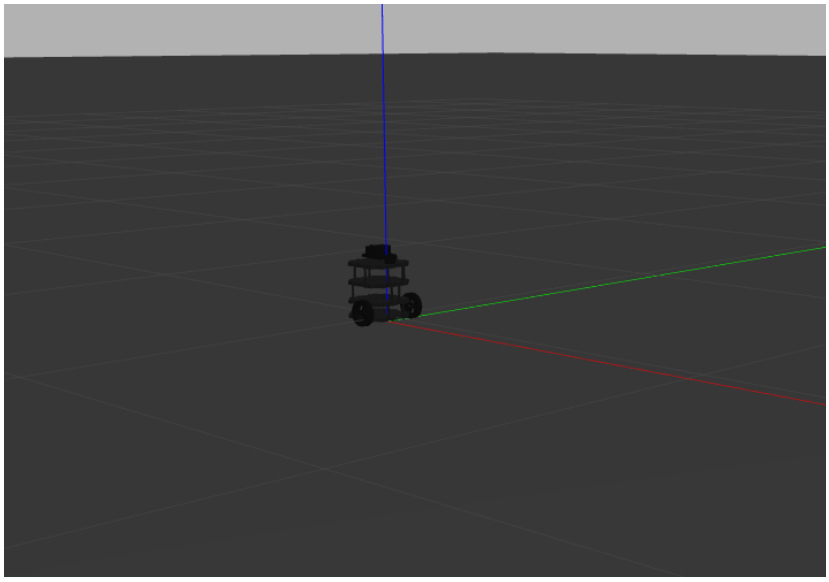


Figura 7: Gazebo con un mundo vacío.

- O directamente mostrar el estado de los topics cuando el robot interactúa en la simulación provista por la misma empresa.

```
roslaunch turtlebot3_gazebo turtlebot3_house.launch
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

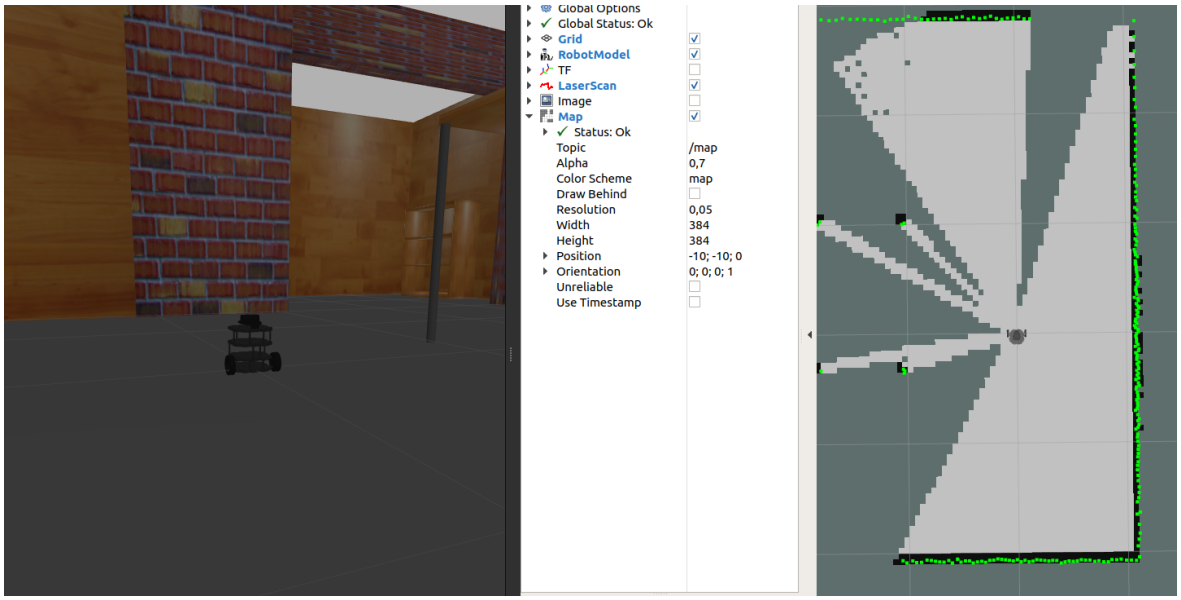


Figura 8: Gazebo y Rviz en un mundo con obstáculos.

Y entonces en el mismo terminal, al presionar las teclas WASD del movimiento se aprecia como la misma simulación lo se desplaza alterando el estado de los topics del robot y el movimiento del mismo.

```
f@f-NC-F5-573G-702
H:~$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
... logging to /home/f/.ros/log/508a42f2-9786-11ee-b1a0-c8ff2841fdfb/roslaunch-f-NC-F5-573G-702H-28574.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.1.131:39309/

SUMMARY
========

PARAMETERS
 * /model: burger
 * /rosdistro: melodic
 * /rosversion: 1.14.13

NODES
  /
    turtlebot3_teleop_keyboard (turtlebot3_teleop/turtlebot3_teleop_key)

ROS_MASTER_URI=http://192.168.1.131:11311

process[turtlebot3_teleop_keyboard-1]: started with pid [28589]

Control Your TurtleBot3!
---------------------------
Moving around:
        w
   a    s    d
        x

w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit
```

Figura 9: Ventana de comandos para controlar mediante WASD.

5.1.5. ROS-Unity Integration

El paquete ROS-TCP-Endpoint en Ubuntu puede ser empleado de dos formas;

- Empleando una máquina virtual en Docker con un repositorio y entorno de trabajo ya hecho,
- Desarrollando todo en un entorno de trabajo catkin.

Para el desarrollo del proyecto se empleó la última opción, instalando así en el directorio;

```
src/catkin_workspace git clone https://github.com/Unity-Technologies/ROS-TCP-Endpoint.git
```

Tras dicha instalación es prudente asegurarse de estar el workspace contado para catkin, realizando una prueba al montar un core de ROS abriendo un terminal de comandos y escribiendo:

```
source devel/setup.bash
roslaunch ros_tcp_endpoint endpoint.launch
```

Una vez el endpoint del server se haya iniciado, el terminal mostrará por pantalla un texto similar a;

```
[INFO] [1603488341.950794]: Starting server on
   192.168.50.149:10000.
```

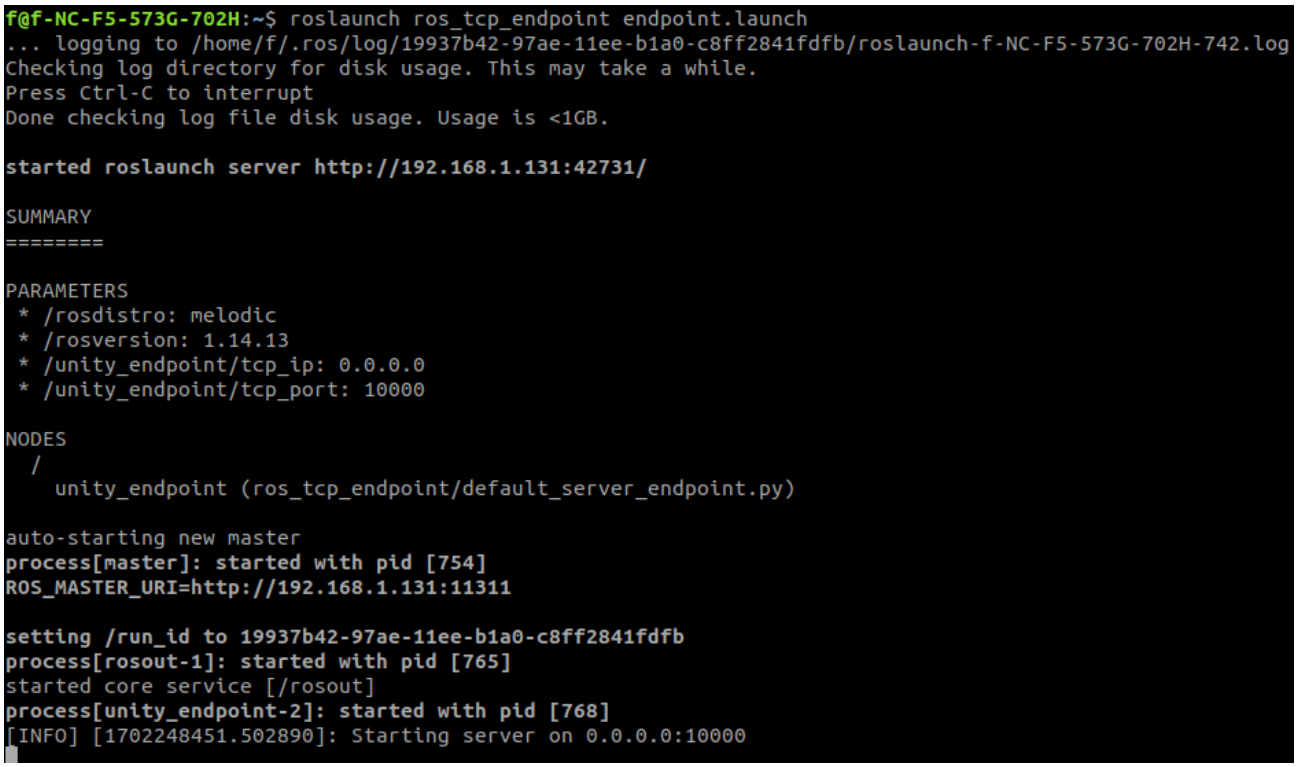


Figura 10: Apertura del endpoint ilustrada.

Por defecto, el servidor atenderá a la IP 0.0.0.0 (i.e. abriendose a todas las direcciones) y el puerto 10000, pero estas características son configurables. Para ser modificadas, en lugar del comando anterior, se deben insertar las siguientes especificaciones;

```
roslaunch ros_tcp_endpoint endpoint.launch tcp_ip:=[IP deseada]
   tcp_port:=[numero de puerto deseado]
```

5.1.5.1 Unity-Robotics-Hub package

Partiendo de la asunción de que Unity fue ya instalado en el SO y de que se creó un proyecto en 3D correctamente, hay que tener en cuenta que la versión empleada de Unity no debe ser posterior a la 2020.

Como se indica en la imagen, en el Package Manager (Administrador de Paquetes) se hace clic en el icono + en la esquina superior izquierda. Seleccionando Add package from git URL (Añadir paquete desde URL de git) para introducir en el cuadro los repositorios https://github.com/Unity-Technologies/ROS-TCP-Connector.git?path=/com.unity.robotics.ros-tcp-connector para instalar el paquete ROS-TCP-Connector y https://github.com/Unity-Technologies/ROS-TCP-Connector.git?path=/com.unity.robotics.visualizations.

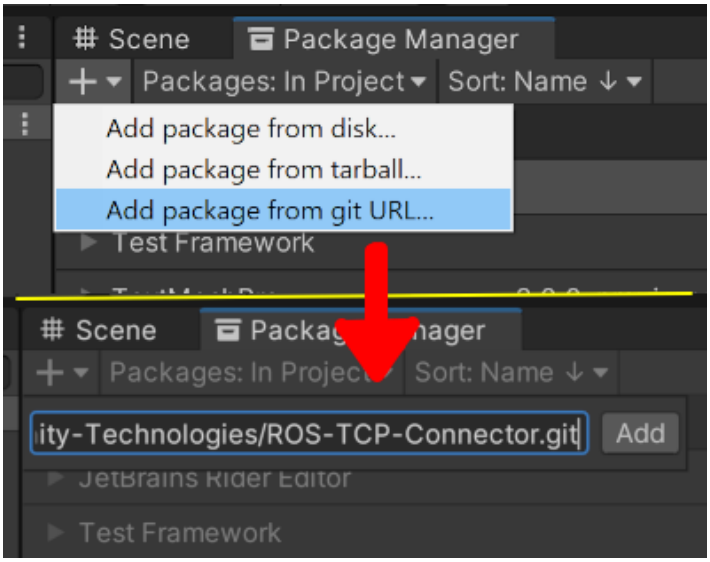


Figura 11: Inserción de paquete externo a Unity.

Habiendo instalado el paquete, en la barra superior en Robotics/ROS se ha de establecer la variable de ROS IP Address con la específicamente IP que fue configurada dentro del archivo .bashrc antes (de haber sido en Docker hubiera bastado con el valor predeterminado 127.0.0.1).

1. Instalación de las demos de Unity Robotics:

   Se deben copiar los paquetes unity_robotics_demo y unity_robotics_demo_msgs en la carpeta catkin_ws/src (el workspace de catkin). Se pueden descargar directamente desde el repositorio tutoriales/ros_unity_integration/ros_packages

2. Compilación de los nuevos paquetes:

   Ejecutar en la ventana de comandos para que ROS pueda encontrar los mensajes enviados;

```
catkin_make
devel/setup.bash
```

3. El path a los archivos del tutorial:

   Al igual que antes en la barra superior, en Robotics, Generate ROS Messages... Ha de cnfigurarse de tal manera que se enlace la ruta al directorio unity_robotics_demo/tutorials/ros_unity_integration/ros_packages/unity_robo
   Además activar las opciones Build N msgs y Build N srvs.

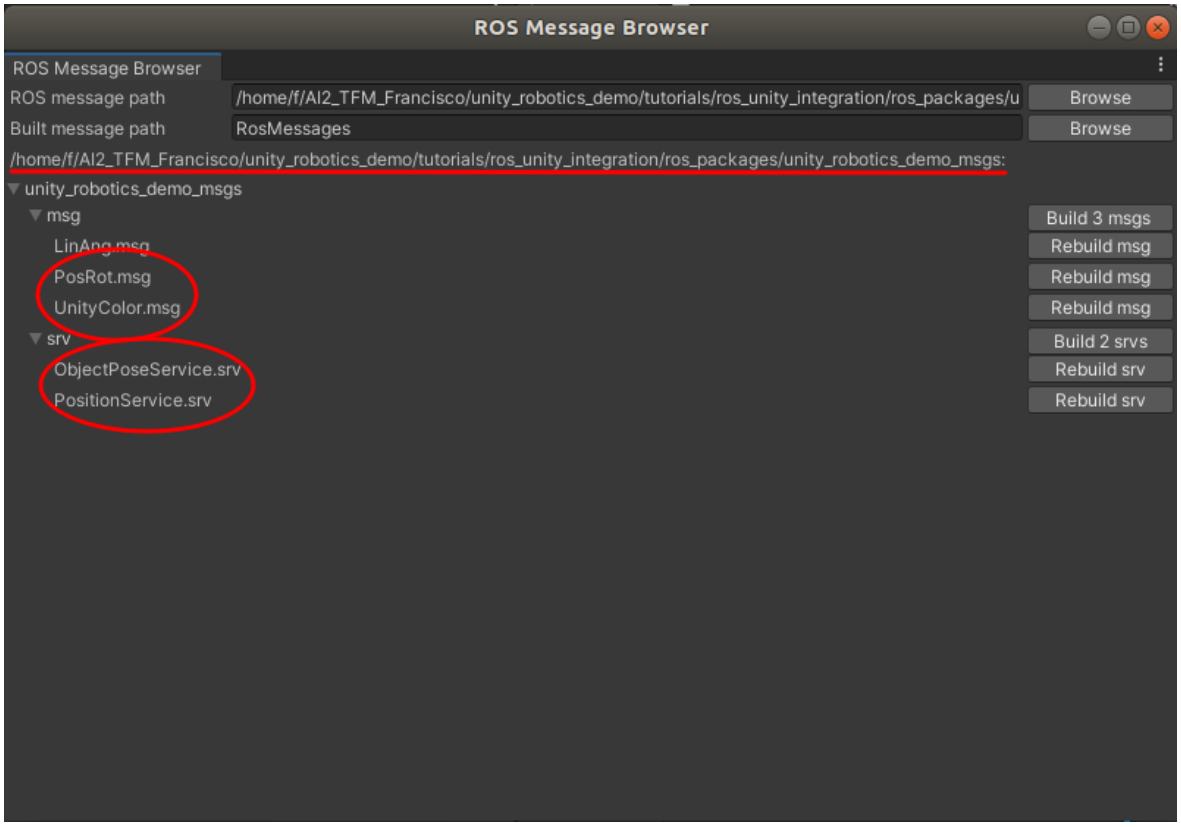


Figura 12: Ventana ROS Message Browser.

5.2. ROS-Unity Integration

El propio repositorio ofrece herramientas tutoriales de prueba de las funcionalidades ofrecidas por el paquete Integration:

5.2.1. Ejemplo de publicador - RosPublisherExample.cs

Un ejemplo de publicador del propio repositorio y que será aprovechado más adelante en el proyecto, empleando el paquete UnityRoboticsDemo previamente instalado:

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosMessageTypes.UnityRoboticsDemo;
```

En el ejemplo se establece una forma muy cómoda de gestionar el control del periodo de lectura del topic (escritura en este caso) como la variable publishMessageFrequency y el tiempo transcurrido en ticks timeElapsed. Además, el paquete requiere que se declare el nombre del topic.

```
public class RosPublisherExample : MonoBehaviour
{
    ROSConnection ros;
    public string topicName = pos_rot;

    // The game object
    public GameObject cube;
    // Publish the cube's position and rotation every N seconds
    public float publishMessageFrequency = 0.5f;

    // Used to determine how much time has elapsed since the
       last message was published
    private float timeElapsed;
```

Para el caso de este proyecto, si bien no se va a crear un nuevo tipo de topic sino que van a ser empleados los provistos por la propia Turtlebot3, es interesante comentar la solución empleada de uso de la estructura que servirá de plantilla para el desarrollo posterior.

El código de creación del topic implica que ha de emplearse la estructura del topic de nombre topicName declarado con la estructura del archivo PosRotMsg.msg enrutado anteriormente. Ha de crearse por supuesto la instanciación de la conexión.

```
    void Start()
    {
        // start the ROS connection
        ros = ROSConnection.GetOrCreateInstance();
        ros.RegisterPublisher<PosRotMsg>(topicName);
    }
```

En el ejemplo propuesto únicamente se altera la orientación de un objeto (un simple cubo).

La estructura del mensaje PosRotMsg tiene los atributos cartesianos junto con un cuaternión que será publicado directamente sobre el topic para que se muestre en pantalla.

```
    private void Update()
    {
        timeElapsed += Time.deltaTime;

        if (timeElapsed > publishMessageFrequency)
        {
            cube.transform.rotation = Random.rotation;

            PosRotMsg cubePos = new PosRotMsg(
                cube.transform.position.x,
                cube.transform.position.y,
                cube.transform.position.z,
                cube.transform.rotation.x,
                cube.transform.rotation.y,
                cube.transform.rotation.z,
                cube.transform.rotation.w
            );

            // Finally send the message to server_endpoint.py
               running in ROS
            ros.Publish(topicName, cubePos);

            timeElapsed = 0;
        }
    }
}
```

Así pues, se crea en la escena un cubo y un GameObject vacío de nombre RosPublisher para añadirle el script RosPublisherExample. Posteriormente se lleva el GameObject del cubo al parámetro del script de RosPublisher.

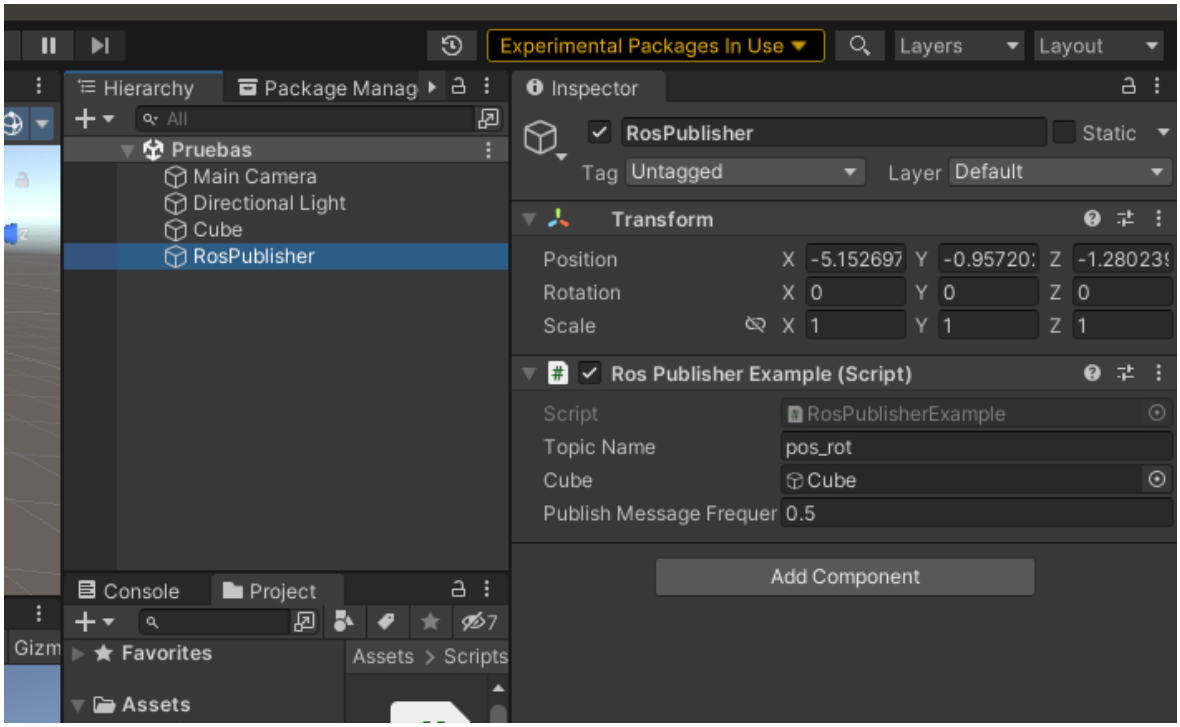


Figura 13: Configuración del script publicador.

Donde poseriormente en el bash se puede realizar la lectura del topic publicado;

```
source devel/setup.bash
roslaunch ros_tcp_endpoint endpoint.launch tcp_ip:=x.x.x.x
   tcp_port:=10000
rostopic echo pos_rot
```

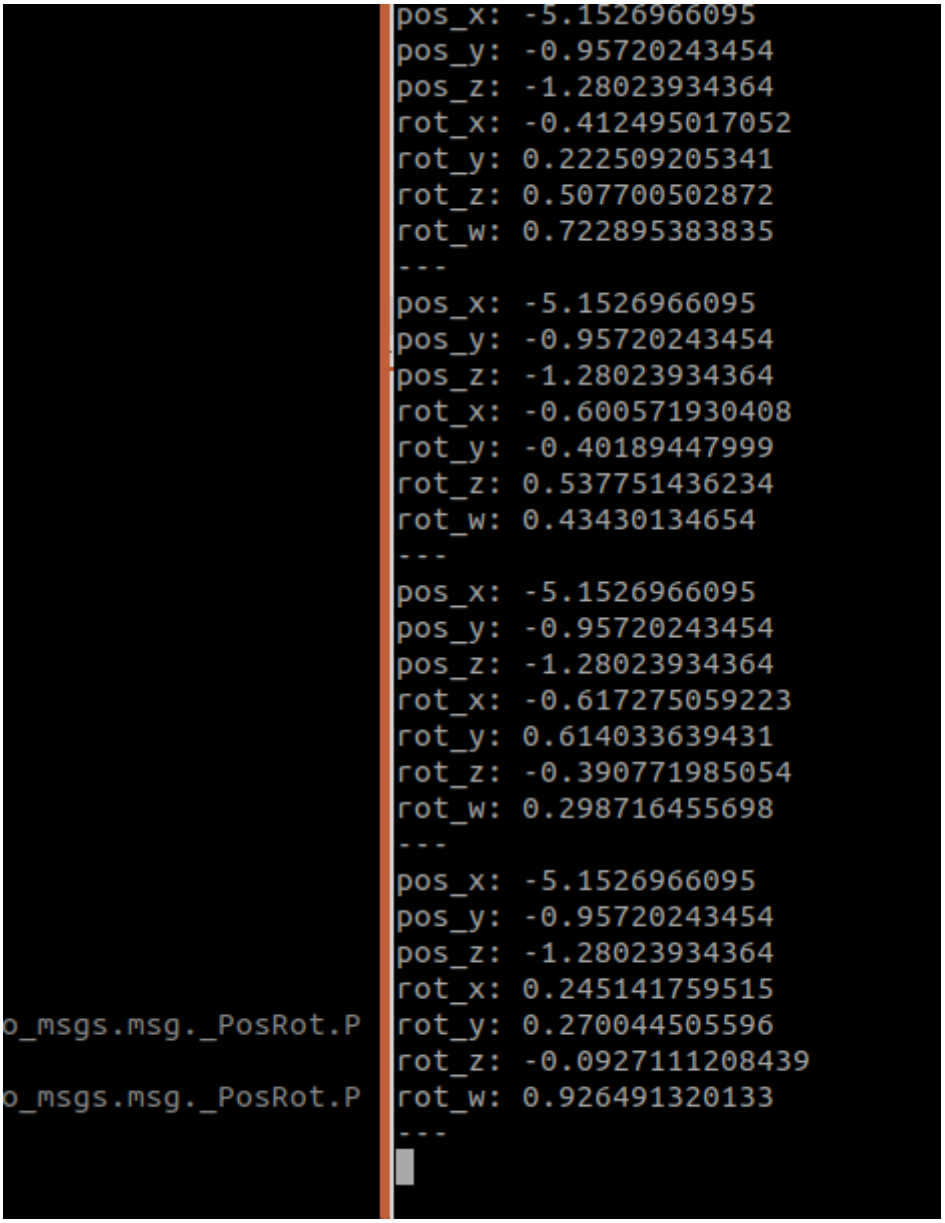


Figura 14: El resultado publicado.

5.2.2. Ejemplo de suscriptor - RosSubscriber.cs

Un ejemplo de suscriptor del propio repositorio y que será aprovechado más adelante en el proyecto, empleando el paquete UnityRoboticsDemo previamente instalado:

```
using UnityEngine;
using Unity.Robotics.ROSTCPConnector;
using RosColor =
    RosMessageTypes.UnityRoboticsDemo.UnityColorMsg;
```

Este caso requiere simplemente que esté recibiendo directamente una actualización a velocidad de tick y la estructura del topic RosColor almacenado

en el objeto colorMessage, que en este caso tiene los tres atributos RGB más A por el alfa de transparencia que van a estar asociados con el Renderer del GameObject (el cubo), siendo este Renderer el gestor gráfico de un GameObject cualquiera.

```
public class RosSubscriberExample : MonoBehaviour
{
    public GameObject cube;

    void Start()
    {
        ROSConnection.GetOrCreateInstance().Subscribe<RosColor>(color,
            ColorChange);
    }

    void ColorChange(RosColor colorMessage)
    {
        cube.GetComponent<Renderer>().material.color = new
            Color32((byte)colorMessage.r, (byte)colorMessage.g,
            (byte)colorMessage.b, (byte)colorMessage.a);
    }
}
```

A continuación, se crea en la escena un cubo y un GameObject vacío de nombre RosSubscriber para añadirle el script RosSubscriber. Posteriormente se lleva el script de RosSubscriberExample.cs al GameObject del cubo.

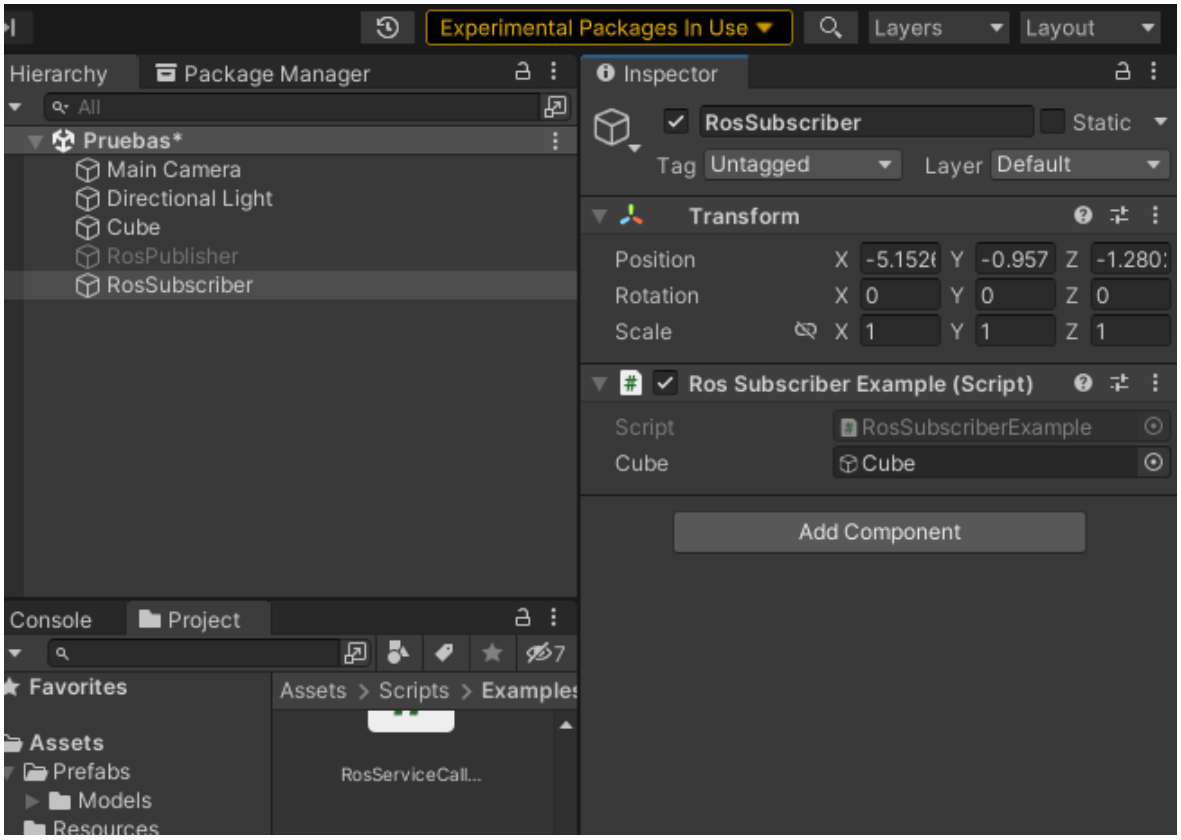


Figura 15: Configuración del script suscriptor.

Poseriormente en el bash;

```
roslaunch ros_tcp_endpoint endpoint.launch tcp_ip:=x.x.x.x
   tcp_port:=10000
rosrun unity_robotics_demo color_publisher.py
```

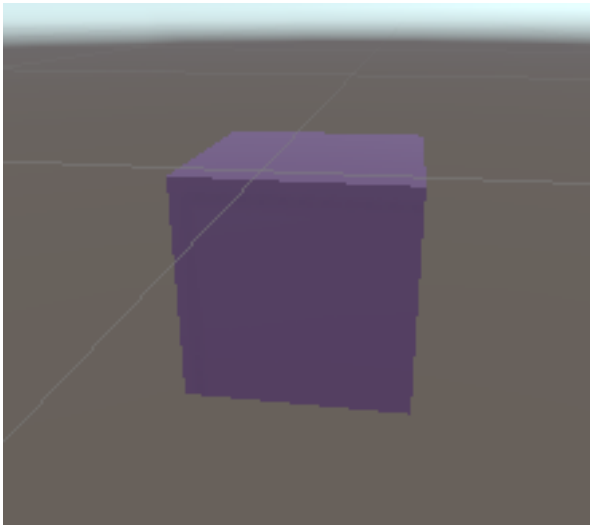


Figura 16: Cambio de color del cubo.

5.2.3. Servicios

Si bien esta opción no será empleada para este proyecto, es importante mencionar el concepto del servicio como opción de comunicación más ordenada y estructurada.

los servicios son una forma de comunicación entre nodos. A diferencia de los mensajes, que son utilizados para la comunicación de datos unidireccional, los servicios permiten la comunicación bidireccional entre nodos en ROS.

Un servicio consiste en dos partes: el cliente y el proveedor de servicios (service client y service server, en inglés). Estas dos partes se comunican a través de una solicitud y una respuesta. Cuando un nodo cliente desea realizar una tarea específica, envía una solicitud al nodo servidor a través de un servicio. El nodo servidor procesa la solicitud y envía una respuesta de vuelta al nodo cliente.

- Se trata de un lenguaje de programación orientado a objetos, lo que significa que se organiza alrededor de objetos que pueden contener datos y código que opera en esos datos.
- Proveedor de Servicio (Service Server): Es el nodo que inicia la solicitud del servicio. En tu ejemplo de Unity, RosServiceCallExample.cs actúa como el cliente de servicio, enviando solicitudes al servidor de servicio ROS.
- Solicitud (Request): Es la información enviada por el cliente al servidor para solicitar una acción específica.
- Respuesta (Response): Es la información devuelta por el servidor al cliente como resultado de la solicitud.

5.3. Instalación del modelo URDF

Una posible solución estaría en este breve tutorial de los pasos seguidos;

El simulador del robot contiene físicas que estarán programadas en un archivo .xacro provisto por la propia empresa junto con el URDF para poder producir un modelo que pueda ser simulado por Gazebo y Unity.

Al igual que en el caso de la instalación de los paquetes ros-tcp-connector y visualizations, en el Package Manager (Administrador de Paquetes) se hace clic en el icono + en la esquina superior izquierda. Seleccionando Add package from git URL e introducir en el cuadro el repositorio https://github.com/Unity-Technologies/URDF-Importer.git?path=/com.unity.robotics.urdf-importer#v0.5.2. Este proceso está también explicado en el siguiente tutorial.

Ya que es el propio Turtlebot3 el que dispone de su gestor de archivos en el directorio;

```
cd catkin_ws/src/turtlebot3/turtlebot3_description/urdf
```

Abriendo el terminal y escribiendo las siguientes líneas se procede a generar el archivo de datos URDF; (donde nombre_del_robot == turtlebot3_burger)

```
rosrun xacro xacro --inorder -o nombre_de_robot.urdf
   nombre_de_robot.urdf.xacro
```

O bien desde un terminal genérico, escribiendo antes el PATH global a los archivos

Y habiendo instalado el paquete para interpretar los URDF en Unity:

1. Dentro del proyecto de Unity/Assets crear un directorio URDF.
2. El archivo turtlebot3_burger.urdf resultante almacenarlo en Unity/Assets/URDF.
3. Copiar el directorio turtlebot3_description en Unity/Assets/URDF.
4. Clic secundario >3D Object >URDF Model (import) >Buscar el archivo .urdf del turtlebot3 >Abrir
5. En la ventana URDF Import Settings que se abrirá >Mesh Decomposer >Unity >Clic en Import URDF

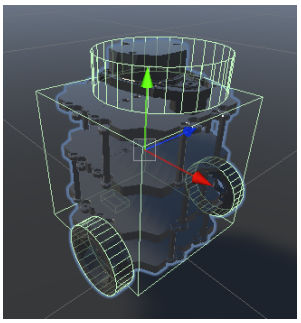


Figura 17: El modelo instalado.

5.4. Demo en Gazebo

Gazebo puede proveer de una simulación que aproveche los mismos topics de ROS para simular la interacción de este con las físicas del mismo. Aunque no será empleado en el proyecto por contar ya con el motor Unity, es interesante mostrar como Ciertos topics afectan al mismo Turtlebot3.

Y es que a parte de la simulación de control del robot y monitoreo de su Lidar...

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
roslaunch turtlebot3_gazebo turtlebot3_house.launch
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

...se va a mostrar un ejemplo que pretende la manipulación de la transformada del robot mediante el script del turtlebot3_teleop_key, ahora si en Unity.

5.4.1. Topic de Transformada - Gazebo

El siguiente desarrollo muestra como puede ser aprovechada la estructura del objeto de la transformada que Unity-Ros Integration emplea para recoger la información proveniente del topic de la transformada, en este caso la estructura Tf2 (que es la transformada) va a ser rellenada con los datos extraidos de la suscripción al topic /tf:

```
[...]
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
using RosMessageTypes.Tf2;
```

Aunque la plantilla de este código siga siendo la misma que la del anterior y esté comentado, hay que añadir que es prudente establecer un tiempo de retraso de lectura similar al tiempo de publicación del topic, a diferencia del ejemplo de la suscripción al color, para permitir al topic actualizar los datos y evitar procesamiento innecesario a la hoa de la modificación de la transformada.

```
public class PositionSubscriberTf : MonoBehaviour
{
    // Parametros fisicos.
    public GameObject robot;
    // Gestion de los topics.
    public string topicName = /tf;
    // Almacen de la informacion del mensage de la transformada
       al que se suscribe.
    private TFMessageMsg lastTransformMsg;
    [SerializeField] private ROSConnection ros;
    // Publish the robot's position and rotation every N seconds
    public float publishMessagePeriod = 0.1f;
    // Used to determine how much time has elapsed since the
       last message was published
```

```
    private float timeElapsed;

    void Start()
    {
        ros = ROSConnection.GetOrCreateInstance();
        ros.Subscribe<TFMessageMsg>(topicName, MovimientoRobot);
    }

    // Actualizar la posicion del cubo.
    void LateUpdate()
    {
        timeElapsed += Time.deltaTime;
        if (timeElapsed > publishMessagePeriod &&
            lastTransformMsg != null &&
            lastTransformMsg.transforms.Length > 0)
        {
```

Lo remarcable en este apartado es que la estructura del topic y el repositorio se almacena en un array cuyo valor inicial (o indexación 0) será el último valor de tipo Vector3Msg capturado y se encuentra en formato Euler, con lo que bastaría con cambiarlo a cuaternion para poder emplearlo en el posicionamiento del propio GameObject del robot.

```
            // Se ha de recoger la ultima transformada del
               array en el topic.
            var transformStamped =
               lastTransformMsg.transforms[0].transform;

            // Al recibir la pose en formato Vector3Msg, se
               establece en la del robot en forma de Vector3.
            robot.transform.position = new Vector3(
                (float)transformStamped.translation.x,
                (float)transformStamped.translation.z,
                (float)transformStamped.translation.y);

            // Al recibir la orientacion en QuaternionMsg se
               establece en la orientacion del robot como
               Quaternion.
            robot.transform.rotation = Quaternion.Euler(0, 90,
               0) * new Quaternion(
                (float)transformStamped.rotation.x,
                -(float)transformStamped.rotation.z,
                (float)transformStamped.rotation.y,
                (float)transformStamped.rotation.w);

            // Reinicia el conteo.
            timeElapsed = 0;
        }
    }
```

```
    // Actualizar valor de transformada.
    void MovimientoRobot(TFMessageMsg
        robotTransform){lastTransformMsg = robotTransform;}
}
```

Para ver bien el resultado sería conveniente activar el Gazebo mediate un mapa vacío con el robot en cuestión y un script de teleoperación;

```
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

5.5. Control mediante mando Xbox

Y de nuevo se requiere de la instalación de un paquete, en este caso para poder recibir señales directamente desde periféricos tales como ratones, teclados o mandos de consola. En primer lugar se va a importar nuevamente un repositorio con el paquete InputManager, solo que en este caso bastaría con buscarlo por su nombre en la pestaña All packages junto al +, directamente por su nombre.

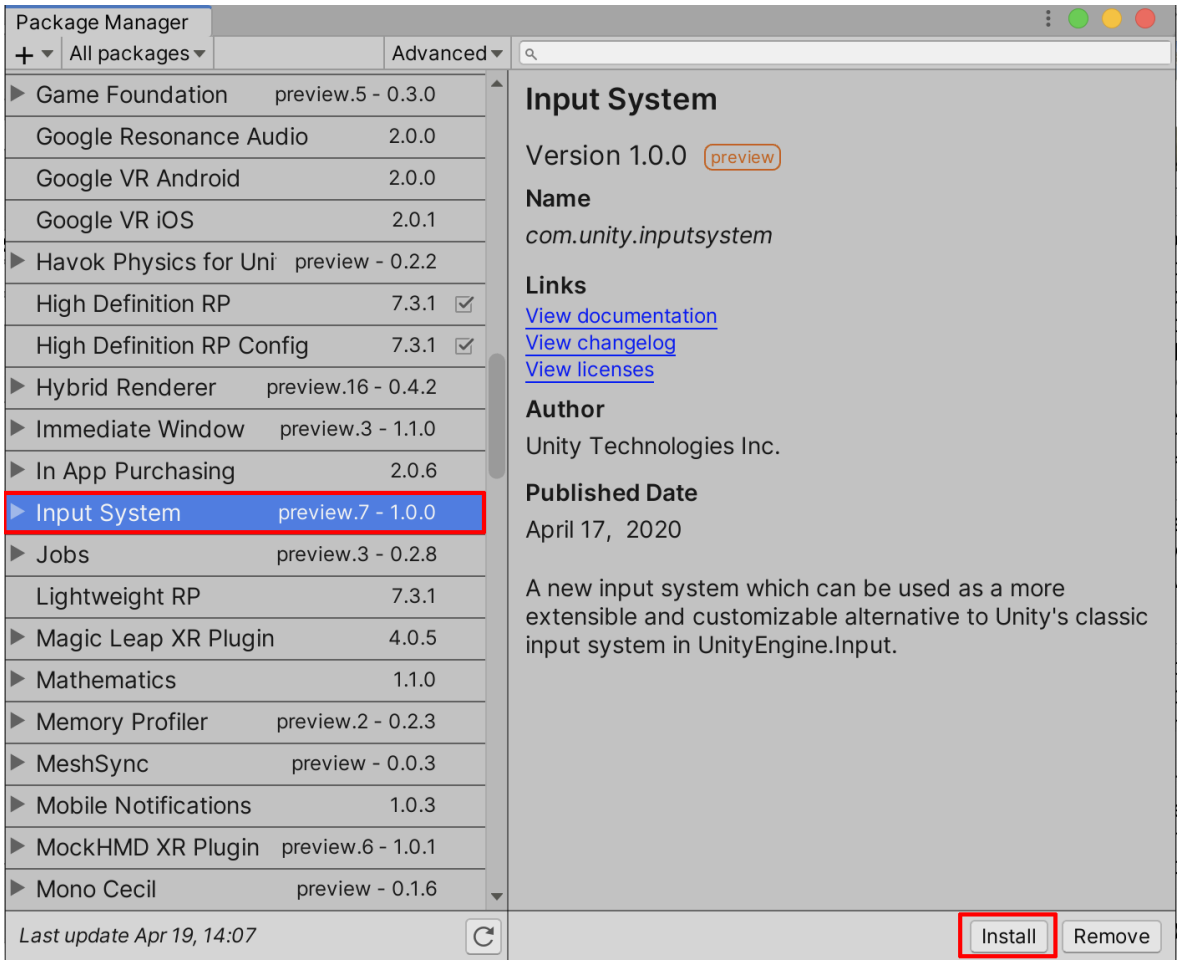


Figura 18: Instalación del paquete InputManager.

Todavía se requiere cierta configuración del menu superior >Edit Z Project Settings, donde en la opción Player es recomendable poner un nombre de compañía.

Más abajo, en Other Settings Está la opción Active Input Handling que bien puede tener seleccionada la opción Both, pero para este proyecto se empleó Input System Package (New).

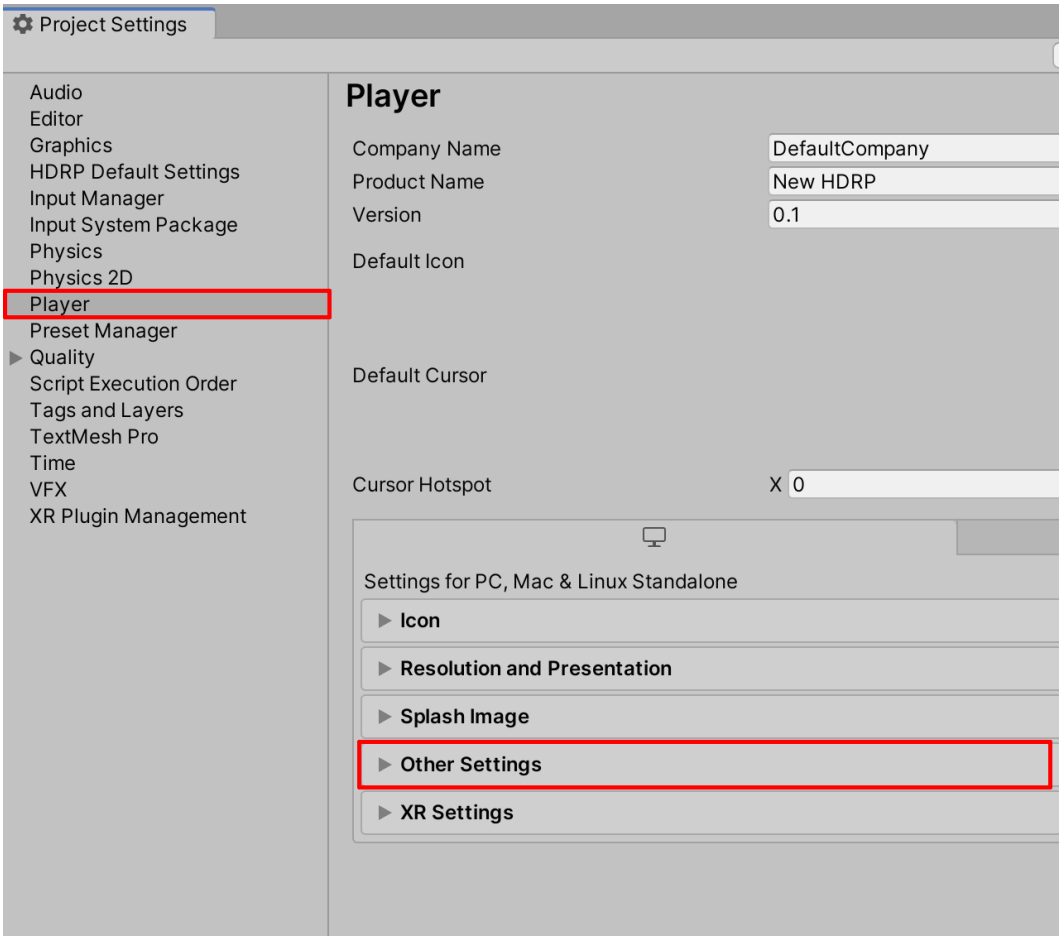


Figura 19: Ventana Project Settings.

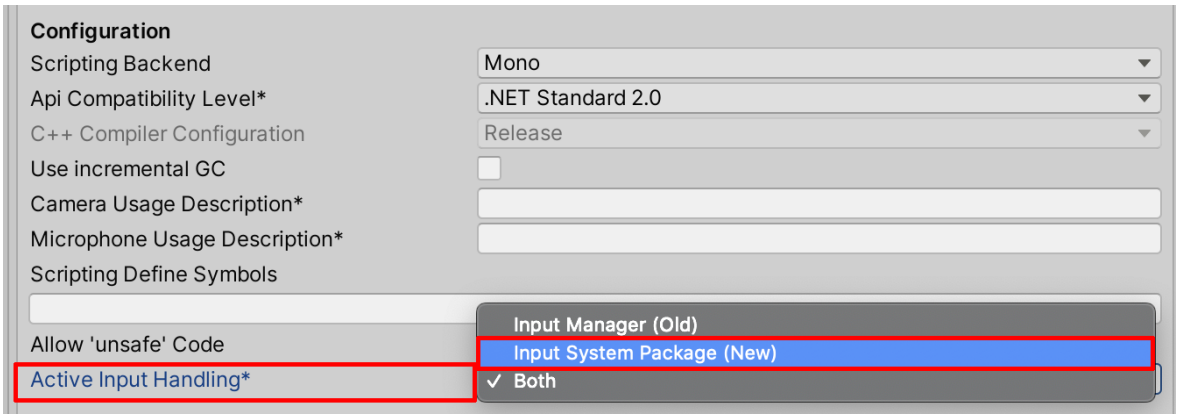


Figura 20: Opción Other Settings.

1. Se crea una nueva Input Action:
   a) Haciendo clic derecho en la ventana del proyecto.
   b) Create >Input Actions.

c) Es recomendable renombrar la nueva acción de entrada a InputMaster.

2. Configurar el esquema de control Xbox:

   a) En la ventana de acciones de entrada haciendo doble clic en InputMaster.

   b) Se selecciona No Control Schemes y agregar un nuevo esquema llamado Xbox Control Scheme.

   c) Dentro del esquema, Gamepad >Xbox Controller >Xbox Controller.

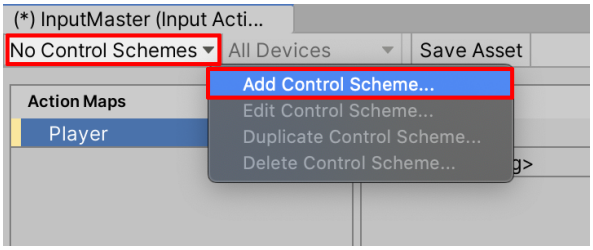


Figura 21: Opción Add Control Scheme...

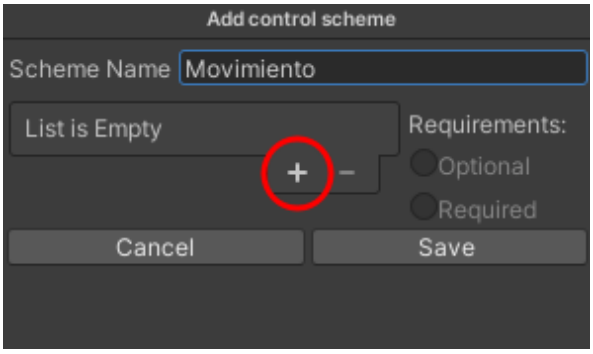


Figura 22: Solo se va a interactuar con el movimiento como input.

3. Se crea un mapa de acciones y asignar acciones:

   a) En el panel de mapas de acciones, un nuevo mapa llamado en este caso Robot0.

   b) En este caso se crea una nueva acción llamada Movimiento y con el clic secundario se asigna la segunda opción tantas veces se desee.

      - En este caso se realizaron cuatro lecturas: las flechas del teclado, las letras WASD, el joystick del mando de Xbox y las flechas del mando de Xbox.

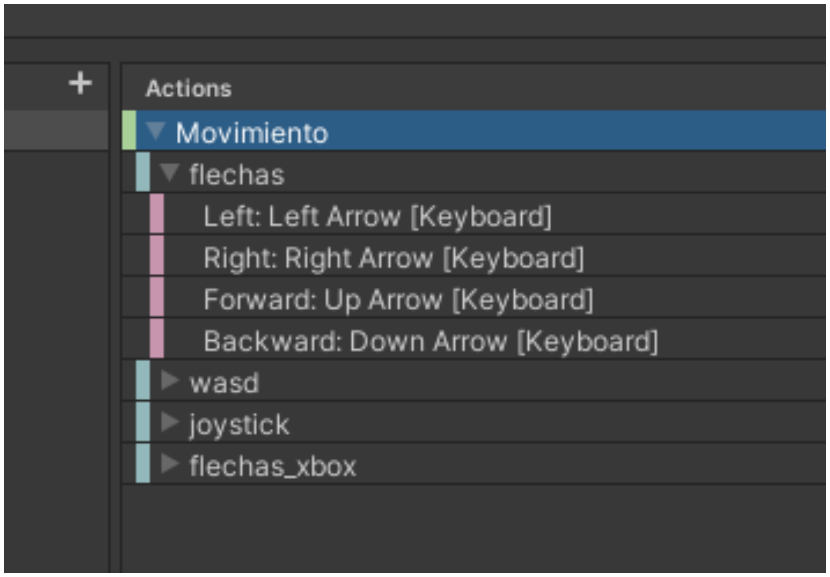


Figura 23: Opciones de movimiento.

4. Generar el script C#:

   a) Seleccionar InputMaster en la ventana del proyecto.

   b) Marcar la casilla Generate C# Class en la ventana del inspector.

   c) Aplicar los cambios para generar el script C# llamado InputMaster.cs.

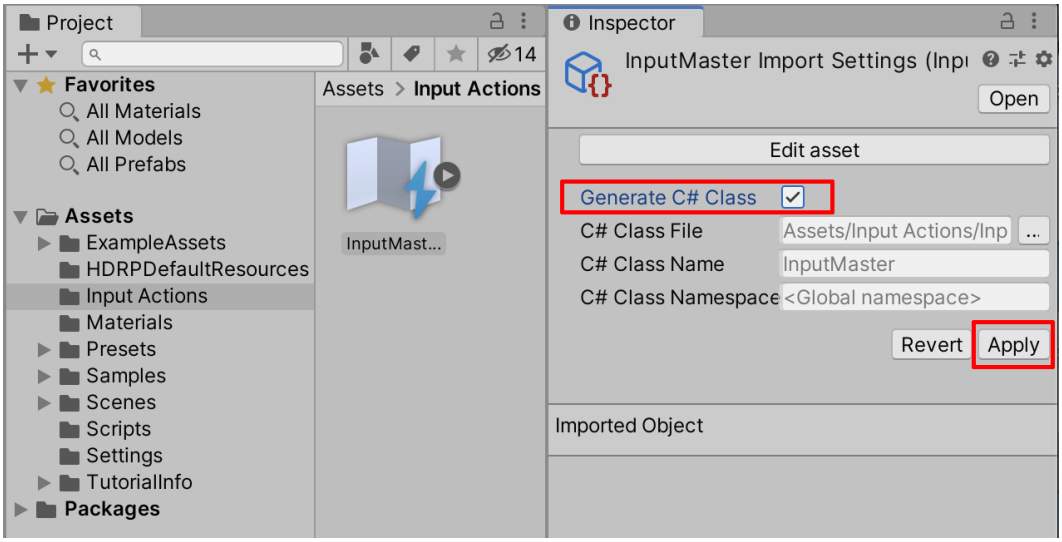


Figura 24: Creación del script.

El resultado es un script C# generado que maneja las acciones de entrada definidas, permitiendo la interacción con un controlador de Xbox en el proyecto de Unity.

5.5.1. Ejemplo de control de un cubo

Para este caso en el script que vaya a usarlo, tendrá que importar el paquete InputSystem.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;
```

En correcto funcionamiento del paquete InputSystem implica que han de ser declarados dos métodos que activen y desactiven las funcionalidades del mismo. Se pretende que este ejemplo de uso sirva a futuro para desarrollar las funcionalidades esperadas para el proyecto. Así pues, por ahora simplemente se propondrá una variable booleana (movementPressed) para activar y desactivar el movimiento.

```
public class CubeMovement : MonoBehaviour
{
    public float speed = 5f;

    InputMaster input;
    Vector3 currentMov = Vector3.zero;
    bool movementPressed;

    private void OnEnable() {input.Robot0.Enable();}
    private void OnDisable() {input.Robot0.Disable();}
```

Se ha de crear un objeto de tipo InputMaster que devolverá un dato matriz de tipo Vector3 (3D). Dicho dato va a tener valores oscilantes entre 0 y 1 dependiendo de que tipo de accionador se active (un joystick podría devolver valores decimales intermedios mientras que una tecla solo se activa y desactiva).

En este caso, la variable ctx (contexto) acumulará valores en función de dicha activación para establecer cuanto en el eje X y Z se ha desplazado el objeto (o en este caso si lo ha hecho para empezar) mediante la variable global que será aprovechada posteriormente.

```
public class CubeMovement : MonoBehaviour
{
    private void Awake()
    {
        input = new InputMaster();
        input.Robot0.Movimiento.performed += ctx => {
            currentMov = ctx.ReadValue<Vector3>();
            movementPressed = currentMov.x != 0 || currentMov.z
                != 0;
            };
    }
```

Para mayor comodidad y mejor tratamiento se ha recluido la función que gestiona este estado en una función a parte llamada constantemente por Update().

```
    void Update()
    {
        handleMovement();
    }
```

En definitiva, este valor acumulado de desplazamiento se irá sumando a los valores cartesianos del objeto transform, es decir, la transformada de la posición del objeto, reflejando así algún tipo de desplazamiento.

Esta operación se ha añadido la variable speed como propuesta de una variable fácil de ajustar externamente a la hora de modular la futura velocidad del robot, que puede llegar a verse perjudicado de realizar esfuerzos excesivos.

```
    void handleMovement()
    {
        Vector3 movimiento = new Vector3(currentMov.x * speed *
            Time.deltaTime, 0, currentMov.z * speed *
            Time.deltaTime);
        transform.position = transform.position + movimiento;
        Debug.Log(transform.position);
    }
}
```

Una vez generado el script con el código anterior y el InputMaster con el controlador del mando de Xbox/teclado se propone su prueba mediante un GameObject en 3D (un cubo por ejemplo). Como se indica en el tutorial mostrado previamente, bastaría con que en dicho objeto, en el inspector, Add Component >Player Input tal y como se muestra en la imagen;

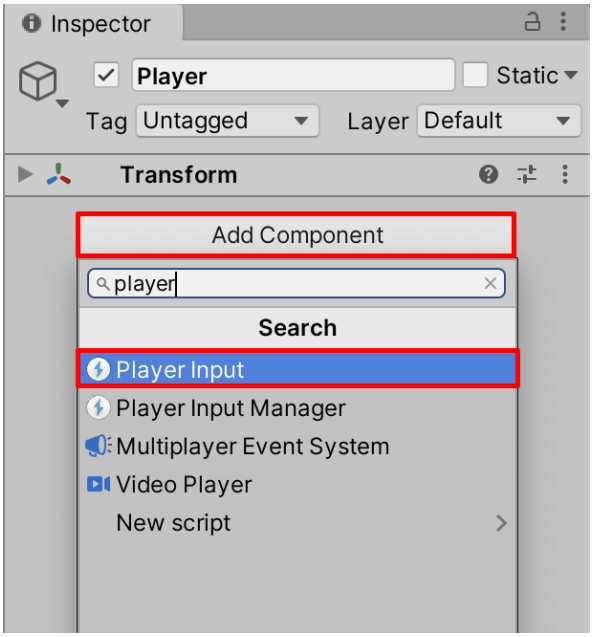


Figura 25: Como añadir un componente nuevo.

Y para finalizarlo, se ha de configurar añadiendole el InputMaster a Actions y modificar el Default Scheme tal como se muestra en la imagen.

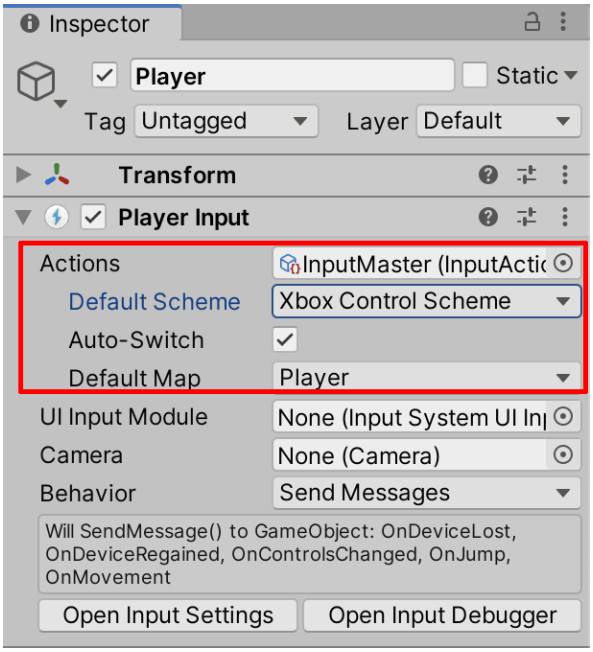


Figura 26: Configurción del InputMaster en un objeto.

5.5.2. Activación de Bluetooth en Ubuntu

Para configurar Ubuntu para recibir enlaces bluetooth se adjunta un pequeño tutorial simple en el que se indican los pasos seguidos a continuación.

Para modificar la variable requerida que permita la radiocomunicación mediante bluetooth es necesario manejar el SO como superusuario de la iguiente manera. En un terminal bastará con cambiar una N por una Y en un archivo;

```
sudo -s
sudo echo 'Y' > /sys/module/bluetooth/parameters/disable_ertm
```

Ahora bien, para activar el bluetooth mediante la configuración y el mando los pasos son;

1. Activar el bluetooth del ordenador.
2. Pulsar el boton X del mando de Xbox.
3. Mantener el botón X y presionar el boton pequeño encima de este hasta que aparezca el mando entre las opciones del bluetooth para clicar en el y establecer la conexión.

5.5.3. InputMaster + cmd_vel

Se procede con la combinación de los paquetes previamente comentados, de los cuales en este caso se emplearán el gestor de inputs y la clase que contiene la estructura del topic que controla la cinemática del robot y publicará en el topic cmd_vel.

```
[...]
using UnityEngine.InputSystem;
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
using RosMessageTypes.Geometry;
```

Por lo que este fragmento respecta, todo ha sido previamente comentado. Siendo TwistMsg la estructura que será llenada de la información producida por el InputMaster y transmitida por el topic /cmd_vel.

```
public class XboxPublisherTwist : MonoBehaviour
{
    private float MITIGADOR = 0.5F;

    // Gestor de InputSystem.
    InputMaster input;
    Vector3 currentMov;
    bool movementPressed;
    private void OnEnable() {input.Robot0.Enable();}
    private void OnDisable() {input.Robot0.Disable();}

    // Parametros fisicos.
    float BURGER_MAX_LIN_VEL = 0.22F;
    float BURGER_MAX_ANG_VEL = 2.84F;
    // Gestion de los topics.
```

```
public string topicName = /cmd_vel;

// Esta variable sera publicada.
private TwistMsg twistMessage;
[SerializeField] ROSConnection ros;
// Publish the robot's position and rotation every N seconds
public float publishMessagePeriod = 0.1f;
// Used to determine how much time has elapsed since the
   last message was published
private float timeElapsed;

private void Awake()
{
    input = new InputMaster();
    input.Robot0.Movimiento.performed += ctx => {
        currentMov = ctx.ReadValue<Vector3>();
        movementPressed = currentMov.x != 0 || currentMov.z
           != 0;
        };
}
```

Hay que tener en cuenta que se ha de cambiar el tipo de mensaje que se tiene que publicar al ser instanciado;

```
private void Start()
{
    // start the ROS connection
    ros = ROSConnection.GetOrCreateInstance();
    ros.RegisterPublisher<TwistMsg>(topicName); // Use
       'Twist' from RosMessageTypes.Geometry
}
```

La parte interesante de este script resulta ser el proceso que traduce el valor Z como el grado de avance que se desea para el robot y el valor X como la valocidad del giro que tendrá el

```
void Update()
{
    timeElapsed += Time.deltaTime;
    if (timeElapsed > publishMessagePeriod)
    {
        // Generacion de los arrays de posicion y
           orientacion.
        Vector3Msg linearVel = new Vector3Msg
        {x = currentMov.z * BURGER_MAX_LIN_VEL, y = 0, z =
           0};
        Vector3Msg angularVel = new Vector3Msg
        {x = 0, y = 0, z = -currentMov.x *
           BURGER_MAX_ANG_VEL * MITIGADOR};
        // Update the Twist message with the new velocities
```

```
            twistMessage = new TwistMsg(linearVel, angularVel);
                // Use 'Twist' from RosMessageTypes.Geometry

            // Publish the Twist message to the /cmd_vel to
                server_endpoint.py running in ROS
            ros.Publish(topicName, twistMessage);

            // Reinicia el conteo.
            timeElapsed = 0;
        }
    }
}
```

Finalmente este archivo puede ser ejecutado con una simple simulación vacía para comprobar que todo se realizó correctamente. Observando como al presionar en un mando de Xbox o un teclado, el robot simulado por Unity reacciona y como su homólogo en Gazebo hace lo propio.

```
roslaunch ros_tcp_endpoint endpoint.launch tcp_ip:=x.x.x.x
   tcp_port:=10000
roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

5.6. Marcadores de desplazamiento

En este caso el script no va a requerir trabajar con ningún topic de ROS y por eso sus librerías no van a ser importadas. Estas son Unity.Robotics.ROSTCPConnector, MessageGeneration y las librerías de las clases del/los topic/s en cuestión. Igual que anteriormente, la gestión del comportamiento de los marcadores la realizará un InputMaster enviando la información directamente desde el usuario.

```
[...]
using UnityEngine.InputSystem;

public class ArrowVisualizator : MonoBehaviour
{
    // Gestor de InputSystem.
    InputMaster input;
    Vector3 currentMov;
    bool movementPressed;
    private void OnEnable() {input.Robot0.Enable();}
    private void OnDisable() {input.Robot0.Disable();}
```

Se van a manipular GameObjects de tipo Prefab (prefabricados con la forma, color y orientación establecidos), dos para ser exactos, uno que muestre el giro que se está llevando a cabo y otro que muestre la dirección y sentido del avance, esto junto con el GameObject del propio robot para que se pueda ajustar la posición de los marcadores internamente relativos a la posición del mismo robot (por ejemplo sobre el mismo constantemente).

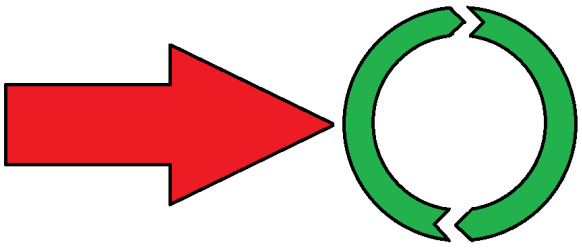


Figura 27: Plantillas usadas en los Prefabs de los marcadores.

```
    // Variables de la flecha.
    public GameObject indicadorMov;
    public GameObject indicadorGiro;
    public GameObject robot; // Para posicionar la flecha sobre
        el.
    public float velGiro = 200f; // Adjust the speed of
        rotation here
    public float velCrecer = 5f; // Adjust the speed of scaling
        here
    Quaternion targetRotation; // Cache the target rotation
    Vector3 originalScale; // Cache the original scale
```

Lo primero que se realiza para mejor optimización es generar mediante el comando Instantiate() desde el principio uno de cada marcador para posteriormente ser desactivados, así independientemente del estado inicial del escenario, siempre serán generados los marcadores.

A partir de ahí, una acción nueva es el tener en cuenta el valor de rotación del disco que muestra el giro del robot, que será convertido a cuaternión para poder aplicarlo a la transformada posteriormente.

```
private void Awake()
{
    // Sustituye el GameObject por las instancias de los
       indicadores.
    indicadorMov = Instantiate(indicadorMov, Vector3.zero,
       Quaternion.identity);
    indicadorGiro = Instantiate(indicadorGiro,
       Vector3.zero, Quaternion.identity);

    originalScale = indicadorMov.transform.localScale;

    input = new InputMaster();
    input.Robot0.Movimiento.performed += ctx => {
        currentMov = ctx.ReadValue<Vector3>();
        movementPressed = currentMov.x != 0 || currentMov.z
           != 0;
        };

    if (movementPressed)
        targetRotation =
           Quaternion.Euler(robot.transform.rotation.eulerAngles
           + Vector3.up * currentMov.x * 45);
}
```

Cuando el movimiento, ya sea giro o avance, no están activos han de desactivarse para indicar la parada del robot. Además se actualiza la longitud del marcador de desplazamiento para que siempre parta de estar a cero.

La parte clave son las primeras lineas, cuando el desplazamiento está activo, ya que se llamará a las siguientes dos funciones para que estas operen los marcadores.

```
void Update()
{
    if (movementPressed)
    {
        indicadorMov.SetActive(true);
        ActualizaFlecha();
        IndicaRotacion();
    }
    else
    {
```

```
            indicadorMov.SetActive(false);
            indicadorGiro.SetActive(false);
            indicadorMov.transform.localScale = originalScale;
        }
    }
```

El siguiente fragmento hace que el marcador se posicione siempre sobre el robot, vire un poco en caso de estar activo el giro (currentMov.x distinto de 0) y se estire en la dirección indicada para el avance.

La primera parte calcula las coordenadas del objeto y la segunda las aplica al objetivo, que en este caso es indicadorMov. Gracias a Time.deltaTime junto con velCrecer y velGiro y el método Lerp, se produce un efecto de gradualidad en el movimiento de los indicadores.

```
    void ActualizaFlecha()
    {
        // Posiciona la flecha sobre el robot.
        Vector3 posActualizada = robot.transform.position +
            Vector3.up * 1;
        // Orienta la flecha segun el robot y si se ordena giro
            la rota ligeramente.
        Vector3 rotActualizada =
            robot.transform.rotation.eulerAngles;
        rotActualizada.y += currentMov.x*45;
        // Cuando se ordene avanzar, la flecha se estirara en
            la direccion deseada.
        float targetScale = currentMov.z; // Target scale.

        // Posicionamiento inmediato sobre el robot.
        indicadorMov.transform.position = posActualizada;
        // Rotacion gradual del objeto hacia la ultima
            direccion.
        indicadorMov.transform.rotation =
            Quaternion.RotateTowards(indicadorMov.transform.rotation,
            Quaternion.Euler(rotActualizada), Time.deltaTime *
            velGiro);
        // Escalado gradual del objeto en la coordenada Z hasta
            el valor maximo.
        indicadorMov.transform.localScale =
            Vector3.Lerp(indicadorMov.transform.localScale, new
            Vector3(0.2f, 0.2f, targetScale + 0.2f),
            Time.deltaTime * velCrecer);
    }
```

Al igual que con el indicador de desplazamiento, el de giro calculará previamente las coordenadas relativas del robot para establecerlo sobre el mismo.

En este caso se ha optado por activar y desactivar el objeto internamente al método de operación ya que así el indicador desaparecerá y solo será

activado cuando explícitamente sea requerido, mientras que el de desplazamiento estará contraido en caso de estar parado el robot y solo girando.

Cabe destacar que dada la sensibilidad y velocidad del giro de Turtlebot3, se ha propuesto cierta tolerancia al error (aproximadamente 0.1 unidades de Unity), aunque esto es susceptible a valoraciones o incluso a ser quitado.

```
    void IndicaRotacion()
    {
        // Posiciona las flechas rotatorias sobre el robot.
        Vector3 posActualizada = robot.transform.position +
            Vector3.up * 0.7f;
        indicadorGiro.transform.position = posActualizada;

        // Si se ha pulsado giro (se considera +-0.1 como
            ruido), el disco rota en ese sentido.
        if(Mathf.Abs(currentMov.x) > 0.1f)
        {
            indicadorGiro.SetActive(true);
            // Rotacion gradual del indicador de giro.
            indicadorGiro.transform.Rotate(Vector3.up,
                Time.deltaTime * velGiro * currentMov.x);
        }
        else
        {
            indicadorGiro.SetActive(false);
        }
    }
}
```

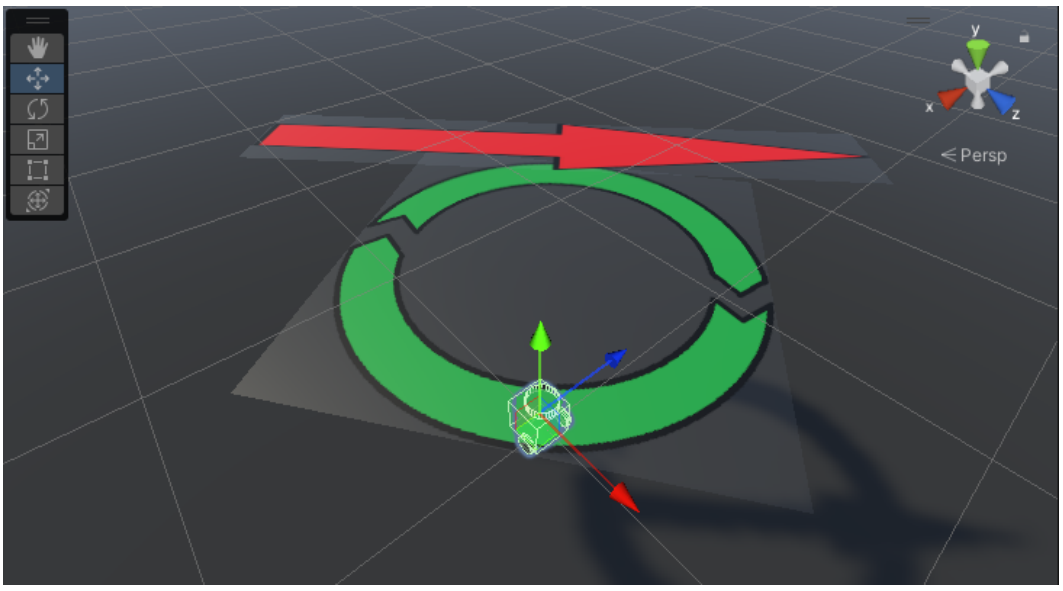


Figura 28: Los dos marcadores en uso.

5.7. Lectura del Lidar (LSD)

Esta será la última parte del proyecto y consistirá en la lectura del sensor de distancia rotatorio que mostrará un mapa aproximado de las colisiones de dicho láser con cualquier obstáculo.

5.7.1. Ejemplo de empleo en simulación con Gazebo

El único cambio notable se trata del empleo del paquete RosMessageTypes.Sensor para el empleo de la estructura LaserScanMsg.

Igual que previamente, bastaría con establecer el objeto del robot en el script a continuación, aunque por ahora no sirve ningún propósito. El GameObject que si se ha de establecer es uno que llevará añadido el componente Ros Connection, y será llamado por ejemplo RosConnector. El script en cuestión es PruebaScan.cs.

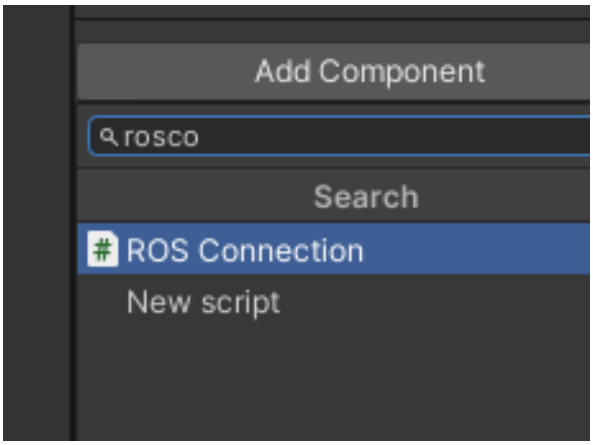


Figura 29: El componente con la IP y el puerto especificados.

Inicio más librerías del código;

```
[...]
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
using RosMessageTypes.Sensor;

public class PruebaScan : MonoBehaviour
{
    // Parametros fisicos.
    public GameObject robot;
    // Gestion de los topics.
    public string topicName = /scan;
    // Almacen de la informacion del mensage de la transformada
       al que se suscribe.
    private LaserScanMsg lastLaserScanMsg;
    [SerializeField] private ROSConnection ros;
    // Publish the robot's position and rotation every N seconds
```

```
    public float publishMessagePeriod = 0.1f;
    // Used to determine how much time has elapsed since the
       last message was published
    private float timeElapsed;

    void Start()
    {
        ros = ROSConnection.GetOrCreateInstance();
        ros.Subscribe<LaserScanMsg>(topicName, LIDARRobot);
    }
```

Lo interesante de este desarrollo es como se muestra que el paquete produce un array cuyo primer valor es siempre el último y en este caso de prueba únicamente se muestra por la Consola el valor transmitido por el topic del láser.

```
    // Actualizar la posicion del cubo.
    void LateUpdate()
    {
        timeElapsed += Time.deltaTime;
        if (timeElapsed > publishMessagePeriod &&
           lastLaserScanMsg != null)
        {
            for (int i = 0; i < 60; i++) // disparos
            {
                float dist = lastLaserScanMsg.ranges[i];
                Debug.Log(dist);
            }
            // Reinicia el conteo.
            timeElapsed = 0;
        }
    }
    // Actualizar valor de odometria.
    void LIDARRobot(LaserScanMsg robotScan) {lastLaserScanMsg =
       robotScan;}
}
```

Para comprobar el funcionamiento de los láseres, en este caso se ha de simular ahora si una casa con obstáculos sobre los que colisione el láser;

```
1 roslaunch ros_tcp_endpoint endpoint.launch tcp_ip:=x.x.x.x
     tcp_port:=10000
2 roslaunch turtlebot3_gazebo turtlebot3_house.launch
```

(Opcionalmente)

```
1 roslaunch turtlebot3_slam turtlebot3_slam.launch
```

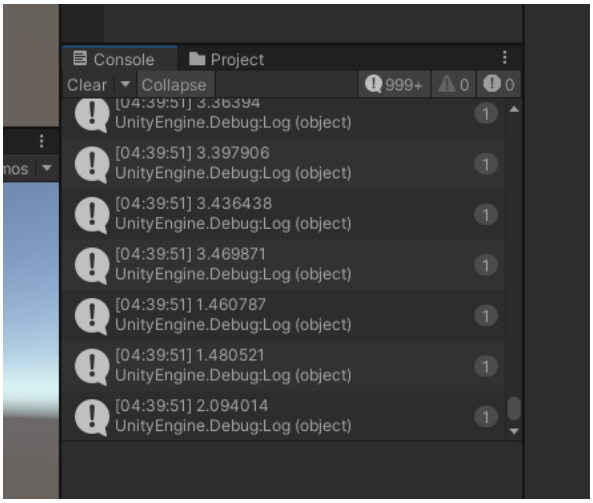


Figura 30: Valores de distancia de cada uno de los disparos consecutivos.

5.7.2. Solución 1: Robot fijo + Mapa móvil

En la siguiente propuesta se sugiere un script ObstacleFinderV1.cs que va a llevar la combinación de las herramientas acumuladas previamente.

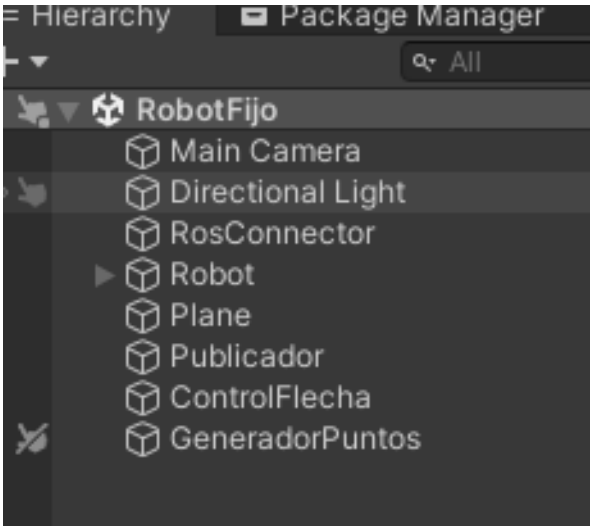


Figura 31: Todos los objetos del caso de estudio.

El controlador de las flechas marcadores es el mismo y se omitirá la explicación de las librerías asociadas que se emplean también en este caso de estudio. Aún así es interesante recordar que el script funciona mediante se le asocian los objetos predefinidos (Prefab) de la flecha de dirección y la de giro junto con el robot para capturar su transformada (para la posición).

En este caso se va a emplear el script del publicador del movimiento que se comunica mediante el topic /cmd_vel, ya explicado también previamente que posteriormente se conectó con la capacidad de introducir inputs mediante la herramienta InputManager.

La novedad en este caso será el proceso de recoger la información enviada a través del topic /scan y convertirla en datos útiles para recrear dichos puntos mediante el array trigon (donde posteriormente se mostrará como las coordenadas trigonométricas se calculan y almacenan).

```
[...]
using Unity.Robotics.ROSTCPConnector;
using Unity.Robotics.ROSTCPConnector.MessageGeneration;
using RosMessageTypes.Sensor;

//
   https://github.com/Unity-Technologies/ROS-TCP-Connector/blob/main/com.uni
public class ObstacleFinderV1 : MonoBehaviour
{
    // Parametros fisicos.
    private int disparos; // El numero de disparos que
       efectuara el Lidar.
    private float[,] trigon; // Valores cartesianos unitarios
       de todos los disparos del Lidar.
    public GameObject marcaPrefab; // El prefab del punto que
       sera generado para mostrar un obstaculo.

    // Para limitar la superposicion de marcas.
    public string topicName = /scan;
    private LaserScanMsg lastLaserScanMsg; // Almacen de la
       informacion del mensage al que se suscribe.
    [SerializeField] private ROSConnection ros;
    // Publish the robot's position and rotation every N seconds
    public float publishMessagePeriod = 0.1f;
    // Used to determine how much time has elapsed since the
       last message was published
    private float timeElapsed;
```

Además se van a añadir un gradiente para alterar los colores de las marcas de las colisiones del Lidar con el entorno y un array de colores para aligerar el proceso de cálculo, no calculandolo sino indexando su valor (mucho más ligero a la hora de ser procesado y funcionar).

```
    public Gradient gradienteColor;
    private Color[] markerColors; // Array de colores
       precalculado en base a la distancia.
```

Tras suscribirse al topic que maneja la información del Lidar y establecer la estructura LaserScanMsg para la recogida de datos, además se establecerán los valores del gradiente (por ejemplo entre 0 y 100) desde el principio, esto para acelerar el procesado y optimizarlo.

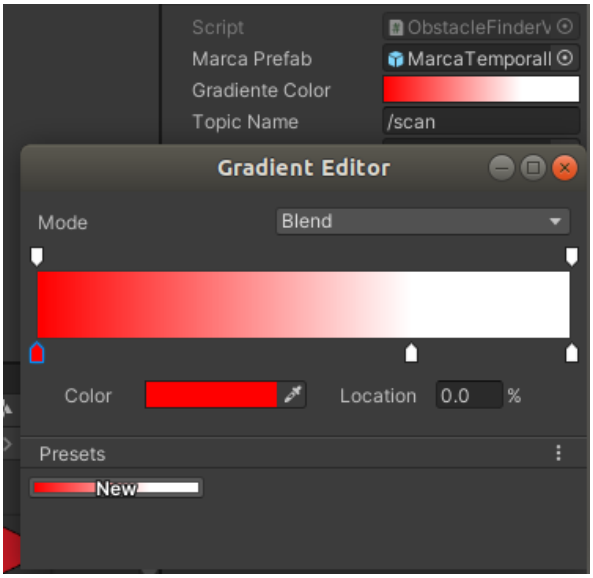


Figura 32: Introducción del gradiente.

```
void Awake()
{
    ros = ROSConnection.GetOrCreateInstance();
    ros.Subscribe<LaserScanMsg>(topicName, LIDARRobot);

    // Array de marcadores de color precalculados basados
       en un gradiente.
    markerColors = new Color[100]; // Por ejemplo un rango
       de 100 gradientes.
    for (int i = 0; i < markerColors.Length; i++)
    {
        float distNormal = i / 100f;
        markerColors[i] =
            gradienteColor.Evaluate(distNormal);
    }
}
```

Al igual que antes, se extrae el valor (almacenado en este caso en lastLaserScanMsg) y se produce en el escenario un GameObject Prefab llamado Marca mediante la función CreaMarca() que será comentada posteriormente.

Con el objeto de reducir el peso del procesamiento en cada iteración del programa se propone;

- Representar por ejemplo solo los puntos pares.
- Representar solo los puntos que entraron en colisión. Esto es, que su distancia indicada por el sensor del láser sea un valor real y no supere la mitad de por ejemplo la distancia máxima a la que teóricamente debería llegar el láser.

```
// Actualizar la posicion del cubo.
void LateUpdate()
{
    timeElapsed += Time.deltaTime;
    if (timeElapsed > publishMessagePeriod &&
       lastLaserScanMsg != null)
    {
        for (int i = 0; i < disparos; i++) // Cada uno de
           los disparos.
        {
            float dist = lastLaserScanMsg.ranges[i];
            // Para reducir carga de trabajo para las
               gafas, se representaran solo los indices
               impares en caso de estar lejos.
            // Ademas, se filtran las distancias que sean
               infinitas.
            if ((i % 2 == 0 || dist >
               lastLaserScanMsg.range_max/2) &&
               !float.IsInfinity(dist))
            {
                CreaMarca(dist, i);
            }
        }
        timeElapsed = 0; // Reinicia el conteo hasta el
           siguiente tick.
    }
}
```

Este pequeño método extraerá el valor del topic /scan y se quedará con varios valores establecidos por el topic; la cantidad de disparos del sensor y el ángulo de separación que trazaba cada uno de los láseres entre ellos. De esta manera mediante una simple operación trigonométrica se pueden sacar las coordenadass relativas en cada momento y llenar el array trigon de vectores unitarios que posteriormente servirán para hayar las coordenadas reales del obstáculo.

```
void LIDARRobot(LaserScanMsg robotScan) // Actualizar valor
   del Lidar.
{
    lastLaserScanMsg = robotScan;
    disparos = lastLaserScanMsg.ranges.Length;
    trigon = new float[disparos, 2];
    float anguloDisparo = 0;
    for (int i = 0; i < disparos; i++)
    {
        trigon[i, 0] = (float)Math.Cos(anguloDisparo);
        trigon[i, 1] = (float)Math.Sin(anguloDisparo);
        anguloDisparo += lastLaserScanMsg.angle_increment;
    }
```

```
    }
```

Función esencial ya que mediante las trigonometrías (unitarias) y la distancia percibida en cada disparo se puede extraer la coordenada que sirva para la instanciación.

```
    void CreaMarca(float dist, int rayo) // Genera una marca en
       el mapa en funcion de la distancia de choque y el rayo
       que le toque del array.
    {
        Vector3 posicion = new Vector3(-trigon[rayo, 1] * dist,
           0, trigon[rayo, 0] * dist); // Posicion cartesiana
           con trigonometria y distancias.
        GameObject marca = Instantiate(marcaPrefab, posicion,
           Quaternion.identity);
```

Extrayendo la distancia máxima y mínima que alcanza el láser, se puede interpolar un valor entre 0 y 1 que sirva para la indexación del color que tomará el GameObject Prefab Marcador.

```
        // Colorear la marca del color correspondiente:
        // Calcula la distancia y la normaliza entre 0 y 1.
        float distNormal = Mathf.Clamp01(dist /
           (lastLaserScanMsg.range_max -
           lastLaserScanMsg.range_min));
        // En funcion de la distancia indica el indice del tono
           de color que debe buscar en el array.
        int colorIndex = Mathf.FloorToInt(distNormal *
           (markerColors.Length - 1));
        Renderer renderer;
        if (marca.TryGetComponent(out renderer))
        {
            renderer.material.color = markerColors[colorIndex];
               // Establece el tono que le corresponde segun el
               indice.
        }
    }
}
```

La idea tras todo este proceso es que se estén generando puntos constantemente y estos se aniquilen continuamente asociando el script Autodestruccion.cs al mismo Prefab de la marca para que cada marca lleve su conteo con procesos paralelos entre si;

```
[...]
public class Autodestruccion : MonoBehaviour
{
    public float tiempoDeVida = 0.1f; // Time en segundos.

    private void Start()
    {
```

```
        // Invoke el metodo Destruye pasado el TiempoDeVida.
        Invoke("Destruye", tiempoDeVida);
    }

    private void Destruye() {Destroy(gameObject);} // Elimina
        el GameObject al que esta asignado este script.
}
```

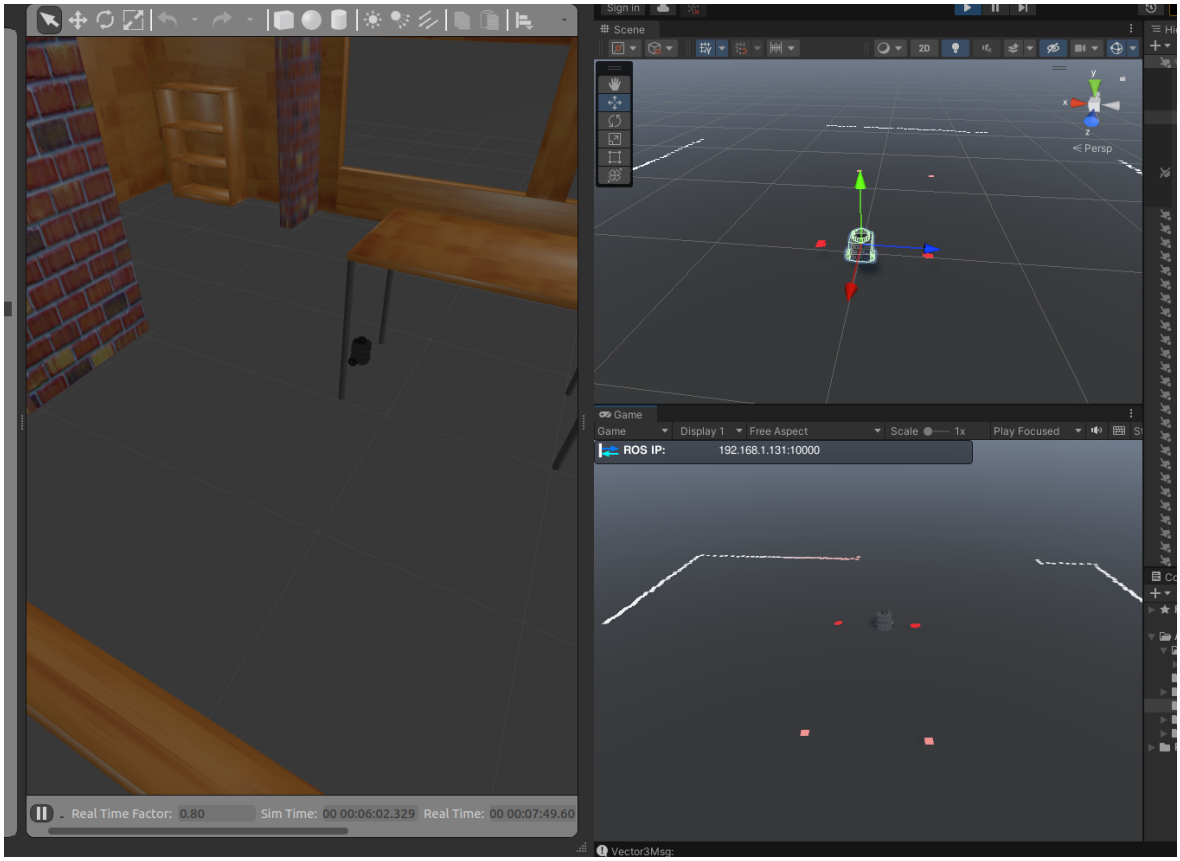


Figura 33: Posición inicial, robot fijo centrado.

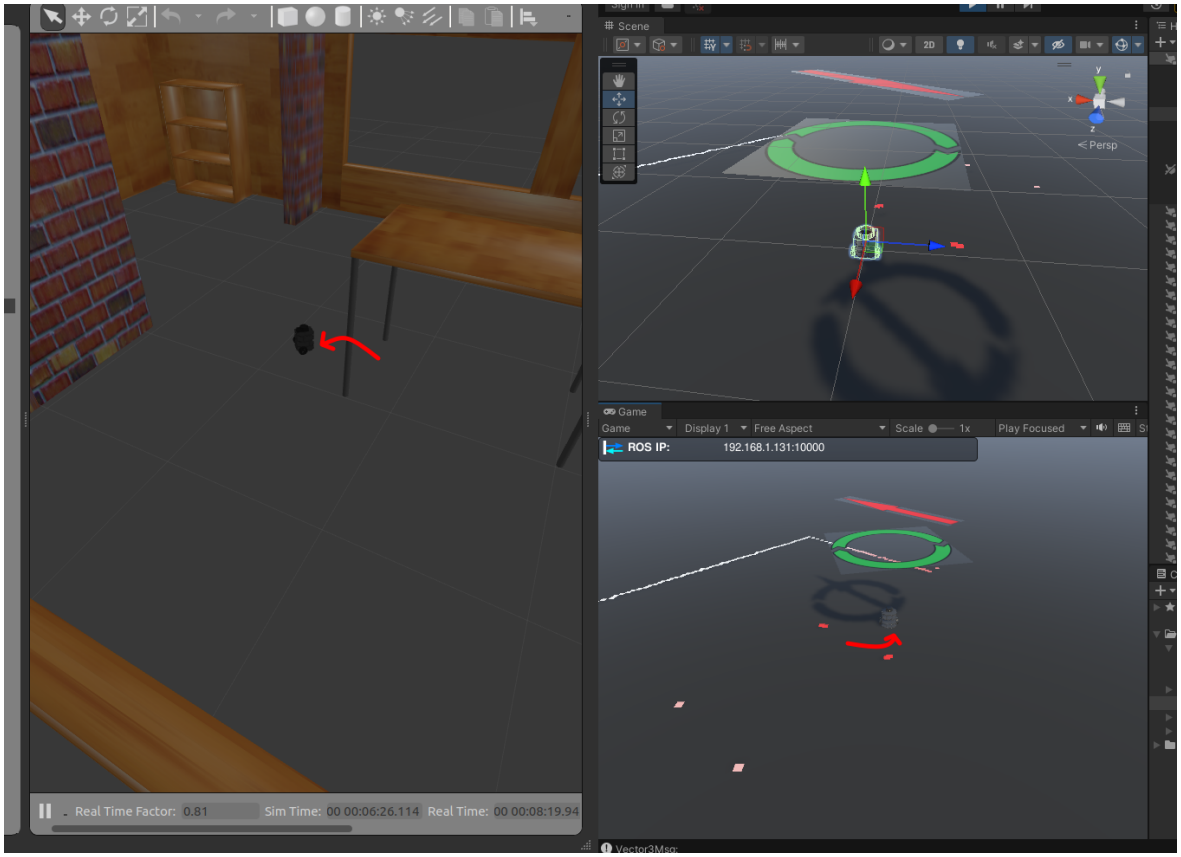


Figura 34: Desplazamiento y mapeo, robot fijo centrado.

Como puntualización, se aprecia como las patas de la mesa son rojas por estar el robot más cerca de ellas, como se establece en la parte de la marca de calor por cercanía del script.

ESTA será la solución escogida por los motivos sugeridos posteriormente. El video demostrativo no de la simulación, sino del experimento real, está adjuntada en el repositorio de git, en el ANEXO final.

5.7.3. Solución 2: Robot móvil + Mapa fijo

TFMessageMsg

En la siguiente propuesta se sugiere un script ObstacleFinderV2.cs que va a tener el añadido de manejar la transformada del robot para que este pueda desplazarse.

Obviando redundar en lo ya comentado en el apartado anterior sobre InputManager y los topics ya creados.

En este caso las marcas van a ser permanentes y por tanto se debe almacenar o bien sus coordenadas o bien el GameObject directamente para poder extraer dicha transformada como se verá a continuación.

En este caso el color no se transformará en el mismo script, sino en otro paralelo introducido mediante el GameObject ModificadorColor, que tiene

asociado un script de tipo ColorDistance y que realizará la gestión del color al tener este que cambiar mientras el robot se vaya desplazando.

```
[...]
public class ObstacleFinderV2 : MonoBehaviour
{
    // Parametros fisicos.
    public GameObject robot;
    public GameObject marcaPrefab; // El prefab del punto que
       sera generado para mostrar un obstaculo.
    // Para limitar la superposicion de marcas.
    private HashSet posicionesMarcas = new HashSet(); //
       Almacen de posiciones de las marcas creadas.
    public float separacionMarcas = 0.15f; // Distancia minima
       tolerada entre las marcas.

    // Gestion de los topics.
    public string topicName = /scan;
    private LaserScanMsg lastLaserScanMsg; // Almacen de la
       informacion del mensage del Lidar al que se suscribe.
    [SerializeField] private ROSConnection ros;
    public float publishMessagePeriod = 0.1f; // Publicacion
       del topic cada publishMessagePeriod segundos.
    private float timeElapsed; // Se usa para llevar un conteo
       del tiempo transcurrido.

    // Transmision de informacion al manejador del cambio de
       color.
    public ColorDistance colorChanger;

    void Awake()
    {
        ros = ROSConnection.GetOrCreateInstance();
        ros.Subscribe<LaserScanMsg>(topicName, LIDARRobot);
    }
```

En este caso, el cálculo de las marcas se establece mediante las coordenadas relativas al robot, de manera que al girar este y desplazarse, las coordenadas no se escriban sin más, sino que tengan en cuenta que la referencia real (el robot) cambia respecto del mundo.

Así tras un proceso matemático de transformadas y coordenadas se puede obtener la coordenada de cada punto relativa al robot e instanciar la marca.

```
    void LateUpdate()
    {
        timeElapsed += Time.deltaTime;
        if (timeElapsed > publishMessagePeriod &&
           lastLaserScanMsg != null)
        {
```

```
float[] disparos = lastLaserScanMsg.ranges; //
    Array de distancias producido por el Lidar.
float incrementoAngular =
    lastLaserScanMsg.angle_increment; // Angulo entre
    disparos del Lidar.
Quaternion localRotation = robot.transform.rotation;
```

Ya que el propio robot transmite la onformación real del mismo tal como los ángulos de apertura de los rayos láser, no habrá mejor indicador que el mismo.

El cuaternión localRotation permite saber la orientación del robot en cada momento, pues el primero de los rayos del Lidar será en el ángulo 0 en terminos relativos al robot. En términos absolutos será el ángulo 0 más el ángulo del robot respecto del mundo, y ese será el valor que ha de tenerse en cuenta. Este es un proceso similar al empleado para el cálculo de las posiciones de las marcas, que también se ajustan a la posición del robot, pues los valores del Lidar son distancias respecto del mismo, no respecto del origen del mundo.

```
Vector3 worldDirection =
    transform.TransformDirection(localRotation *
    Vector3.forward);
float anguloRadianes =
    -robot.transform.rotation.eulerAngles.y *
    Mathf.Deg2Rad; // Paso a radianes de la
    orientacion inicial.
//float anguloRadianes = worldDirection.y *
    Mathf.Deg2Rad; // Paso a radianes de la
    orientacion inicial.
```

Se transforman las coordenadas relativas en absolutas;

```
//float anguloRadianes = worldDirection.y *
    Mathf.Deg2Rad -
    robot.transform.rotation.eulerAngles.y *
    Mathf.Deg2Rad; // Paso a radianes de la
    orientacion inicial.
```

Para finalizar con el recorrido del array de disparos, para obtener las coordenadas unitarias de cada disparo, extreaer las trigonométricas que serán proporcionales a las coordenadas de cada disparo para posteriormente multiplicar la distancia de cada coordenada y así poder instanciar las marcas en dichas coordenadas.

```
foreach (float distancia in disparos) // Para cada
    uno de los disparos...
{
     if (!float.IsInfinity(distancia)) // Se filtran
         las distancias que sean infinitas.
     {
```

```
                    float xUnit =
                        (float)Math.Cos(anguloRadianes);
                    float yUnit =
                        (float)Math.Sin(anguloRadianes);
                    Vector3 posicionMarca =
                        robot.transform.position - new
                        Vector3(xUnit * distancia, 0, yUnit *
                        distancia); // Posicion cartesiana.
                    CreaMarca(posicionMarca);
                }
                anguloRadianes += incrementoAngular;
            }
            timeElapsed = 0; // Reinicia el conteo hasta el
                siguiente tick.
        }
    }
```

Código similar al caso anterior.

```
    // Actualizar valores del Lidar.
    void LIDARRobot(LaserScanMsg robotScan)
    {
        lastLaserScanMsg = robotScan;
        colorChanger.constructor(lastLaserScanMsg.range_max,
            lastLaserScanMsg.range_min); // Indica al coloreador
            de marcas las distancias extremas.
    }
```

Recogiendo el listado de marcas (y rellenandolo al instanciarlas), no solo se deben almacenar los datos de las marcas, sino entregarselo al script que les cambiará el color.

```
    // Instancia una nueva marca de obstaculo en el mapa.
    void CreaMarca(Vector3 posicion) // Genera una marca en el
        mapa en caso de que no haya una cerca de donde se
        requiera.
    {
        if (!hayMarcaCerca(posicion)) // No pondra una nueva
            marca en caso de haber otra cerca.
        {
            GameObject marca = Instantiate(marcaPrefab,
                posicion, Quaternion.identity);
            posicionesMarcas.Add(posicion); // Posicion de la
                nueva marca a la lista de marcas.
            colorChanger.nuevaMarcaCreada(marca); // Indica al
                coloreador de marcas una nueva marca creada.
        }
    }
```

Un breve método que indica mediante operación interna muy optimizada la separación entre dos objetos marca.

```
    // Comprueba que no haya otras marcas cerca de la marca
       indicada.
    bool hayMarcaCerca(Vector3 nuevaMarca)
    {
        // Revisara todos los macadores para comprobar que no
           esten demasiado cerca.
        foreach (Vector3 otraMarca in posicionesMarcas)
        {
            if (Vector3.Distance(nuevaMarca, otraMarca) <
               separacionMarcas)
                return true; // Si encuentra un obstaculo lo
                   indica y se sale.
        }
        return false; // Si no encontro ninguno lo indicara.
    }
}
```

Para la ejecución del script se necesitaron los siguientes objetos;

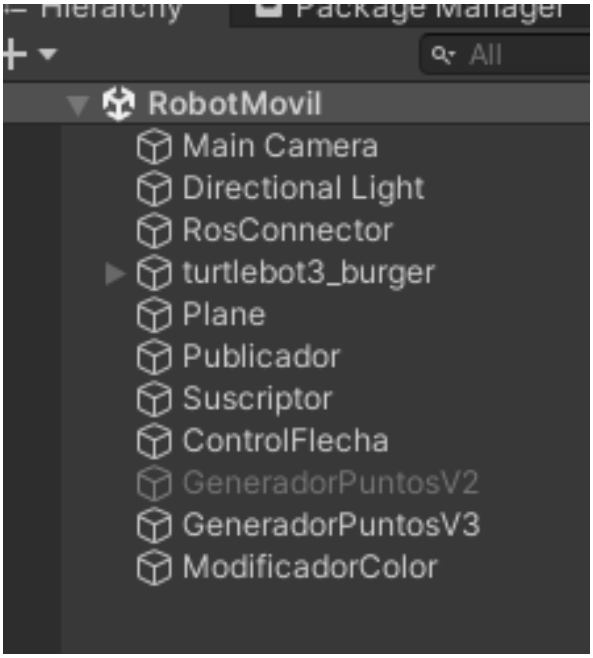


Figura 35: Todos los objetos del caso de estudio.

En realidad el script posee dos versiones, donde en una se emplea la posición transformada a partir del mismo topic y en la otra se emplea la posición relativa al modelo URDF. Esta posición no tiene que ser real, sino la de la proyección en la simulación con el objetivo de instanciar los puntos en una posición relativa al robot, pues este al moverse, cambiaría el origen del disparo alterando la toma del punto.

Para la ejecución del script se necesitaron los siguientes objetos;

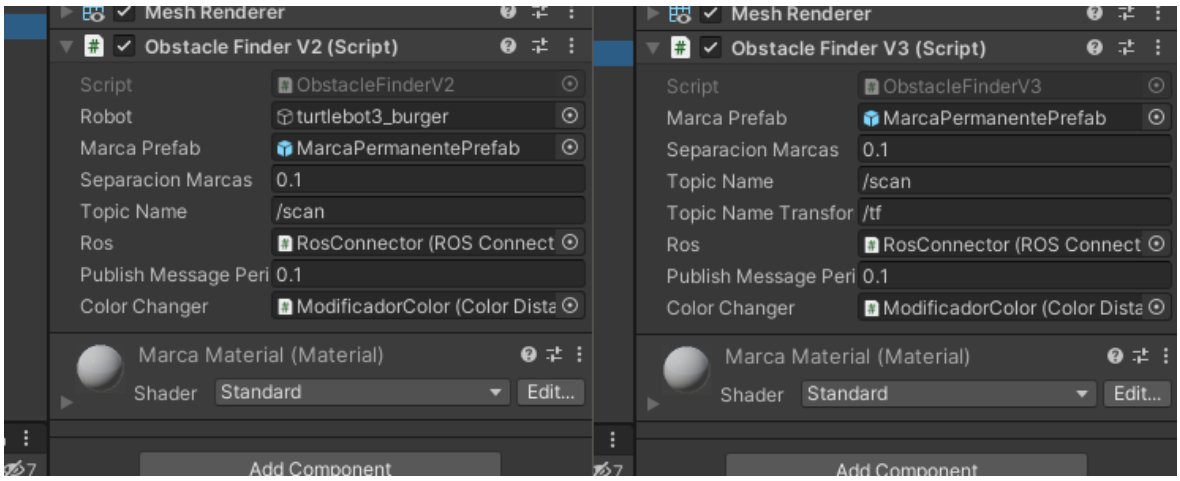


Figura 36: Variables de trabajo de los dos scripts.

5.7.3.1 Color en función de distancia

De nuevo, con un gradiente declarado externo al script, se indexará un color en función de la distancia del objeto.

```
[...]
public class ColorDistance : MonoBehaviour
{
    public GameObject robot;
    private List marcasCreadas = new List();
    public float changePeriod = 0.1f; // Publicacion del topic
       cada publishMessagePeriod segundos.
    private float timeElapsed; // Se usa para llevar un conteo
       del tiempo transcurrido.
    private float rangoMax; // Rango entre el alcance del Lidar
       y el minimo detectado de distancia (aportado por proceso
       padre).
    private Color[] rangoColoresMarca; // Array de colores
       precalculado en base a la distancia.
    public Gradient gradienteColor;

    private void Awake()
    {
        float distNormal;
        rangoColoresMarca = new Color[100]; // Por ejemplo un
           rango entre 0 y 100.
        for (int i = 0; i < rangoColoresMarca.Length; i++)
        {
            distNormal = i / 100f;
            rangoColoresMarca[i] =
               gradienteColor.Evaluate(distNormal); // Colores
               precalculados en el gradiente.
        }
```

```
}
```

Teniendo eso bastaría con modificar el color de cada marca cada changePeriod tiempo.

```
private void Update()
{
    timeElapsed += Time.deltaTime;
    if (timeElapsed > changePeriod)
    {
        ChangeColors();
        timeElapsed = 0; // Reinicia el conteo hasta el
            siguiente tick.
    }
}
```

Pequeña función para actualizar las variables fisicas de distancia del Lidar y construir el array de gradientes de color.

```
public void constructor(float max, float min) {rangoMax =
    max - min;}
```

Y esta otra pequeña función permite a componentes externos, tales como el script anterior, rellenar de objetos marca la lista de marcas.

```
public void nuevaMarcaCreada(GameObject marcaNueva)
    {marcasCreadas.Add(marcaNueva);}
```

Y así, este método recorre el array de marcas modificando su variable Renderer.color para ajustarlas a la distancia del robot.

```
private void ChangeColors()
{
    foreach (GameObject marca in marcasCreadas)
    {
        float dist =
            Vector3.Distance(robot.transform.position,
            marca.transform.position);
        // Colorear la marca del color correspondiente:
        // Calcula la distancia y la normaliza entre 0 y 1.
        float distNormal = Mathf.Clamp01(dist / rangoMax);
        // En funcion de la distancia indica el indice del
            tono de color que debe buscar en el array.
        int colorIndex = Mathf.FloorToInt(distNormal *
            (rangoColoresMarca.Length - 1));
        Renderer renderer;
        if (marca.TryGetComponent(out renderer))
        {
            renderer.material.color =
                rangoColoresMarca[colorIndex]; // Establece
                el tono que le corresponde segun el indice.
        }
    }
```

```
    }
}
```

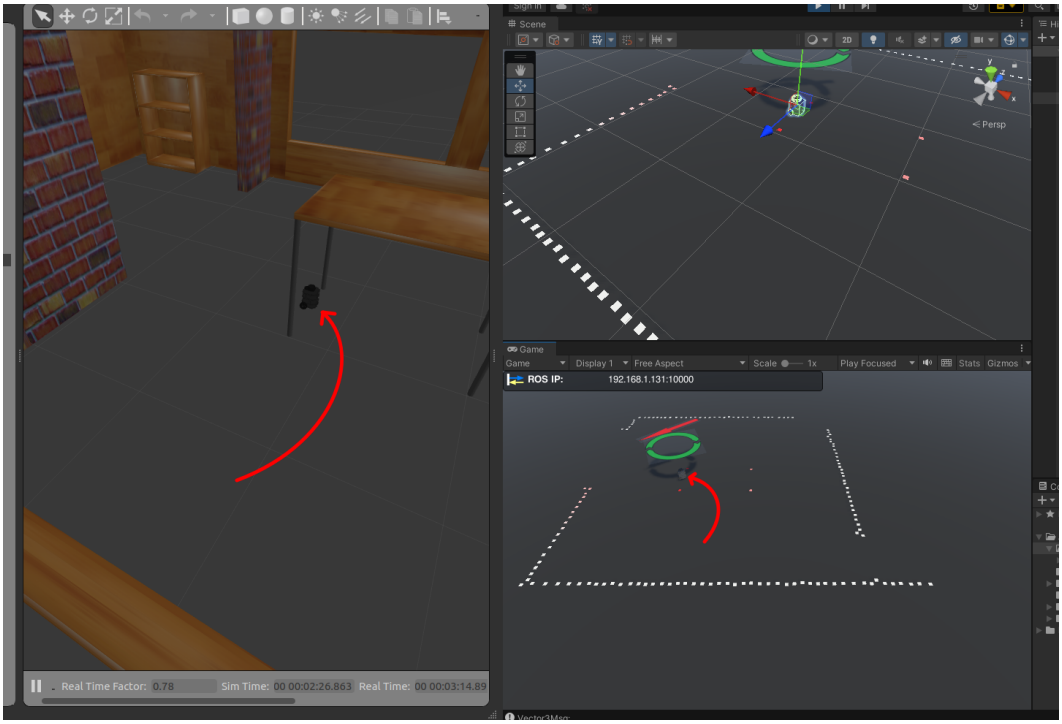


Figura 38: Desplazamiento y mapeo, robot móvil.

Al igual que en el caso anterior, se aprecia como se enciende el tono rojo de la pared a medida que el robot se acerca a los puntos de manera dinámica y continua. Ya no porque se generen los puntos más rojos, sino porque los ya generados van modificando su valor en función de la cercanía.

5.8. RESULTADOS

En el repositorio anexado se entregan dos simulaciones y una experimentación empleando la primera solución tambien con un experiemnto con el Turtlebot3 real y en el caso de la otra solución, únicamente la simulación.

En el desarrollo del proyecto, se implementó una estrategia clásica pero práctica, simple y directa para la adquisición de puntos en un mapa utilizando tecnología LiDAR (Light Detection and Ranging). En este contexto, la elección de la opción de muestreo continuo alrededor del robot ha sido fundamental para alcanzar los objetivos de manera eficiente.

En primer lugar, es crucial comprender la importancia del LiDAR, un sistema que utiliza pulsos láser para medir distancias y crear mapas tridimensionales del entorno circundante. En lugar de optar por un enfoque estático convencional, donde el LiDAR se mantiene inmóvil mientras escanea su entorno, se decidió implementar una técnica más dinámica y muestreo dinámico.

En este proyecto, el robot se encuentra en la primera propuesta en el mapa SIMULADO en una posición fija mientras que el LiDAR realiza un muestreo continuo alrededor de la ubicación del robot REAL. Esta elección estratégica tiene diversas ventajas. En primer lugar, permite una cobertura más completa del entorno, ya que el LiDAR puede capturar información desde todos los ángulos posibles alrededor del robot. Esto resulta fundamental para obtener un mapeo tridimensional preciso y detallado, otra ventaja es que tal como está dispuesto el código, los puntos se disipan inmediatamente, permitiendo la actualización contínua del mapa.

Además, la opción de muestreo continuo minimiza la posibilidad de errores asociados con el movimiento del robot durante la adquisición de datos, asegurando así una mayor precisión en el modelado del entorno, pues obstáculos temporales tales como personas o cualquier interrupción puede ser MALINTERPRETADO como un valor significativo de terreno.

Cabe destacar que, durante el proceso de diseño, también consideré la opción alternativa de permitir que el LiDAR SIMULADO se mueva mientras el entorno SIMULADO permanece estático en un mapa fijo. Sin embargo, la elección de mantener el robot en posición fija demostró ser igual de adecuada para las especificaciones del proyecto y la segunda opción queda pues como una potencial alternativa, garantizando una captura de datos más efectiva y una generación de mapas tridimensionales menos robusta por no poder OLVIDAR el ruido.

6. ODS - Objetivos de desarrollo sostenible

Este proyecto se centra en gran medida en la creación de una herramienta para el análisis de la viabilidad del control manual en el mapeo y detección de entornos, utilizando herramientas como Unity y ROS, junto con los recursos proporcionados por el Turtlebot3. Aquí hay algunas consideraciones y posibles conexiones con los Objetivos de Desarrollo Sostenible (ODS):

1. Acceso a la Tecnología y Educación de Calidad (ODS 4 - Educación de Calidad): Al buscar crear una herramienta que sea viable para usuarios no familiarizados y que permita la familiarización con herramientas de programación y motor gráfico, tu proyecto podría contribuir al objetivo de proporcionar educación de calidad y facilitar el acceso a la tecnología.

2. Desarrollo de Herramientas para la Innovación (ODS 9 - Industria, Innovación e Infraestructura): Al explorar y desarrollar nuevas formas de control y detección de entornos utilizando Unity, ROS y Turtlebot3, estás contribuyendo a la innovación tecnológica, que es un componente clave del ODS 9.

3. Interfaz Intuitiva para Usuarios Poco Cualificados (Posiblemente ODS 10 - Reducción de Desigualdades): Al diseñar una interfaz de control simple, dinámica e intuitiva, podrías contribuir indirectamente a la reducción de desigualdades, permitiendo que personas con diversos niveles de habilidades interactúen con la tecnología de manera más efectiva.

4. Visualización Explicativa (Posiblemente ODS 11 - Ciudades y Comunidades Sostenibles): La creación de un producto visual y explicativo podría ser relevante para la comunicación de información sobre entornos y podría ser útil en contextos relacionados con la planificación urbana y la sostenibilidad (ODS 11).

5. Optimización y Exportación de Modelos (Posiblemente ODS 12 - Producción y Consumo Responsables): Si tu proyecto incluye herramientas para optimizar y exportar modelos, esto podría relacionarse con el objetivo de producción y consumo responsables al reducir el desperdicio de recursos en el desarrollo de modelos.

El estudio de este proyecto puede tener impactos positivos en términos sociales, económicos y ambientales, y cómo puede contribuir al desarrollo sostenible y la mejora de la calidad de vida. Estas conexiones son meras sugerencias y dependen de la continuación y la entrada en detalle del proyecto.

7. Conclusiones

Para el logro del objetivo del proyecto, la elección de la primera solución se revela como la más idónea, dado su alineamiento con la naturaleza temporal de las marcas, que permanecen inmunes a objetos transitorios y terrenos cambiantes. La implementación del motor de simulación Unity para la operación remota de un robot móvil no solo ha demostrado ser exitosa, sino que ha arrojado resultados que permiten una accesibilidad sencilla a través de la red, marcando así el inicio de un proceso hacia un mapeo avanzado de terrenos.

- La elección acertada del motor Unity se fundamenta en la eficacia de sus paquetes y la programación en C#, junto con el entorno Visual Studio, facilitando la programación del entorno simulado de manera ágil y accesible. Esta solución se destaca por su bajo costo, siendo compatible incluso con dispositivos móviles y aquellos de menor capacidad de procesamiento y memoria.

- Cabe resaltar que la segunda solución se ve afectada por el ruido del sensor, ya que la simulación representa un escenario ideal y poco confiable como herramienta de verificación.

- En cuanto a las herramientas de operación del robot, la simplicidad reina, basándose en dos controles de avance y giro (conocidos como controles de tanque), respaldados por el paquete Unity de control de inputs, que demuestra ser lo suficientemente simple y eficaz para asegurar un rendimiento óptimo en el control del usuario común.

- Los marcadores e indicaciones de distancia, movimiento y desplazamiento han sido diseñados de manera explicativa, proporcionando al usuario una comprensión clara del estado cinemático del robot y su posición con respecto a obstáculos.

En resumen, este proyecto ha implicado la ejecución de múltiples tareas y la resolución de problemas mediante el uso de diversas herramientas, marcando un aprendizaje significativo en la implementación de soluciones tecnológicas.

8. Propuestas a Futuro

Tras exponer las conclusiones, se vislumbran diversas líneas de mejora e investigación que podrían potenciar el proyecto:

1. Aplicación de Realidad Aumentada (AR): Explorar la integración de hololens para el control remoto del robot mediante un mando de Xbox, aprovechando las capacidades de AR para mejorar la experiencia de usuario.

2. Implementación de la Segunda Solución en un Mapa Fijo: Probar la aplicabilidad de la segunda solución del robot móvil simulado en Unity en un entorno con mapa fijo, evaluando su desempeño en condiciones más controladas.

3. Aplicación de Solución SLAM: Investigar y aplicar soluciones SLAM (Simultaneous Localization and Mapping) para garantizar un mapeo preciso a pesar del ruido y las interrupciones del LIDAR, mejorando la robustez del sistema.

4. Validación con Turtlebot3 Waffle: Realizar una aplicación de prueba con Turtlebot3 Waffle y verificar la consistencia en la simulación entre ambos casos, asegurando coherencia en el manejo de los topics en ROS.

5. Exploración de Motores Alternativos: Experimentar con motores alternativos como Unreal Engine en C++, evaluando su velocidad y opciones gráficas superiores en comparación con Unity.

6. Interfaz Gráfica Avanzada: Desarrollar una interfaz gráfica más avanzada que proporcione al usuario información intuitiva sobre la dirección, sentido, velocidad y otras variables secundarias del robot, reduciendo la necesidad de mirar directamente al robot para obtener esta información.

9. ANEXO - Enlace a código y vídeo en Github

https://github.com/fguedel/TELEOPERACI-N-DE-ROBOTS-M-VILES-BASADA-EN-UNITY.git

Referencias

[CDFS+14] R. Codd-Downey, P. Mojiri Forooshani, A. Speers, H. Wang, and M. Jenkin. From ros to unity: leveraging robot and virtual environment middleware for immersive teleoperation. Julio 2014.

[Fra21] Lena Sophie Franke. Una breve historia de la robótica – robots modernos, 1 2021.

[HGOM18] Ahmed Hussein, Fernando García, and Cristina Olaverri-Monreal. Mobile delivery robots: Mixed reality-based simulation relying on ros and unity 3d. Septiembre 2018.

[LNS+20] Yuzhou Liu, Georg Novotny, Nikita Smirnov, Walter Morales-Alvarez, and Cristina Olaverri-Monreal. Mobile delivery robots: Mixed reality-based simulation relying on ros and unity 3d. Octubre 2020.

[NS01] Yoshihiko Nakamura and Akinori Sekiguchi. The chaotic mobile robot. *IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION*, 17(6), Diciembre 2001.

[Wat98] Keigo Watanabe. Control of an omnidirectional mobile robot. Abril 1998.

[yZmLxC18] Han ye Zhang, Wei ming Lin, and Ai xia Chen. Path planning for the mobile robot: A review. *Symmetry*, 10:450, Octubre 2018.