Escuela
Politécnica
Superior

Interfaz inmersiva de realidad virtual para la teleoperación de robots móviles.

Grado en Ingeniería Robótica

Trabajo Fin de Grado

Autor:
Manuel Tovar Martínez
Tutor/es:
Miguel A. Cazorla Quevedo
Félix Escalona Moncholí

Julio 2021

Universitat d'Alacant
Universidad de Alicante

Interfaz inmersiva de realidad virtual para la teleoperación de robots móviles.

Autor
Manuel Tovar Martínez

Tutor/es
Miguel A. Cazorla Quevedo
Departamento de Ciencia de la Computación e Inteligencia Artificial
Félix Escalona Moncholí
Departamento de Ciencia de la Computación e Inteligencia Artificial

Grado en Ingeniería Robótica

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2021

Preámbulo

"La fusión de los mundos real y virtual es uno de los aspectos tecnológicos que más interesantes me parecen, es una manera de continuar evolucionando como especie, añadiendo nuevas capacidades a nuestros propios sentidos. La principal razón de este trabajo es esa, investigar en este ámbito y contribuir a su desarrollo. Su finalidad está más enfocada a la teleoperación, dotar a los operadores de nuevas técnicas que permitan trasladarlos al lugar de trabajo del robot de manera inmersiva."

Agradecimientos

En primer lugar, tengo que agradecer a mis tutores Miguel A. Cazorla y Félix Escalona su disposición para brindarme ayuda en cualquier problema, así como proveerme de las gafas de realidad virtual, elemento imprescindible en la realización del proyecto. En relación a esto, también agradecer a mi amigo Aarón su colaboración, prestándome el ordenador que ha sido la base del sistema esclavo, y que sin él no habría sido posible el diseño de esta interfaz de teleoperación. Por último, no me puedo olvidar de mis amigos y familia, que sin su apoyo, favores, cariño y risas, no sólo habría sido imposible la realización de este trabajo, si no el logro de superar la carrera.

"Todo lo que llamamos real
está compuesto por cosas
que no pueden considerarse
como reales"

Niels Bohr.

Índice general

Índice de figuras

Índice de tablas

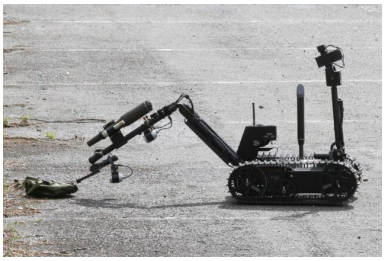
Índice de Códigos

1 Introducción

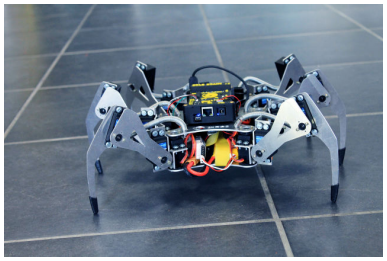
La Telerrobótica es el área de la robótica que estudia todo lo relacionado con el control de robots a distancia, ya sea una distancia física real o una diferencia de escala entre operador y máquina. En ella se combinan dos importantes campos, teleoperación y telepresencia.

Mientras que la teleoperación es la encargada de gobernar al robot en la realización de un trabajo remoto, la telepresencia busca crear una conexión entre el operador y el entorno en el que se encuentra el robot, haciendo que el primero sienta que está en ese lugar. En este trabajo se busca unificar ambos ámbitos destacando en el de telepresencia, ya que se pretende trasladar el mundo percibido por los sensores del robot a un entorno completamente inmersivo para el usuario, como es el de la realidad virtual.

Cabe destacar que uno de los grandes motivadores del desarrollo tecnológico siempre ha sido garantizar la seguridad de los humanos en tareas potencialmente peligrosas, ya sea por la manipulación de piezas peligrosas, el trabajo en entornos hostiles o las lesiones provocadas por la falta de ergonomía. La robótica tiene un gran potencial en este sentido, ya que permite la realización de este tipo de trabajos evitando la presencia de las personas directamente en el entorno. Por este motivo, se desarrollan sistemas robóticos, tanto totalmente autónomos como tele-operados como el de la figura 1.1a.



(a) Robot EOD para la desactivación de artefactos explosivos. (*Atria Innovation*, 2020)



(b) Robot Erle-Spider para labores de inspección en espacios reducidos. (*CincoDias*, 2015)

Figura 1.1: Robots para tareas en entornos peligrosos o inaccesibles.

Por otra parte, la ciencia lleva tiempo intentando fabricar pequeños robots capaces de adentrarse en entornos inaccesibles para los humanos. Estos dispositivos permiten desarrollar tareas de reparación y mantenimiento en plantas de extracción de gas y petróleo, centrales nucleares o grandes obras de ingeniería civil que, por lo intrincado de su diseño, requieren de un mantenimiento en lugares inaccesibles para las personas (figura 1.1b). Otra de las principales aplicaciones de estos robots a pequeña escala es la robótica médica ya que permiten

realizar exámenes médicos y operaciones quirúrgicas con gran precisión.

Las características propias de las tecnologías inmersivas, aportan un enorme potencial para mejorar la teleoperación de robots. En este proyecto se ha desarrollado una interfaz mediante realidad virtual que consigue conectar el mundo real percibido por robots, con el mundo virtual creado en un ordenador. Para ello se ha seguido la estructura indicada en la figura 1.2. Mediante un ordenador con sistema operativo Linux se controla al robot que funciona bajo Robot Operating System (ROS), permitiendo así la lectura y escritura de datos mediante topics, (en este caso se simula un entorno real usando Gazebo7 y el robot Turtlebot2). Este PC se comunicará con otro terminal con Windows10 mediante sockets Transmission Control Protocol - Internet Protocol (TCP-IP) ejecutados desde el software para la programación de videojuegos Unity, y utilizando el paquete de ROS Rosbridge junto con la librería de ROS para Unity, RosSharp. Por último, se hará el tratamiento necesario de los datos tanto en Unity como en Linux para crear el mundo virtual y se mostrará en las gafas Virtual Reality (VR) Oculus Quest 2, conectadas a Unity mediante el Plug-In nativo Extended Reality (XR) Plug-In.

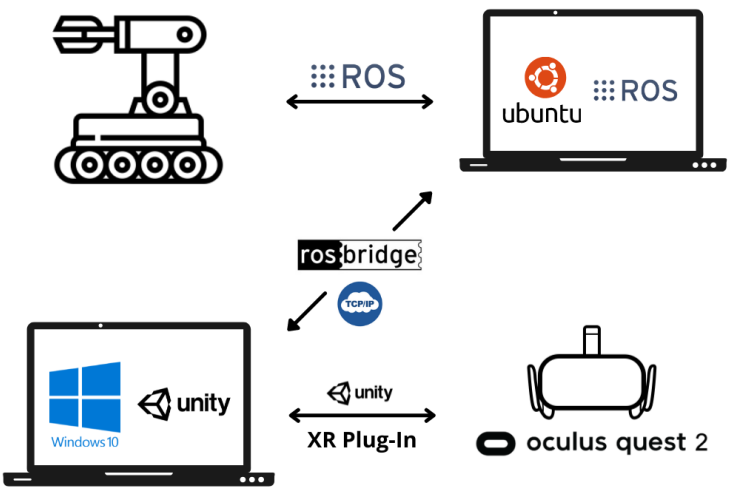


Figura 1.2: Arquitectura del sistema.

En los siguientes apartados se explicarán detalladamente los procedimientos llevados a cabo para realizar esta interfaz, de qué manera se traslada el mundo real al virtual y cómo este puede ser utilizado por el operario para controlar al robot y percibir información valiosa del entorno.

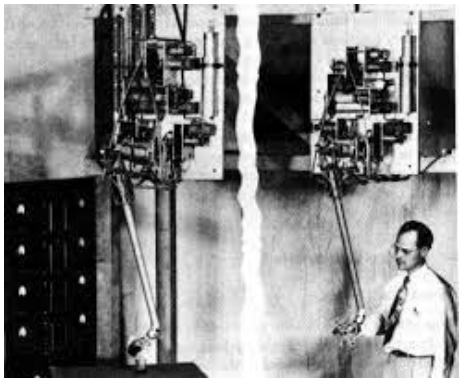
2 Marco teórico

2.1 Contexto histórico

Los inicios de la teleoperación se remontan al 1947, cuando Raymond Goertz del Argonne National Laboratory de Estados Unidos lideró las primeras investigaciones en este ámbito. Estas eran motivadas por la industria nuclear y tenían como objetivo la manipulación a distancia de elementos radioactivos.

En 1948 se obtuvieron los primeros frutos con el desarrollo del primer manipulador maestro-esclavo teleoperado, denominado M1 (figura 2.1a). Este sistema era enteramente mecánico, de manera que los movimientos realizados en el maestro se transmitían eje a eje al manipulador esclavo. En los siguientes años, Goertz continuó con las investigaciones en busca de mejorar este primer sistema de teleoperación de manera que fuera accionado por motores en lugar de manualmente. Finalmente, en 1954 presentó el primer manipulador maestro-esclavo con accionamiento eléctrico y servo-control, llamado E1 (figura 2.1b), (Basañez y Suárez, 2009), (*Raymond Goertz*, s.f.).

(a) M1. Primer manipulador maestro-esclavo.



(b) E1. Primer manipulador maestro-esclavo eléctrico.

Figura 2.1: Raymond Goertz manipulando elementos radioactivos con ayuda de los primeros sistemas teleoperados (Basañez y Suárez, 2009).

A partir de ese momento, esta nueva tecnología se dio a conocer y fue investigada y desarrollada por otros laboratorios y empresas. Así, en los años sesenta aparecieron máquinas diseñadas para aplicaciones submarinas con sistemas cada vez más avanzados, debido a la inclusión de cámaras y otros sensores que mejoraban la sensación de telepresencia.

Posteriormente, en los años setenta, la teleoperación superó sus primeras etapas de desarrollo gracias a su utilización en el control de los vehículos espaciales Lunojod I y Lunojod II que fueron enviados a la Luna en 1970 y 1973, respectivamente (Ballesteros, 2017).

En las últimas décadas se ha dado una evolución paralela a la teleoperación en ámbitos como la informática, las comunicaciones o la robótica. Esto ha posibilitado un incremento en las capacidades del sistema remoto, mejorando fundamentalmente las prestaciones de la interfaz hombre-máquina.

2.2 Teleoperación

La teleoperación indica el control de un sistema o máquina a distancia. En la figura 2.2 se puede ver el funcionamiento de los sistemas de teleoperación maestro-esclavo y los elementos que lo componen. A continuación se explicará cada uno de ellos (Montalvo y cols., 2019), (Ballesteros, 2017).

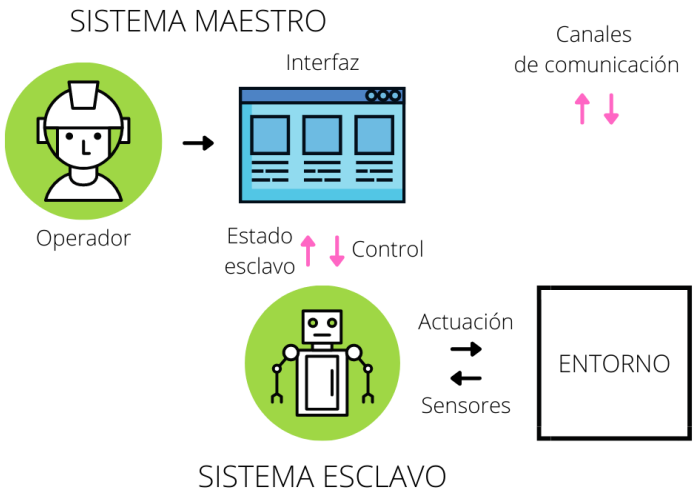


Figura 2.2: Arquitectura de los sistemas de teleoperación.

Operador . Ser humano que hace uso de la interfaz para realizar el control de la operación a distancia. Este control puede ser total o limitarse a marcar objetivos que el esclavo debe cumplir.

Interfaz . Medio que permite la interacción del operador con el sistema de operación. Se refiere tanto a los instrumentos que permiten que el operador lleve a cabo el control, como a los componentes que ofrecen al operador información sobre el estado de la teleoperación.

En este trabajo se va a diseñar una interfaz de teleoperación por lo que en el siguiente

apartado profundizaremos más sobre sus tipos y utilidades (2.3).

Canales de comunicación . Medio por el que se transmite la información en ambos sentidos entre la interfaz y el sistema esclavo. Puede ser mediante cables o de manera inalámbrica. En este proyecto se realiza mediante wifi.

Control . Módulo que se encarga de procesar las señales dadas por el operador y convertirlas en acciones efectivas que el esclavo pueda realizar.

Dispositivo teleoperado o esclavo . Máquina que trabaja en el sitio remoto y está siendo controlada por el operador. Puede ser un manipulador, un vehículo, un robot o un dispositivo similar.

Entorno . Lugar remoto en el que actúa el dispositivo teleoperado. En él realiza las tareas encomendadas por el operador y recoge información mediante los sensores, que será mostrada por la interfaz y utilizada por el operador y el control.

2.3 Interfaces de teleoperación

Las interfaces son uno de los elementos fundamentales de los sistemas de teleoperación, en ellas se realizan labores imprescindibles como la recepción de información y la transmisión de comandos entre los sistemas maestro y esclavo. Es de gran importancia el diseño de la interfaz de manera que su uso por el operador sea óptimo. La información mostrada debe ser clara y los comandos de control sencillos e intuitivos.

Pueden dividirse en cuatro categorías fundamentales: convencionales, multimodales, adaptativas e inmersivas. A continuación, se va a realizar un breve estudio de los diferentes tipos de interfaces de teleoperación y una comparativa de estos.

Interfaces convencionales: en la actualidad, existen numerosos dispositivos que pueden convertirse en interfaces de teleoperación: dispositivos móviles como smartphones o tablets, paneles de control con botones, joysticks, etc. Una interfaz de teleoperación convencional se puede definir como aquella que se compone de los elementos básicos para el control de un robot a distancia: información sensorial de los robots, órdenes y comandos mandados a los robots, estado actual de los robots y las tareas (batería, posibles fallos del sistema, etc.) y elementos para la navegación (Chen y cols., 2007).

Interfaces multimodales: la interacción multimodal hace referencia a la interacción con el entorno a través de modos de comunicación naturales, es decir, aquellos que implican los cinco sentidos humanos (vista, oído, olfato, gusto y tacto). Por ejemplo, interfaces en las que se añade información auditiva aumentando la percepción del entorno por el operador, o interfaces hápticas con realimentación de fuerzas en los que el operador siente el entorno con sus propias manos. En general, las interfaces multimodales permiten una comunicación más cómoda y natural (Bourguet, 2003), (Grifoni, 2009). En la figura 2.3 vemos un ejemplo de estas.

Figura 2.3: Una de las primeras interfaces de teleoperación hápticas, diseñada por Shadow Robot (*haptx.com*, s.f.).

Interfaces adaptativas: las interfaces adaptativas introducen algoritmos de minería de datos o de machine learning con el objetivo de ayudar o incluso sustituir al operador en tareas complicadas para este. En general, estas interfaces buscan extraer y proporcionar la información importante al humano para facilitarle la toma de decisiones. Son muy buenas en situaciones en las que la velocidad, la precisión o el grado de compresión de la tarea son características esenciales (Hansberger, 2015), (Félix Díez, 2016). En la figura 2.4 vemos de qué manera funcionan estas interfaces.

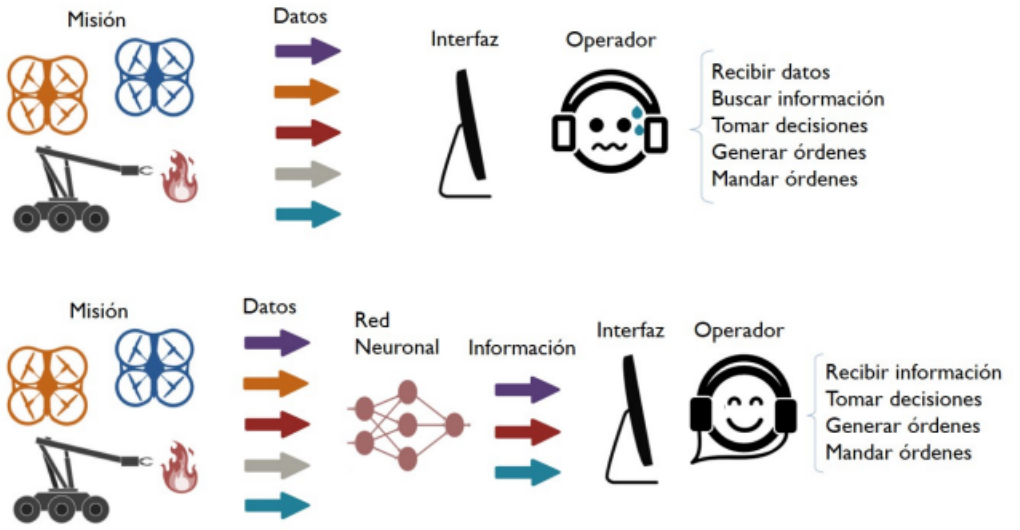


Figura 2.4: Mejoras que introducen las interfaces adaptativas en la teleperación (Tapia, 2017).

Interfaces inmersivas: el objetivo de este tipo de interfaces es aumentar la percepción del operador, dándole sensación de estar él mismo en el lugar remoto. Para esto se emplean

teconologías emergentes como cámaras 3D o gafas de realidad virtual y aumentada (Lin y cols., 2014). Existen tres tipos de tecnologías inmersivas: Virtual Reality (VR), que muestra el escenario real de manera virtual, Augmented Reality (AR), que superpone en vídeos del entorno elementos virtuales que aportan información, y Mixed Reality (MR) que combina las dos anteriores creando nuevos escenarios que combinan elementos reales y virtuales.

Por su propio diseño, las interfaces inmersivas aportan ventajas propias de las interfaces multimodales ya que introducen elementos virtuales que logran una recepción de la información más natural por parte del operador. En este trabajo se utilizan gafas de realidad virtual que sintetizan el entorno y lo muestran tan solo con usar la vista. De esta manera, se combinan elementos de las interfaces multimodales e inmersivas introduciendo numerosas ventajas sobre las interfaces convencionales. En la tabla 2.1 podemos ver como las interfaces adaptativas, multimodales y sobre todo inmersivas, solucionan muchos de los problemas que se dan en las interfaces convencionales.

Problema	**Tarea afectada**	**Posibles soluciones**
Campo visual limitado	Detección, guiado	Cámaras 360, interfaz inmersiva con multicámara
Falta de orientación	Navegación, guiado	Mapas, interfaz adaptativa con algoritmos de orientación, notables mejoras con interfaz inmersiva
Percepción no natural	Percepción del entorno, telemanipulación	Interfaz multimodal, interfaz inmersiva
Percepción contextual incorrecta	Telemanipulación	Múltiples cámaras, interfaz inmersiva con multicámara
Percepción de profundidad incorrecta	Telemanipulación, guiado	interfaz inmersiva VR
Imagen de vídeo degradada	Percepción del entorno	Interfaz con elevados fps
Delays en la comunicación	Percepción del entorno	Predicciones usando interfaces adaptativas

Tabla 2.1: Problemas de las interfaces convencionales, solucionados con el uso de interfaces adaptativas, multimodales e inmersivas (Chen y cols., 2007).

2.4 Interfaces VR

Las tecnologías inmersivas están aún en desarrollo por lo que la utilización de este tipo de interfaces es muy escaso y no hay demasiados proyectos funcionales en la actualidad. No obstante, existen varias empresas que están investigando el uso de gafas de realidad virtual para el control de robots. En general, se está aplicando en robots manipuladores ya que trabajan en entornos cerrados y el problema de trasladar el entorno real al mundo virtual se simplifica mucho (Gaschler y cols., 2014), (Fang y cols., 2014). Para establecer un contexto, en este apartado se van a mostrar algunos de los primeros sistemas que usaron VR, así como uno de los más avanzados hasta el momento.

Primeros sistemas VR: los primeros intentos de desarrollo utilizando esta tecnología emergente se encontraban con obstáculos como la falta de hardware especializado o la falta de sistemas de seguimiento ya implementados. Estos trabajos comenzaban a indagar en la teleoperación inmersiva y la traslación del entorno real al mundo virtual, pero se encontraban atados a sistemas hardware y librerías específicas. El soporte para los sistemas de seguimiento y otros imputs estaba limitado y era complicado manejar las distintas tecnologías VR ya que no estaba extendido el uso de las gafas de realidad virtual, por lo que debían diseñarse las estructuras de representación digital necesarias (Allard y cols., 2004), (Codd-Downey y cols., 2014). En la figura 2.5 vemos una de estas representaciones del "entorno virtual", se trata de un cubo de 6 caras con proyección de vídeo en cada una de ellas.

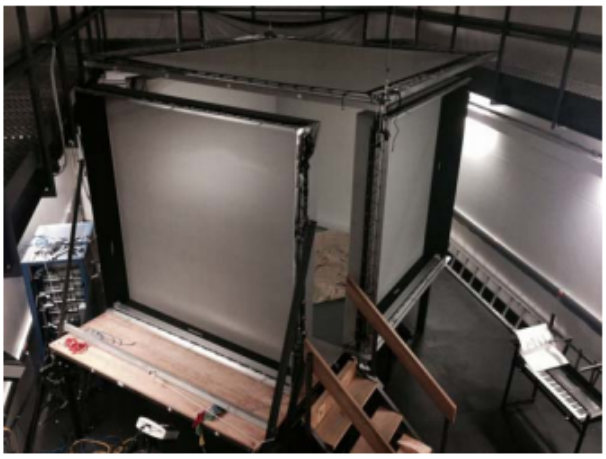
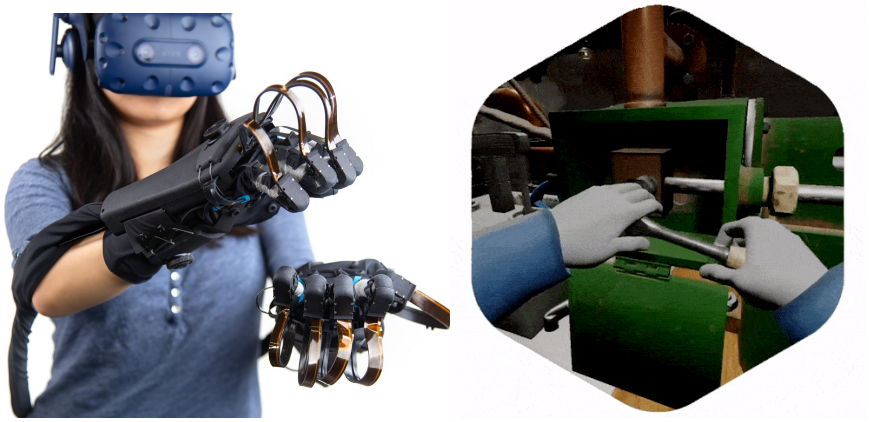


Figura 2.5: Una de las primeras representaciones de un "entorno inmersivo". Cubo con proyección de vídeo en cada una de sus 6 caras (Codd-Downey y cols., 2014).

Guantes Haptx DK2: estos guantes están diseñados para funcionar como interfaz háptica táctil, son usados para controlar las manos robóticas Shadow Robot mostradas en el apartado anterior (2.3), entre otras. Cuentan con 133 puntos de retroalimentación táctil en cada mano de manera que, según dice la marca, tu piel se desplaza de la misma forma que si tocases el objeto real. En los últimos años, han desarrollado la tecnología necesaria para introducir unas gafas de realidad virtual al sistema, consiguiendo imagen y audio

envolvente del entorno, además de realimentación táctil. Así, este sistema logra ser una de las interfaces inmersivas más avanzadas del mercado y está siendo usado para tareas de simulación de procesos, formación laboral y diseño industrial. Además está siendo integrado en empresas de todo el mundo para el control de robots a distancia, ya que permite la teleoperación de manos, pinzas y brazos robóticos, aportando información táctil crítica al operador (Needleman, 2018), (*haptx.com*, s.f.). El sistema Haptx se muestra en la figura 2.6.



(a) Sistema Haptx. **(b)** Visión del mundo virtual.

Figura 2.6: Sistema Haptx de guantes VR con realimentación visual y táctil (*haptx.com*, s.f.).

3 Objetivos y motivación

La realización de este trabajo partió del interés por las tecnologías inmersivas: realidad virtual, aumentada y mixta. Tecnologías emergentes que mediante elementos virtuales aportan información del mundo real que con nuestros cinco sentidos seríamos incapaces de percibir. Estas nuevas técnicas, combinadas con la teleoperación de robots, que permite que el humano realice un trabajo sin necesidad de encontrarse en el lugar del mismo, introducen numerosas mejoras que se han visto en el capítulo anterior (2.3) y son los aspectos iniciales que motivaron el desarrollo de este sistema. Además, como hemos visto, casi todas las interfaces inmersivas son usadas para el control de robots manipuladores ya que trabajan en entornos cerrados. Este proyecto se introduce en una nueva rama de investigación usando estas tecnologías para la teleoperación de robots móviles, abordando el problema del entorno cambiante, lo que aporta una motivación extra. Estas motivaciones, junto con otras relacionadas con la implementación y programación del sistema, nos llevan a buscar los objetivos mostrados en la siguiente lista.

- Objetivos generales.

  Mejorar en el uso de los lenguajes de programación Python y C++.

  Aprender sobre la programación en nuevos lenguajes como C# y ShaderLab.

  Aprender sobre la programación de videojuegos en Unity, enfocada a realidad virtual.

  Profundizar en el uso de ROS.

- Objetivos específicos.

  Idear e implementar el tratamiento necesario de los datos proporcionados por el robot, para trasladar el mundo real al virtual.

  Realizar un sistema de teleoperación capaz de operar un robot móvil a distancia.

  Implementar una interfaz de teleoperación inmersiva usando realidad virtual.

4 Metodología

En las siguientes líneas se van a comentar las herramientas y softwares utilizados en la realización de este sistema de teleoperación.

4.1 Turtlebot 2

Turtlebot es el robot estándar que funciona bajo la plataforma ROS. Se trata de un robot móvil diseñado principalmente para iniciarse en el mundo de la robótica y aprender sobre el uso de ROS. Es un dispositivo sencillo, pequeño, asequible y programable bajo código abierto (open source), al que pueden añadirse varios sensores, lo que lo hace una herramienta perfecta para educación, investigación o diseño de prototipos. Existen tres versiones de este robot. Turtlebot 1, desarrollado en 2010 por Tully (Gerente de Plataforma en Open Robotics) y Melonee (CEO de Fetch Robotics), basado en el robot Roomba de iRobot. Turtlebot 2, diseñado en 2012 por Yujin Robot basado en el robot de investigación, iClebo Kobuki. Y Turtlebot 3, desarrollado en 2017 como una mejora de su predecesor añadiendo nuevas características (*turtlebot.com*, s.f.).

En este caso se ha escogido el Turtlebot 2 ya que es el más utilizado y cuenta con una gran cantidad de código abierto, librerías y paquetes para su uso en ROS, y amplia información tanto oficial como de la comunidad. Trabaja sobre la base de investigación móvil Kobuki que cuenta con odometría de alta precisión y, esto, combinado con otros sensores que se verán a continuación, hace que sus puntos fuertes sean el SLAM y la navegación, por lo que es una muy buena opción para este trabajo.

Hardware: como se ve en la figura 4.1, el Turtlebot2 está formado por una base Kobuki a la que, mediante una serie de ”bandejas”, se le añaden distintas secciones en su parte superior. Así se consigue espacio para colocar sensores e incluso para sostener el ordenador que controlará al robot. En la tabla 4.1 aparecen sus especificaciones hardware y sensores más importantes.

En este trabajo se hará uso en mayor medida del sensor 3D Kinect, los datos extraídos de este junto con los aportados por otros sensores como el scanner láser son esenciales para el traslado del mundo real al virtual. También son imprescindibles los sensores de odometría para lograr el movimiento del robot o los bumpers para la detección de colisiones.

Software: al ser un robot basado en código abierto no necesita demasiado software dedicado,

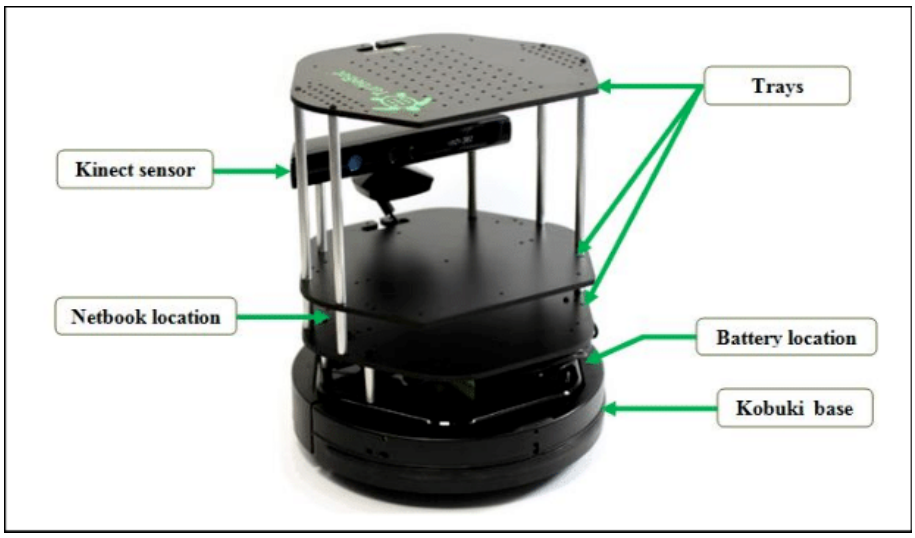


Figura 4.1: Turtlebot2 y sus componentes (*Components of Turtlebot 2*, s.f.).

Velocidad lineal máxima	70 cm/s
Velocidad angular máxima	180 º/s
Capacidad de carga	5 kg (suelo duro), 4 kg (alfombra)
Tamaño (L x W x H)	354 x 354 x 420 mm
Peso (+ SBC + Batería + Sensores)	6.3 kg
Odometría	25718.16 ticks/revolution, 11.7 ticks/mm
Gyro	calibración por defecto: 1 eje (100 º/s)
Bumpers	izquierdo, centro, derecho
Sensores de detección de pendientes	izquiero, centro, derecho
Sensor de caída de rueda	rueda izquierda y rueda derecha
Sensor 3D	Kinect / Orbbec Astra
Scanner Laser	Rplidar
Audio, LEDs y botones programables	sí
Batería	iones de litio, hasta 8800 mAh

Tabla 4.1: Especificaciones Hardware del Turtlebot 2 (*Turtlebot2 datasheet*, s.f.), (*roscomponents.com*, s.f.).

solo cuenta con los drivers Kobuki para ROS y para otros entornos en C++, y simulación del robot disponible en Gazebo. También existen algunas aplicaciones externas que pueden ser usadas con Kobuki y Turtlebot.

4.2 ROS

Robot Operating System es un framework o plataforma de trabajo diseñado para el desarrollo de software enfocado a la robótica. Apareció en 2007 para dar soporte al proyecto del

robot Stanford Artificial Intelligence Robot (STAIR), pero desde 2008 el desarrollo continúa primordialmente en el instituto de investigación robótico Willow Garage. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en distintos nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. Aunque la comunicación entre nodos puede darse mediante tres tipos de canales: topics, servicios y acciones, en el sistema solo se utilizan topics ya que pueden enviar mensajes sin necesidad de esperar respuesta del subscriptor, lo que los hace un mecanismo rápido y eficaz, ideal para la teleoperación de robots (*ros.org*, s.f.).

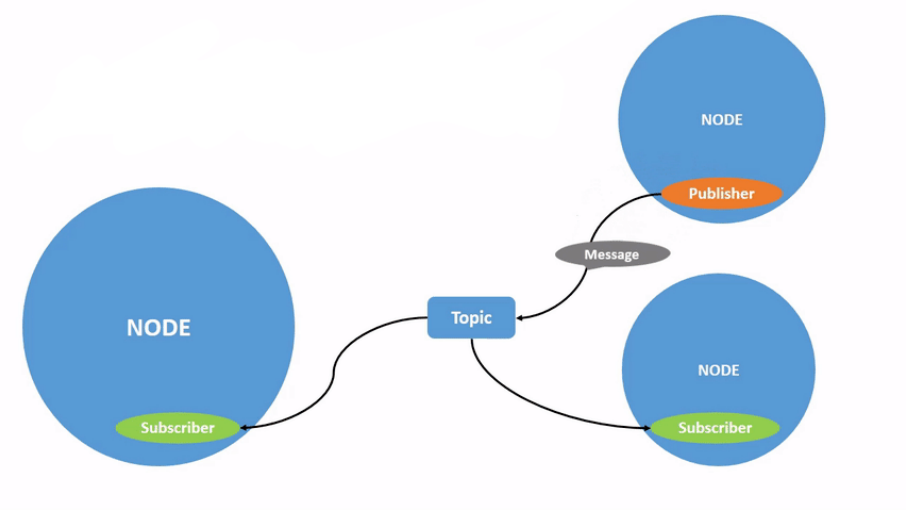


Figura 4.2: Funcionamiento de los Topics en ROS (*Understanding ROS 2 topics*, s.f.).

En la figura 4.2 vemos el funcionamiento de los topics y por qué es un mecanismo de comunicación muy bueno para el control de robots. Los nodos ”node” son programas ejecutables que usa ROS para comunicarse. El nodo ”Publisher” publica mensajes continuamente en un topic, estos pueden ser por ejemplo la lectura del sensor de odometría del Turtlebot. Entonces, un nodo ”Subscriber” puede acceder a ese topic en cualquier momento y subscribirse a ese mensaje de manera que obtiene los datos del sensor y puede utilizarlos en su programa. Como vemos, no es una arquitectura punto a punto, es decir, pueden existir varios nodos subscriptores para un único Publisher, y varios publicadores para un único Subscriber. Todo esto hace que la comunicación sea óptima ya que al no necesitar respuesta del subscriptor y no ser una estructura punto a punto, el Publisher puede estar enviando mensajes constantemente, en tiempo real, sin necesidad de esperar un ”feedback” o a que la acción anterior acabe. Así, el nodo subscriptor recibe y procesa el mensaje cuando lo necesita (Cousins, 2010), (*Understanding ROS 2 topics*, s.f.).

Además de este excelente sistema de comunicación, ROS cuenta con otra característica que lo hace un entorno muy potente, ros-pkg. Es una extensa librería de paquetes aportados tanto de manera oficial como por parte de usuarios de la comunidad, que implementan funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc (*Robot Operating System*, s.f.). Estos son, principalmente, los aspectos que han motivado la elección de ROS para este proyecto.

4.2.1 Rosbridge

Es un paquete que proporciona una funcionalidad JavaScript Object Notation (JSON) Application Programming Interface (API) a ROS para posibilitar la comunicación con programas que no son ROS. Está formado por dos partes, el protocolo rosbridge que es una especificación para enviar comandos basados en JSON a ROS, y la implementación, conformada por los paquetes de rosbridge_suite, que son los siguientes (*rosbridge*, s.f.):

rosbridge_library: paquete principal de rosbridge. Es responsable de leer la cadena JSON y enviar los comandos a ROS y viceversa.

rosapi: hace que ciertas acciones de ROS sean accesibles a través de llamadas de servicio.

rosbridge_server: proporciona una capa de transporte WebSocket. Un WebSocket es una capa de comunicación bidireccional de baja latencia entre clientes (navegadores web) y servidores. Al proporcionar una conexión WebSocket, rosbridge_server permite que las páginas web se comuniquen con ROS utilizando el protocolo rosbridge.

En este sistema de teleoperación se usa el servidor creado por rosbridge_server para comunicar Unity con ROS. Esto facilita la comunicación maestro-esclavo ya que permite la publicación o subscripción de topics utilizando sockets TCP-IP, mediante una conexión sencilla y accesible como es la conexión wifi.

4.3 Unity

Unity es un motor para la programación de videojuegos Two-Dimensional (2D) y 3D multiplataforma creado por Unity Technologies. Su primera versión se lanzó en la Conferencia Mundial de Desarrolladores de Apple en 2005 para generar proyectos solo para Mac. En 2015 se lanzó su última versión Unity 5 que introduce desarrollo multiplataforma, mecanismos de depuración, arreglo de bugs y texturas, y compatibilidad con otros programas de diseño 3D entre otras muchas características. Vemos un ejemplo de programación en una de las versiones más actuales del entorno en la figura 4.3.

Los motores gráficos que utiliza son Open Graphics Library (OpenGL) (en Windows, Mac y Linux), Direct3D (solo en Windows), y OpenGL for Embedded Systems (OpenGL ES) (en Android e iOS). Esta es una característica importante ya que las gafas VR con las que se ha trabajado solo funcionan bajo Direct11 por lo que es imprescindible que el diseño en Unity sea compatible con esta API.

Cuenta con soporte integrado para Nvidia y usa el lenguaje ShaderLab para la creación de sombreadores. Estos son programas informáticos que realizan cálculos gráficos y que interaccionan directamente con la unidad de procesamiento gráfico Graphics Processing Unit (GPU). Son utilizados para realizar transformaciones de vértices o coloreado de píxeles, entre otras labores, haciendo el procesamiento directamente en la GPU y no en los scripts que conforman el videojuego, reduciendo así el coste computacional y el esfuerzo del ordenador.

Figura 4.3: Entorno de programación de videojuegos Unity (*Unity3d*, s.f.).

Esta ha sido también una característica clave para la realización del sistema.

Por último, cabe destacar que, en los últimos años, a su amplia compatibilidad con distintas plataformas objetivo: PC (Windows, Linux, etc.), PlayStation4, XboxOne, Nintendo Switch, entre otras, se han añadido varios dispositivos de realidad extendida como PlayStation VR, Microsoft Hololens o la gama de productos de Oculus que son los utilizados en este proyecto (*Unity (motor de videojuegos)*, s.f.).

4.3.1 Unity XR Plug-In

Unity ha desarrollado un Plug-In que permite a los proveedores de tecnologías XR integrarse con el motor de Unity y hacer un uso completo de sus funciones. Este enfoque basado en complementos mejora la capacidad de Unity para realizar correcciones rápidas de errores, distribuir actualizaciones del Software Development Kit (SDK) y admitir nuevos dispositivos XR y tiempos de ejecución, sin tener que modificar el motor central (*Unity XR Plug-In*, s.f.). En la figura 4.4 vemos como este Plug-In (zona azul) recibe las implementaciones de las distintas funcionalidades XR de los proveedores de estas tecnologías, y las procesa para que puedan ser utilizadas por los desarrolladores (zona rosa). Así se consiguen aplicaciones de realidad aumentada, mixta o virtual como la de este trabajo.

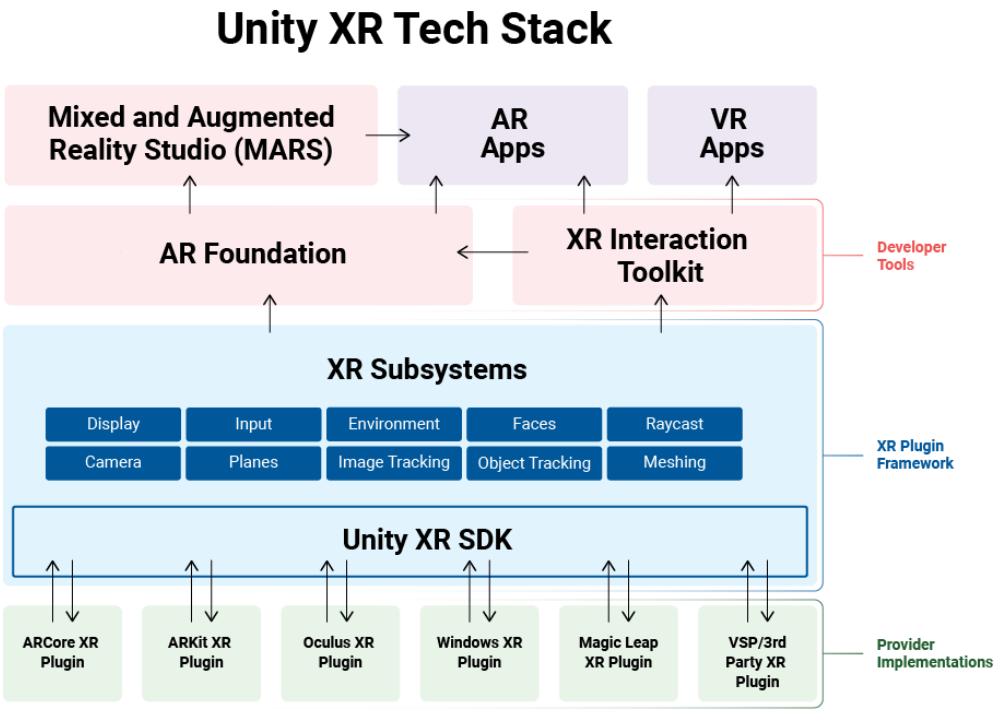


Figura 4.4: Funcionamiento del Plug-In XR de Unity (*Unity XR Plug-In*, s.f.).

4.3.2 Librería ROS#

ROS# o ROSSharp es un conjunto de herramientas y bibliotecas de software de código abierto implementadas en C# para comunicarse con ROS desde aplicaciones que funcionan bajo .NET, en particular Unity. Cuenta con el paquete RosBridgeClient para establecer las comunicaciones mediante protocolos JSON utilizando el paquete de ROS RosBridge comentado anteriormente, de manera que permite publicar y recibir topics mediante TCP-IP, como se ha hecho en este sistema. La librería implementa varios tipos de mensajes y datos propios de ROS, así como Publishers y Subscribers para ciertos topics. Además permite introducir nuevos tipos y funcionalidades creadas particularmente, compilando la librería en Visual Studio y añadiéndola a Unity (*ROS-Sharp*, s.f.).

4.4 Oculus Quest 2

Es un sistema de realidad virtual todo en uno diseñado por Oculus, empresa perteneciente a Facebook. Es un dispositivo portátil y puede funcionar de manera independiente ejecutando aplicaciones VR para Android descargadas directamente de la Oculus Store o conectarse al PC mediante su cable de alta velocidad Oculus Link para jugar a juegos VR de PC. Gracias a esta conexión con el ordenador, también pueden ser conectadas a Unity mediante

el XR Plug-In comentado, y ser usadas para el desarrollo de aplicaciones como haremos en este caso. Es un sistema económico y accesible dentro de la gama de gafas de realidad virtual del mercado y cuenta con un procesador Snapdragon XR2 ultra rápido y gráficos de última generación, con una resolución de 1832 x 1920 píxeles por ojo. Además, posee dos controladores Oculus Touch con tecnología de seguimiento de manos, muy útiles para el movimiento y otras funcionalidades (*Oculus Quest 2*, s.f.). Estas características hacen que estas gafas VR sean una muy buena opción para realizar una interfaz de teleoperación inmersiva como la de este proyecto. En la figura 4.5 se muestra el sistema de gafas VR Oculus Quest 2.

Figura 4.5: Sistema de realidad virtual Oculus Quest 2 (*Oculus Youtube*, s.f.).

5 Desarrollo

En los siguientes apartados se va a explicar cómo, utilizando las herramientas mostradas en la sección anterior (4), se ha llevado a cabo el diseño, la programación y la implementación de cada uno de los elementos del sistema. También se va a detallar su uso y de qué manera se integran con el resto de componentes para lograr el funcionamiento de la arquitectura mostrada en la figura 1.2, explicada en la introducción del trabajo (1). Volvemos a exponer esta imagen en la figura 5.1 para indicar que elementos conforman los sistemas maestro, esclavo y canales de comunicación y tener una referencia para este apartado.

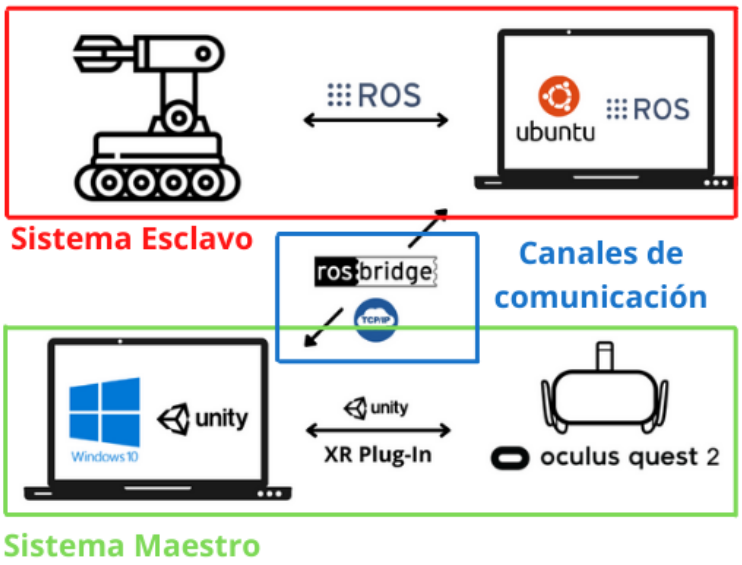


Figura 5.1: Sistemas maestro y esclavo, y canales de comunicación.

5.1 Bases del sistema de teleoperación

En primer lugar, debemos conocer cuáles son los procedimientos a seguir para instalar y configurar las herramientas, paquetes y librerías necesarias para construir los cimientos de nuestros sistemas maestro y esclavo. Así, obtendremos la base del sistema de teleoperación y, sobre esta, se podrá comenzar a trabajar en la comunicación y la realización de la interfaz.

Sistema Esclavo: Como vemos en la figura 5.1, el sistema esclavo está conformado por el robot (que en este caso es el Turtlebot2) y un ordenador que trabaja sobre el sistema operativo Ubuntu y utiliza ROS para gobernarlo y establecer la comunicación. Como se ha comentado anteriormente, por la situación actual de pandemia el uso del robot se verá limitado a una simulación. De este modo, para construir las bases del sistema esclavo instalamos ROS Kinetic (*ROS Kinetic*, s.f.) en nuestra computadora y agregamos los paquetes de Gazebo7 (5.1) (software para la simulación de robots) y Turtlebot2 (5.2). Estos paquetes pueden ser añadidos mediante los comandos mostrados a continuación, ejecutados desde la terminal de Ubuntu y con ROS ya instalado.

Código 5.1: Paquetes de Gazebo7 para ROS Kinetic

```
1 $ sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable lsb\_release -cs main" > /etc/apt/sources.list.d/gazebo-stable.list'
2 $ wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
3 $ sudo apt-get update
4 $ sudo apt-get install gazebo7
5 $ sudo apt-get install libgazebo7-*
```

Código 5.2: Paquetes de Turtlebot2 para ROS Kinetic y Gazebo7

```
1 $ sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-kobuki-ftdi ros-kinetic-ar-track-alvar-msgs ros-kinetic-turtlebot-gazebo
```

Sistema Maestro: El sistema maestro lo compone un ordenador funcionando sobre Windows10, en el que se utiliza Unity para implementar la interfaz inmersiva de realidad virtual y comunicarse tanto con el robot como con las gafas VR gracias a su Plug-In de XR, y las propias gafas de realidad virtual Oculus Quest 2 manejadas por un operador. Así, debemos instalar Unity (*Unity*, s.f.) en nuestro PC y crear un nuevo proyecto 3D, añadir el Plug-In "XR Plug-In" desde su gestor de paquetes "Package Manager" y agregar el paquete oficial de Oculus para la integración de gafas VR en proyectos de Unity desde su repositorio AssetsStore (*Oculus Integration for Unity*, s.f.).

Además de todos estos elementos para Unity, debemos instalar en el ordenador el software que posibilita la conexión por cable con las Oculus Quest 2 (*Oculus Link*, s.f.), que deberá realizarse mediante un cable Universal Serial Bus (USB) 3 de alta velocidad, y que permitirá la ejecución de la interfaz en las gafas.

Canales de comunicaión: Para las comunicaciones se utiliza el protocolo TCP-IP, ejecutado en Unity mediante la librería de código abierto comentada en la metodología, ROS-Sharp (*ROS-Sharp*, s.f.), que debemos agregar a nuestro proyecto creado en Unity en el sistema maestro. Y en ROS, gracias al paquete ROS-Bridge que puede instalarse mediante el siguiente comando (5.3) en la terminal Ubuntu, en el sistema esclavo.

Código 5.3: Paquetes de ROS-Bridge para ROS Kinetic

```
1       $ sudo apt-get install ros-kinetic-rosbridge-server
```

5.2 Modelo virtual del Turtlebot2

Tras preparar la base de nuestro sistema de teleoperación podemos comenzar a trabajar en la interfaz inmersiva de realidad virtual. Lo primero que debemos hacer es transferir el modelo real del Turtlebot2 a un modelo virtual en Unity (*How to link Unity3D and Gazebo for robot control*, s.f.), para ello hacemos uso del formato de ROS, Unified Robot Description Format (URDF). Se trata de un fichero Extensible Markup Language (XML) que representa el modelo completo del robot y que es muy útil para realizar representaciones 3D, como necesitamos en este caso.

En la librería ROS-Sharp existe una carpeta llamada "ROS" que contiene los archivos necesarios para realizar la transferencia del modelo URDF desde el terminal ROS a Unity y realizar las comunicaciones más básicas. En el sistema esclavo debemos crear un catkin workspace y en su carpeta "src" añadimos el directorio "file_server", encontrado en la librería.

Una vez hecho esto y tras compilar nuestro workspace con "catkin_make" ejecutaremos el fichero .launch contenido en "file_server" que publicará el URDF del robot:

Código 5.4: Publicación del URDF del Turtlebot2

```
1   $ roslaunch file_server publish_description_turtlebot2.launch
```

Entonces, en Unity, en la ventana "RosBridgeClient", "Transfer URDF from ROS" (figura 5.2), nos aseguramos de poner la dirección Internet Protocol (IP) del ordenador que está ejecutando la simulación (en caso del robot real sería la IP del robot), y la ruta del directorio URDF de la librería ROS-Sharp en "Asset Path". El resto de configuración puede dejarse como aparece en la imagen. Clickamos en "Read Robot Description" y se transferirá el modelo del robot a Unity.

Al transferir el modelo del Turtlebot, se creará un objeto 3D (gameobject) que estará formado por cada una de los elementos que componen el robot con sus medidas y propiedades reales. Contará con un objeto 3D "hijo" ("child") para cada una de estas partes, obteniendo así toda la estructura del robot en 3D con articulaciones, sensores y actuadores de este. El funcionamiento de los dos últimos se tendrá que implementar mediante la comunicación con los sensores y actuadores reales del robot, realizando el procesamiento adecuado de los datos, lo que llevará a la construcción de la interfaz, objetivo de este trabajo. En la figura 5.3 se puede ver el modelo virtual del Turtlebot2 transferido a Unity y, a la izquierda de ella, cada uno de los elementos que lo conforman.

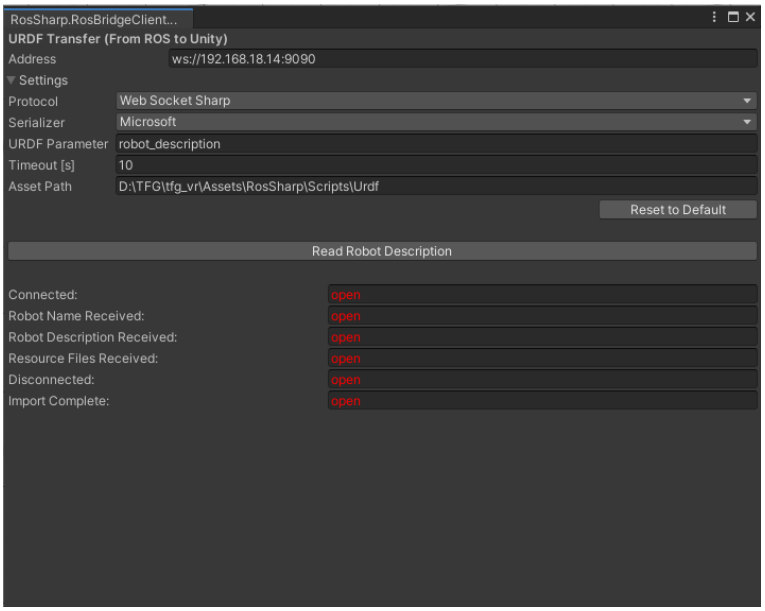


Figura 5.2: Transferencia del URDF del Turtlebot2 a Unity.

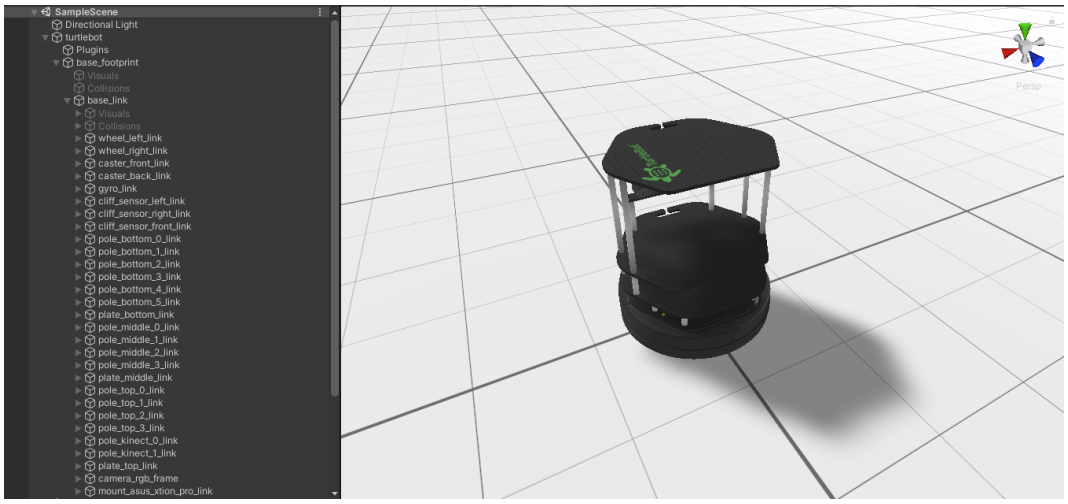


Figura 5.3: Modelo Virtual del Turtlebot2 en Unity.

5.3 Comunicación básica maestro-esclavo

Una vez transferido el modelo del robot a Unity podemos centrarnos en las comunicaciones para lograr que este objeto 3D se convierta en una representación virtual del robot en tiempo real.

En Unity, a cada objeto 3D podemos agregarle numerosas propiedades, funciones y/o interacciones al clickar en este y pulsar en ”add component” en la ventana ”Inspector”. Algunas de estas (las más generales) están implementadas por los desarrolladores del entorno y pueden

ser añadidas y configuradas directamente. Sin embargo, lo interesante de esto es que nosotros mismos podemos crear nuestros propios componentes para el objeto haciendo uso de scripts en lenguaje C#, teniendo infinidad de posibilidades. Aquí es donde entra la librería ROS# que aporta varias de estas propiedades para conseguir la comunicación con robots y permite implementar nuevos subscribers y publishers que serán esenciales en la creación de la interfaz.

De esta manera, al objeto 3D creado con el modelo virtual del Turtlebot, llamado "turtlebot", añadimos el componente "ROS Connector", el cual permitirá la conexión mediante IP con ROS-Bridge y posibilitará a otros scripts suscribirse o publicar topics en el robot. Lo configuramos escribiendo la IP del robot (en esta caso el PC con la simulación) como vemos en la figura. 5.4.

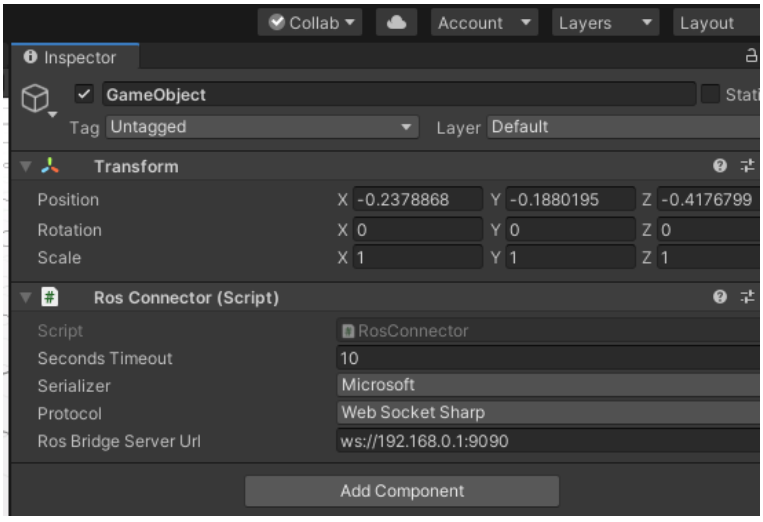


Figura 5.4: ROS Connector, script para la comunicación con ROS-Bridge.

Así, podemos hacer la primera prueba que tan solo realizará la conexión entre Unity y el robot. Primero ejecutamos en el sistema esclavo la simulación que creará el servidor ROS-Bridge y permitirá que el maestro se comunique (5.5):

Código 5.5: Ejecución del entorno simulado del robot

```
1   $ roslaunch gazebo_simulation_scene gazebo_simulation_scene.launch
```

Finalmente, damos "play" en Unity lo que hará funcionar nuestro proyecto, ejecutando el script "ROS Connector" enlazado al turtlebot. Si todo está correcto la conexión habrá sido un éxito mostrando en la consola de Unity el mensaje mostrado en la figura 5.5 (donde también se muestra el mensaje de desconexión) y en la terminal de Ubuntu el mensaje "Client Connected" y los topics que está publicando y a los que se está subscribiendo el cliente. En la figura 5.6 se muestra un ejemplo en el que se da la conexión con 1 cliente y este se subscribe al topic "/odom".

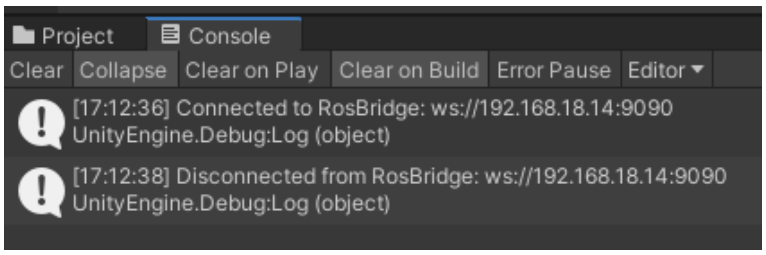


Figura 5.5: Mensaje mostrado en la consola de Unity si la conexión es satisfactoria.

```
[spawn_turtlebot_model-6] process has finished cleanly
log file: /home/manu/.ros/log/24b85c7e-d2d8-11eb-bb92-c8ff28f79445/
spawn_turtlebot_model-6*.log
2021-06-21 23:32:40+0200 [-] [INFO] [1624311160.453021, 22.280000]:
 Client connected.  1 clients total.
2021-06-21 23:32:40+0200 [-] [INFO] [1624311160.888060, 22.760000]:
 [Client 0] Subscribed to /odom
```

Figura 5.6: Mensaje mostrado en la terminal Ubuntu si la conexión es satisfactoria. Se indican el número de clientes y los topics subscritos y publicados por estos.

Este procedimiento habrá que hacerlo cada vez que queramos que un objeto de Unity se comunique con el robot ya que como se ha dicho, el script "ROS Connector" conecta el puente generado por ROS-Bridge con Unity.

5.4 Movimiento

Llegados a este punto tenemos el modelo virtual del turtlebot preparado y sabemos hacer uso de la librería ROS# para establecer las comunicaciones entre el maestro y el esclavo. El siguiente paso para lograr la interfaz, y el más básico en cuanto a teleoperación y control de robots se refiere, será conseguir el control del movimiento del robot por parte el operador. Para ello, nos subscribiremos y publicaremos a ciertos topics desde el maestro y realizaremos un procesamiento de los datos en el esclavo, como explicamos a continuación.

Como se ha comentado antes, en la ventana "Inspector" de Unity podemos añadir a un objeto scripts de la librería ROS# para la comunicación. De este modo, creamos un objeto llamado "RosConnector" y le agregaremos los siguientes. No es necesario añadirlos al gameobject del turtlebot ya que mediante parámetros de esos scripts hacemos referencia al URDF del robot.

ROS Connector: es el script comentado en la comunicación básica. Crea el puente entre Unity y ROS-Bridge para lograr las comunicaciones y permite a otros scripts la publicación y subscripción de topics.

Joint State Subscriber: suscriptor del topic "/joint_states". Recibe la información sobre el estado de las articulaciones, en este caso las ruedas derecha e izquierda del Turtlebot, para ser usada en el URDF del robot.

Joint State Patcher: procesa la información del joint state subscriber para adaptarla al URDF del robot y que el estado de las articulaciones virtuales sea el mismo que en la realidad.

Odometry subscriber: se suscribe al topic "/odom" que aporta la información de la posición del robot en el espacio real y la aplica al objeto con el URDF del robot consiguiendo la realimentación virtual de la posición del robot.

Joy Axis Reader: este script lee los valores numéricos (float) que provienen del movimiento de los ejes del joystick que servirá para mover el robot. Se añaden dos; uno para el movimiento vertical y otro para el horizontal.

Joy Publisher: unifica los valores leídos por "joy axis reader" en un solo mensaje y lo publica en el topic "/joy". Esto permitirá la subscripción a los datos en el esclavo, de manera que se conseguirá el movimiento del robot.

En la figura (5.7) vemos como estos scripts son añadidos al objeto en Unity.

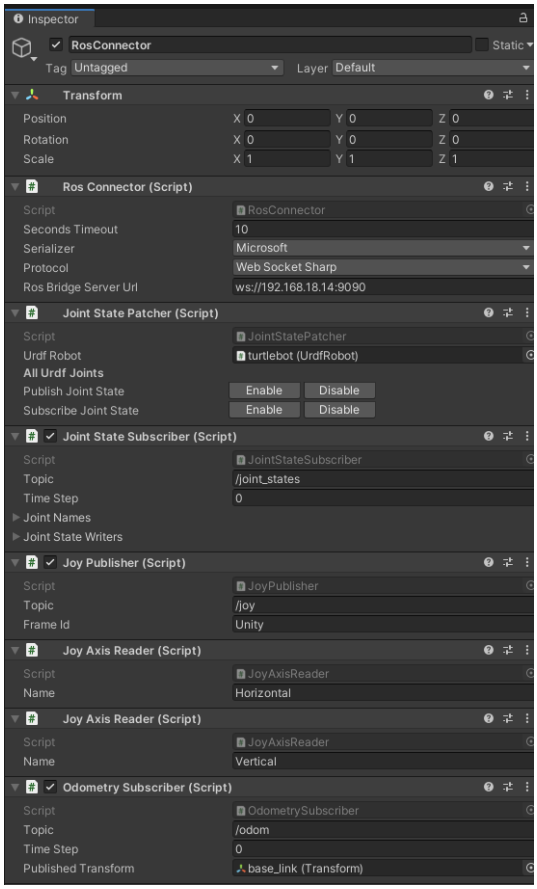


Figura 5.7: Scripts necesarios para el control del movimiento del robot y la realimentación de la posición.

Una vez hecho esto, necesitamos suscribirnos al topic "/joy" en el esclavo para traducir los valores numéricos provenientes de los ejes del joystick en velocidades aplicables a los motores del Turtlebot. Para ello, hacemos uso del script escrito en lenguaje python "joy_to_twist.py" el cual se ejecuta desde el mismo .launch con la simulación del robot (5.5). El topic contiene un vector de floats que puede tener hasta 6 posiciones. Cada una de estas posiciones da la información del movimiento del joystick en 6 ejes distintos: horizontal, vertical, y cuatro diagonales. Estos valores se transforman en datos valiosos para las articulaciones del robot con el uso de varios parámetros de escala y estableciendo movimiento lineal o angular según el eje del que se reciba la información.

De esta forma, el funcionamiento del control del movimiento del esclavo por parte del maestro quedaría así: en el entorno virtual se establece el puente que permite la comunicación con el robot. Entonces, se realizan las subscripciones a los topics que dan la información sobre la posición del robot y el estado de sus articulaciones, aplicando y actualizando estos datos al objeto con el URDF del robot en el mundo virtual, lo que crea una representación virtual de la posición y estado del robot real. Al mismo tiempo, el maestro lee los valores del joystick que controlará el movimiento del robot y los publica en un topic al que el esclavo se subscribe y procesa, moviendo los motores de las ruedas del Turtlebot consecuentemente.

Como se indica, el movimiento del robot será controlado por un joystick y su posición será actualizada en tiempo real en el mundo virtual. Así, obtenemos uno de los pasos más importantes de la interfaz y por lo tanto, de este trabajo, que es el control del movimiento de un robot a distancia, usando tecnologías de realidad virtual.

5.5 Colisiones

Para representar las colisiones virtualmente se probaron varios procedimientos que permite Unity. Estos sirven para dotar de esta propiedad a los objetos pero, o eran muy costosos computacionalmente, o no daban buenos resultados para lo requerido en este trabajo. Finalmente se llegó a la conclusión de que lo más práctico y eficaz, que a la vez es lo más sencillo de programar, era utilizar los bumpers con los que cuenta el Turtlebot. De esta manera, en el script que permite el movimiento del robot comentado antes ("joy_to_twist.py"), nos subscribimos al topic "/mobile_base/events/bumper" y, simplemente, si no se detecta ningún bumper activado quiere decir que no existe colisión y permitimos la traducción de los valores del joystick a velocidades. En contraposición, si se detecta algún bumper activado sabemos que hay colisión con algún objeto y no permitimos esta traducción haciendo que no llegue ningún valor de velocidad a las ruedas del robot. En este momento, el robot se para independientemente de las órdenes del operador desde el joystick y, como existe realimentación de la posición en tiempo real, el modelo del robot se detendrá de la misma manera en la interfaz consiguiendo así representar las colisiones virtualmente de una manera fiel a la realidad.

5.6 Point of View

Recapitulando, hemos conseguido crear una interfaz de teleoperación con la que controlamos el movimiento del robot a distancia y tenemos realimentación virtual de su posición y de las posibles colisiones que pueda haber. Estos elementos son imprescindibles a la hora de teleoperar un robot pero, otro de los componentes más importantes y, principalmente, el que ha causado la realización de este proyecto, es el de trasladar el entorno real en el que se encuentra el robot a la realidad virtual. Para ello haremos uso de el sensor 3D Kinect del Turtlebot entre otros, y explicaremos los procesos seguidos en las siguientes líneas.

En primer lugar trasladaremos al mundo virtual lo que ve el robot en primer plano, es decir, su punto de vista Point of View (POV) y más tarde entraremos en otras cuestiones como representar el entorno completo. Esto se ha realizado de dos formas: reproducir la imagen de la cámara en 2D en una pantalla que vemos delante de nosotros en la realidad virtual, lo que puede tener algunas ventajas. Y representar lo que ve el robot en 3D de manera fiel a la realidad, utilizando nubes de puntos (*PointClound Streaming from ROS*, s.f.).

5.6.1 Imagen 2D

Realizar este proceso es mucho más sencillo ya que tan solo tenemos que trasladar la imagen 2D recogida por la cámara a una pantalla que aparecerá delante de nosotros. Puede parecer que esto no tiene utilidad, pero puede aportar ventajas a la hora de buscar objetos entre otras cosas, ya que podemos ver la imagen con mayor calidad y detalle que en 3D, donde se pierde gran cantidad de información. No obstante, este método no aporta la sensación de profundidad y se pierde la capacidad inmersiva por lo que tan solo es un recurso que puede ser usado en ciertos momentos para complementar la interfaz.

Para implementarlo, al objeto "RosConnector" en Unity, le añadimos el script de la librería ROS#, "Image Subscriber". Este componente se subscribe al topic "/camera/rgb/image_raw/compressed" y convierte la imagen comprimida en una textura 2D que puede ser aplicada a un objeto. Entonces, delante del robot virtual, a una distancia de 2m, colocamos un plano paralelo al plano x-y, de escala (x=0.3, y=1, z=0.3), como si de una pantalla de cine se tratase y lo enlazamos a la cámara Red Green Blue (RGB) del URDF del robot para que se mueva con esta. A ese plano aplicamos la textura 2D extraída del topic y obtenemos la primera representación virtual del POV del robot en 2D. El resultado se muestra en las figuras 5.8 y 5.9 donde vemos que ocurre lo comentado, la imagen es de alta calidad por lo que puede ser útil para fijarnos en ciertos detalles, pero no existe la sensación de profundidad e inmersión.

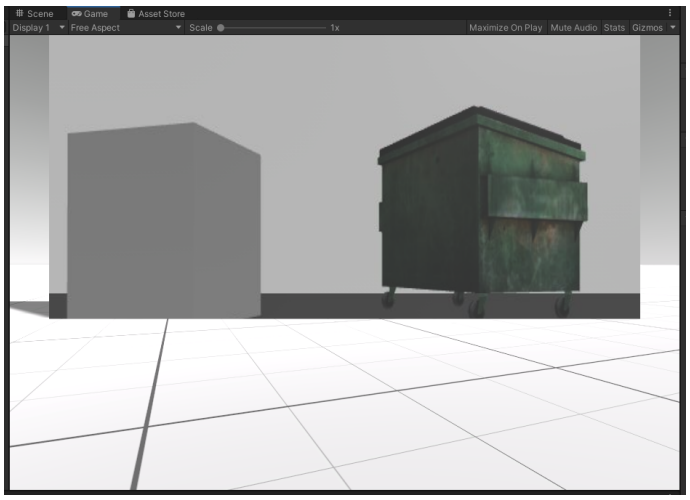


Figura 5.8: Representación virtual de la imagen 2D de la cámara. 1ª persona.

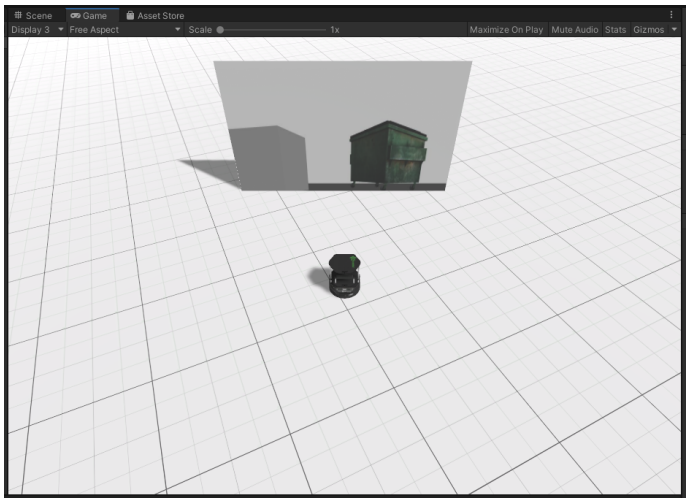


Figura 5.9: Representación virtual de la imagen 2D de la cámara. 3ª persona.

5.6.2 Nube de puntos 3D

Para trasladar el mundo real al virtual de la mejor manera posible, es necesario dar esa sensación de profundidad e inmersividad de la que hablamos. Esto se consigue con el uso de cámaras 3D como la Kinect, con la que cuenta el Turtlebot. Esta cámara tiene la capacidad de observar el entorno y codificarlo en una nube de puntos 3D, dónde cada punto tiene una posición x, y, z, con respecto al origen, que será la posición de la cámara. Esta nube es publicada en un topic por el robot de manera que podemos acceder a ella desde Unity o desde otro script como es el caso.

La nube de puntos leída por la Kinect tiene un máximo de 307200 puntos. Esto aporta gran información y detalle pero es una cantidad demasiado elevada para que Unity pueda recibirla, procesarla y mostrarla en el mundo virtual, en tiempo real. En la figura 5.10 vemos

mostrada por pantalla la nube de puntos original, recibida del sensor 3D del Turtlebot, donde se aprecia lo que comentamos, cuenta con una gran cantidad de puntos que aportan mucho detalle pero que procesarlos todos sería demasiado costoso computacionalmente. Para solucionar este problema creamos el script "pov.cpp" en lenguaje c++, que será el encargado de procesar la nube, reducirla y publicarla de nuevo para su posterior subscripción en Unity, con ayuda de la Point Cloud Library (PCL) (*Point Cloud Library*, s.f.).

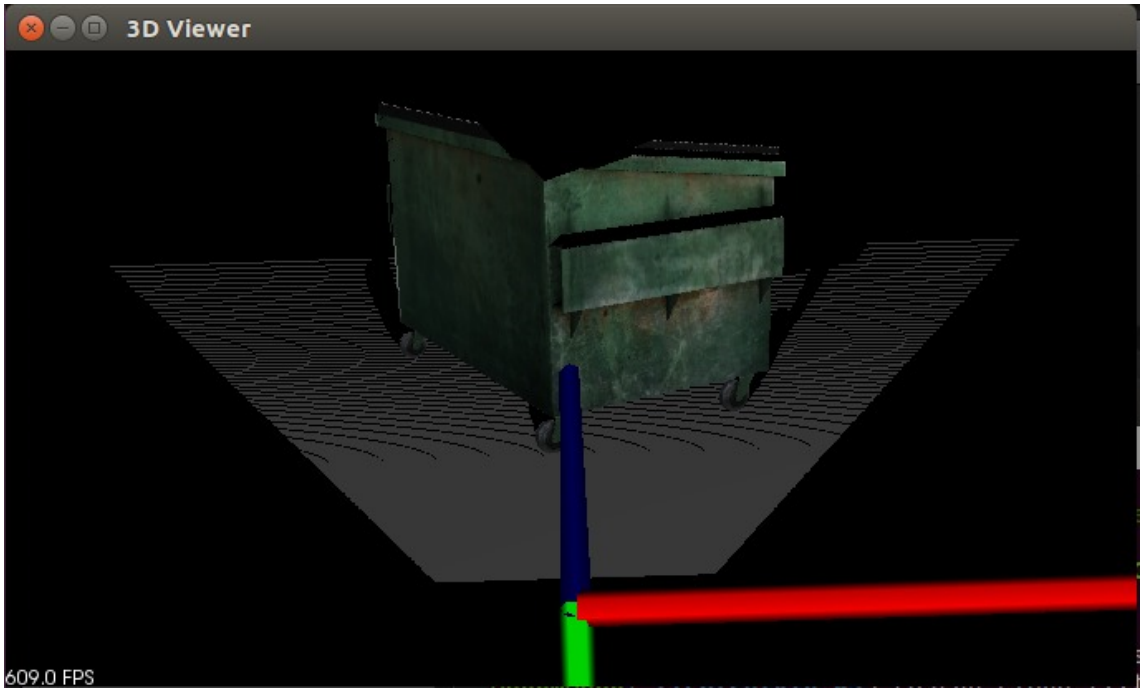


Figura 5.10: Nube de puntos original, recibida por la Kinect.

El fichero "pov.cpp" se encontrará en el sistema esclavo dentro del directorio "cloud_filtering" perteneciente al catkin workspace comentado en apartados anteriores, creado para la ejecución de la simulación del entorno del robot. El funcionamiento de este programa es el siguiente: se crea un subscriptor al topic publicado por la kinect con la nube de puntos, "/camera/-depth/points". Este mensaje es convertido a una nube de puntos de la PCL, PCLPointCloud2 utilizando las funciones propias de esta librería. Entonces, se pasa por un algoritmo de voxel-grid, perteneciente también a la PCL, que agrupa una cierta cantidad de puntos de la nube en cubos 3D. El número de puntos agrupados puede ser modificado y ajustado con ciertos parámetros del algoritmo y cada punto lleva su información de color RGB con él. Estos cubos 3D son reducidos a un punto para cada uno utilizando su centroide y se convierten de nuevo a una nube de puntos que se publica en otro topic llamado "depth_points_filtered". De esta forma, procesamos la nube obtenida de la cámara 3D y la reducimos notablemente en número de puntos, de manera que podrá ser utilizada por Unity para crear el mundo virtual. El algoritmo comentado, así como otros comparados con este y los parámetros de ajuste, serán explicados en detalle en el apartado de experimentación 6.1.

Una vez hemos logrado reducir el tamaño de la nube, procedemos a leerla y procesar-

la en Unity para poder mostrarla como objetos virtuales. Para ello, al objeto ”RosConnector” le añadimos el componente ”Point Cloud Subscriber” que se subscribirá al topic ”depth_points_filtered” con la nube de puntos reducida. Este script extraerá los puntos de la nube y los colores RGB correspondientes a cada uno y los agrupará en vectores. Entonces, creamos un nuevo gameobject que se llamará ”PointCloudRenderer” y le añadimos el script ”Point Cloud Renderer” al que enlazaremos el ”Point Cloud Subscriber” por parámetros. Este programa lee los vectores de puntos y colores recibidos del subscriber y los aplica a una malla (mesh) que será la que podrá mostrarse en el entorno. Estas mallas aportan muchas ventajas ya que introducen algoritmos para representar y renderizar cada punto de manera que la malla completa se trata como un sólo objeto, siendo mucho más eficientes computacionalmente que mostrar cada punto por separado.

De esta forma, habremos completado la parte de la interfaz en la que el punto de vista (POV) del robot es trasladado del mundo real al virtual de una manera inmersiva. Para comprobar este proceso, tras ejecutar la simulación en gazebo (código 5.5), debemos ejecutar el script ”pov.cpp” que procesa y reduce la nube de puntos, y que pertenece al ros package ”cloud_filtering”. Para ello, y previamente habiendo compilado el workspace con ”catkin_make” modificando su ”cmakelists.txt” para obtener un ejecutable llamado ”pov”, debemos llamar a este último desde la terminal como se muestra en el siguiente comando (5.6). En la terminal se mostrará el mensaje ”Publishing” si todo está correcto.

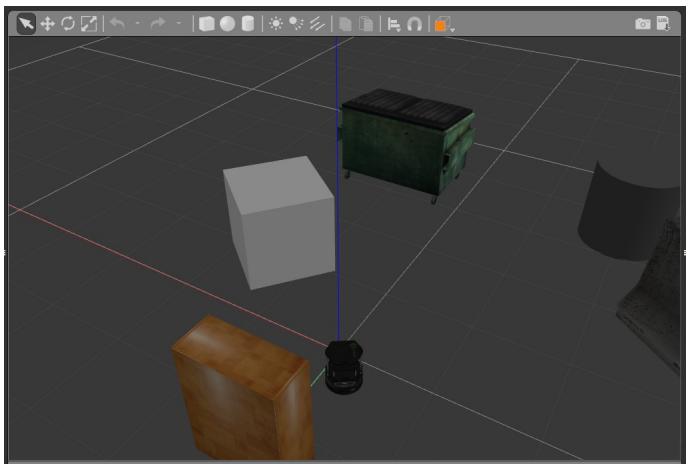
Código 5.6: Ejecución del script ”pov.cpp” que procesa y reduce la nube de puntos recibida del sensor 3D.

```
1   $ rosrun cloud_filtering pov
```

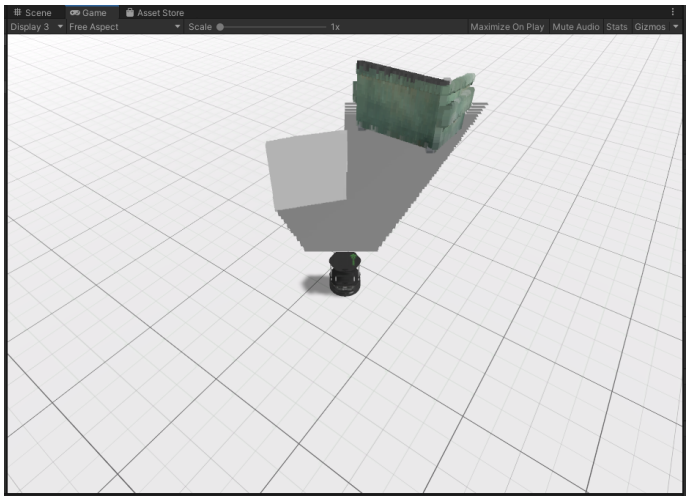
Finalmente, damos ”play” en Unity para poner a funcionar la interfaz y ejecutar los scripts que se subscriben y muestran la nube reducida. En la figura 5.11 vemos el resultado.

Se aprecia como, pese a haber perdido información respecto a la nube de puntos original, el mundo real (simulado en gazebo) es representado de manera fiel a la realidad en Unity, consiguiendo percibir en la realidad virtual lo que está viendo el robot en ese momento. También vemos como la sensación de profundidad si se consigue en comparación con el método 2D mostrado en el apartado 5.6.1, por lo que se logra la inmersión buscada en el proyecto. Otro aspecto a tener en cuenta es el del suelo, en este caso, el suelo que detecta la cámara también es mostrado, lo que aumenta esta sensación de estar en el lugar en el que se encuentra el robot.

Cabe destacar que en el script ”Point Cloud Renderer” existe el parámetro ”Point Size” que, como su nombre indica, permite cambiar el tamaño de cada punto de la malla. Esto es muy útil ya que podemos modificar en Unity como se ven los objetos dependiendo de cuántos puntos tenga la nube que se recibe, consiguiendo así elementos sólidos sin huecos entre sus puntos. Esta característica es sobre todo importante al visualizar los elementos en primera persona ya que en tercera, aunque el tamaño de los puntos no sea el correcto, puede pasar desapercibido por la perspectiva. En la figura 5.12 vemos como se ven los objetos en primera persona con un tamaño de cubo de 1, indicado a la derecha por el parámetro ”Point Size”.



(a) Punto de vista real (simulación en gazebo).



(b) Punto de vista en la interfaz de realidad virtual.

Figura 5.11: Comparativa entre el punto de vista real del robot y su representación virtual.

5.7 Puntos 3D

Se ha comentado que en Unity se recibe la nube de puntos, se transforma a una malla y se muestra, pero no se ha explicado de qué manera se consiguen los cubos 3D alrededor de cada punto y cómo se logra cambiar su tamaño con el parámetro ”Point Size”. Este es un aspecto importante ya que, en primer lugar, es imprescindible a la hora de hacer que el mundo mostrado en la interfaz se parezca a la realidad y no se trate solamente de pequeños puntos distribuidos por el espacio, y en segundo lugar, es necesario que este proceso se realice de manera óptima sin comprometer demasiado el coste computacional para conseguir una interfaz fluida.

Esta característica se ha conseguido con el uso de sombreadores, elemento introducido ya en la metodología del trabajo (4.3) y que ha sido imprescindible para conseguir los objetivos.

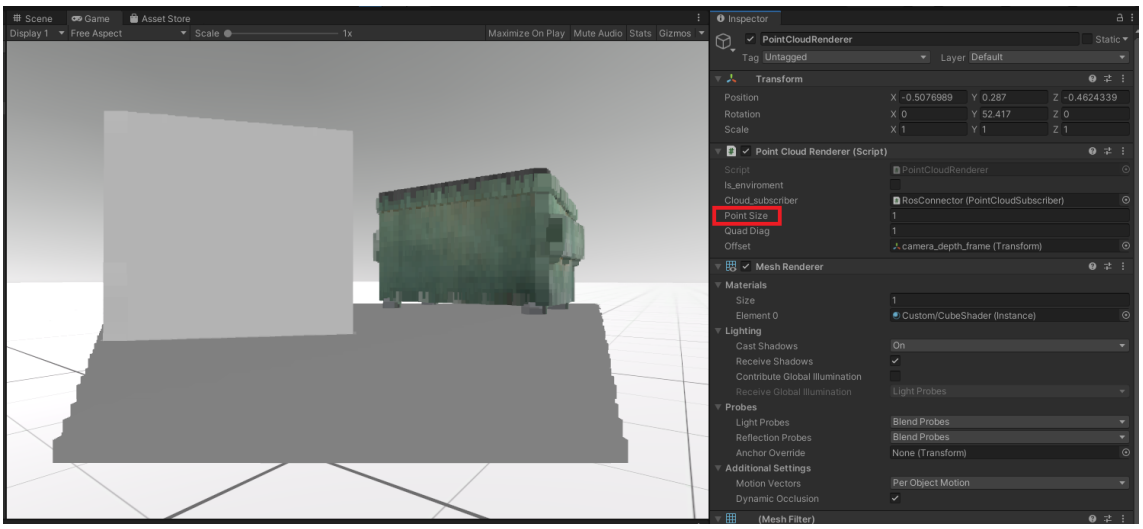


Figura 5.12: Punto de vista del mundo virtual en 1ª persona. Parámetro Point Size para variar el tamaño de cada cubo 3D.

Como ya se ha comentado, estos sombreadores pueden programarse en su propio lenguaje "ShaderLab" e interactúan directamente con la GPU para realizar trabajos de procesamiento gráfico sin comprometer la capacidad de cómputo de nuestro PC. De esta forma, estos "Shaders" son estrictamente necesarios para crear el cubo 3D al rededor de cada punto ya que se comprobó que hacer esto directamente en scripts de Unity punto por punto era totalmente inviable.

El procedimiento indicado se realiza en el fichero "CubeShader.shader". En este se utiliza el parámetro "Point Size" indicado en la figura 5.12 para establecer el tamaño del cubo. Este valor se divide entre 2 para obtener la distancia que habrá desde el centro del punto donde se está aplicando el shader, hasta cada lado del cubo. Con este dato establecemos cada vértice del cubo con su posición (x, y, z) = (pointsize/2, pointsize/2, pointsize/2) dependiendo del vértice a escribir, y le aplicamos las coordenadas de su textura base a cada uno. Finalmente, utilizamos la función "TRI_STRIP" para triangular los vértices de cada cara y dibujar el cubo sobre el punto. En la figura 5.13 se muestra el funcionamiento de esta función, una vez tenemos los vértices, se triangulan y rellenan formando la figura requerida.

Una vez implementado, un shader puede ser aplicado a un objeto en Unity como si fuera un material. De esta forma, el software lo trata como tal y los procesos que este realize son dirigidos por la GPU sin que Unity tenga nada que ver. Así, cuando escribamos nuestro shader, este será añadido a la malla de la nube de puntos como material y, por las propiedades de esta, el sombreador actuará en cada punto dibujando el cubo 3D sobre él, consiguiendo el resultado mostrado en las imágenes de la tabla 5.1.

Aunque en la tabla 5.1 podemos ver los resultados para distintos tamaños de puntos, no se pueden sacar conclusiones reales sobre cuál es mejor o peor ya que también influye la cantidad de puntos que contenga la nube recibida. En la sección de experimentación 6.2 se realizarán pruebas y comparativas más en detalle teniendo en cuenta ambos factores, de manera que se

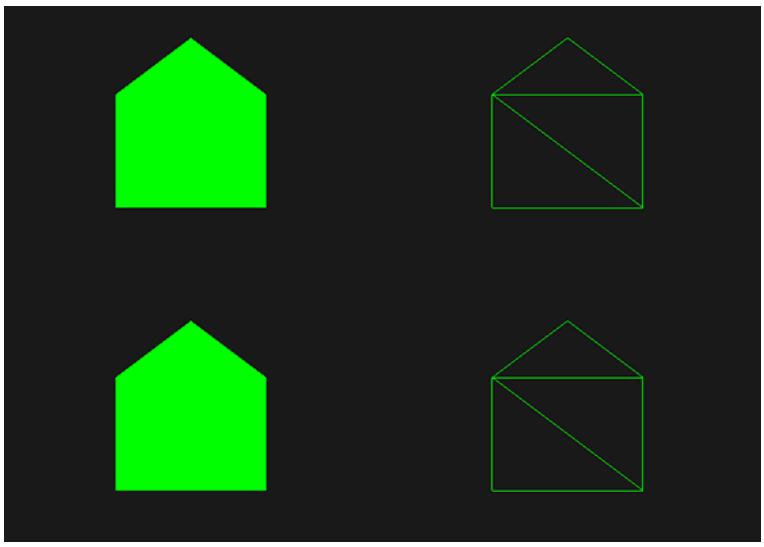
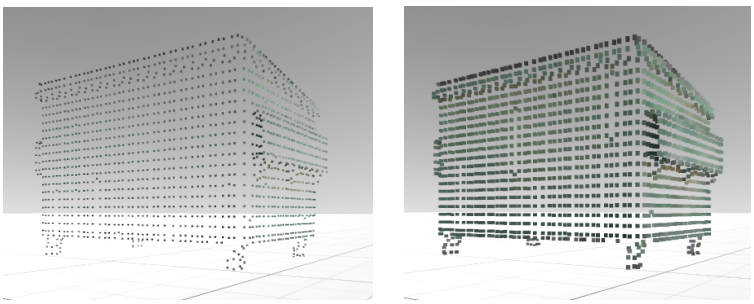
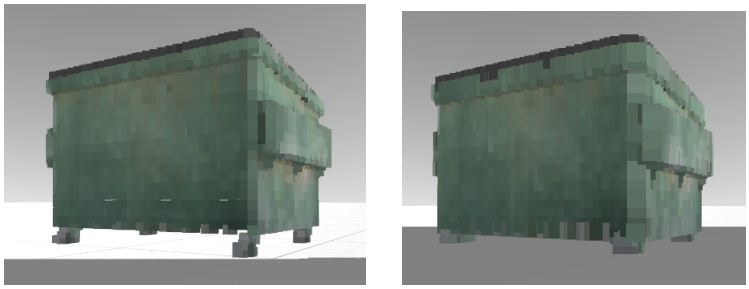


Figura 5.13: Funcionamiento de la función TRI_STRIP de ShaderLab.



$PointSize = 0.1$ $PointSize = 0.2$



$PointSize = 0.5$ $PointSize = 1$

Tabla 5.1: Punto de vista del robot en el mundo virtual, con distintos Point Size para cada punto.

obtengan resultados concluyentes.

5.8 Entorno

Llegados a este punto, ya se puede considerar que tenemos una interfaz inmersiva de realidad virtual con la que podemos teleoperar a distancia un robot, controlar su movimiento y observar en primera y tercera persona lo que está viendo en tiempo real. No obstante, una de las ventajas de este tipo de interfaces es que podemos agregar características adicionales de manera que la teleoperación por parte del operador sea más sencilla o se introduzcan elementos que mejoren las capacidades de operación. En este apartado vamos a tratar el tema del entorno del robot. Con esta interfaz podemos analizar el entorno y mostrarlo en la realidad virtual de manera que se sigue aumentando la sensación de inmersividad y puede ayudar en tareas de búsqueda y reconocimiento, entre otras. Aunque esta característica aporte numerosas ventajas, también conlleva su parte negativa. La representación del entorno no es en tiempo real, sino una adaptación de lo que el robot ha visto anteriormente. Y, aunque pueda ser actualizada en tiempo real con lo que el robot observa en el momento, las zonas donde no alcanza la cámara no serán actualizadas. De esta forma, puede ser un elemento muy bueno en ciertos aspectos pero siempre habrá que combinarlo con la visión en tiempo real que ya ha sido implementada.

Para la realización de esta nueva característica haremos uso del paquete de ROS "rtabmap" (*rtabmap_ros*, s.f.), el cual es usado para la realización de proyectos robóticos de Simultaneous Localization and Mapping (SLAM) Red Green Blue Depth (RGB-D) con restricciones en tiempo real. SLAM es una técnica para construir un mapa de un entorno desconocido, a la vez que se estima la trayectoria del robot al desplazarse dentro de este entorno. Como se ha explicado, en esta interfaz se hará uso de las técnicas SLAM para construir el mapa del entorno del Turtlebot y mostrarlo en la realidad virtual. Cabe decir que, aunque existe otro paquete de ROS solamente para crear mapas a partir de redes de ocupación, llamado Octomap, este no ha sido utilizado por las dificultades que pone al crear los mapas con información RGB, en comparación a "rtabmap", ya que es una característica imprescindible a la hora de trasladar el mundo real al virtual de manera fiel a la realidad.

El procedimiento a seguir para la construcción del mapa utilizando rtabmap_ros es el siguiente: tras añadir el paquete rtabmap a ROS y ejecutar la simulación del Turtlebot en el entorno (código 5.5), debemos llamar al .launch de rtabmap que inicia los algoritmos de SLAM en la simulación del Turtlebot (código 5.7).

Código 5.7: Ejecución de los algoritmos de slam de rtabmap_ros en una simulación del Turtlebot.

```
1   $ roslaunch rtabmap_ros demo_turtlebot_mapping.launch simulation:=true
```

Estos scripts comienzan a publicar varios topics con datos sobre el mapa que se va construyendo a partir de sensores del robot como la cámara 3D, el scaner láser y/o la odometría. Nosotros buscamos una nube de puntos para poder subscribirnos desde Unity y mostrarla en el mundo virtual pero, si accedemos a los topics de este tipo, nos damos cuenta que pese a ser nube de puntos 3D, estos tan sólo tienen componentes 2D. Esto se debe a que estos datos deben ser combinados con el grafo que contiene los datos de los nodos más recientes del mapa (se encuentra en el topic "/rtabmap/mapData") para construir así el mapa 3D.

Para realizar esto último, llamamos al nodo map_assembler de rtabmap, indicando el topic "/rtabmap/mapData" para lograr lo explicado (código 5.8).

Código 5.8: Ejecución del nodo map_assembler que construye el mapa 3D a partir del mapData.

```
1   $ rosrun rtabmap_ros map_assembler mapData:=/rtabmap/mapData ↩
        ↪ _regenerate_local_grids:=true --Grid/FromDepth true
```

Como comentamos, el nodo map_assembler combina la información de los topics publicados por rtabmap, para ensamblar el mapa 3D y publicarlo, esta vez sí, como nubes de puntos 3D, entre otras representaciones. Entonces, en Unity, crearemos un nuevo objeto y le agregaremos de nuevo un script "ROS Connector" y un "Point Cloud Subscriber" que se subscribirá al topic que proporciona map_assembler con la nube de puntos del entorno, "/map_assembler/cloud_map". En este caso, la información x, y, z, r, g, b de cada punto viene establecida de otro modo por lo que tendremos que realizar algunas modificaciones en el subscriptor para extraer cada punto correctamente. En otro gameobject añadiremos el script "Point Cloud Renderer" y, teniendo en cuenta estas modificaciones, mostraremos, de la misma forma que el punto de vista del robot, la malla con los puntos de la nube, aplicando el material del shader explicado en el apartado anterior (5.7). De esta forma, podremos realizar con el robot un repaso por el entorno y mostrarlo en la realidad virtual consiguiendo trasladar el entorno real completamente. En la figura 5.15 podemos ver el resultado de trasladar el entorno real que percibe el robot (5.14) dando una vuelta de 360º sobre si mismo. Se ve como, gracias a los algoritmos de SLAM se construye el mapa 3D y, gracias al procesamiento en Unity y al ajuste del tamaño de los puntos con el parámetro "Point Size", se consigue una representación virtual satisfactoria llevando el entorno del robot a la realidad virtual.

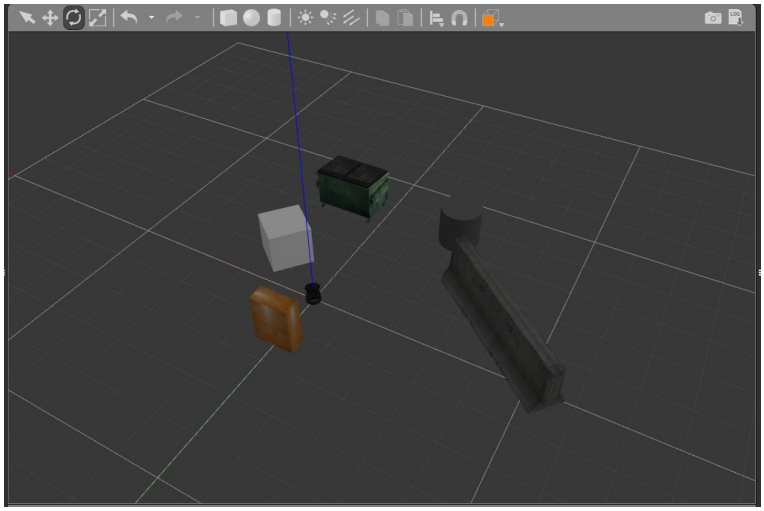


Figura 5.14: Visión del entorno real (simulado en gazebo).

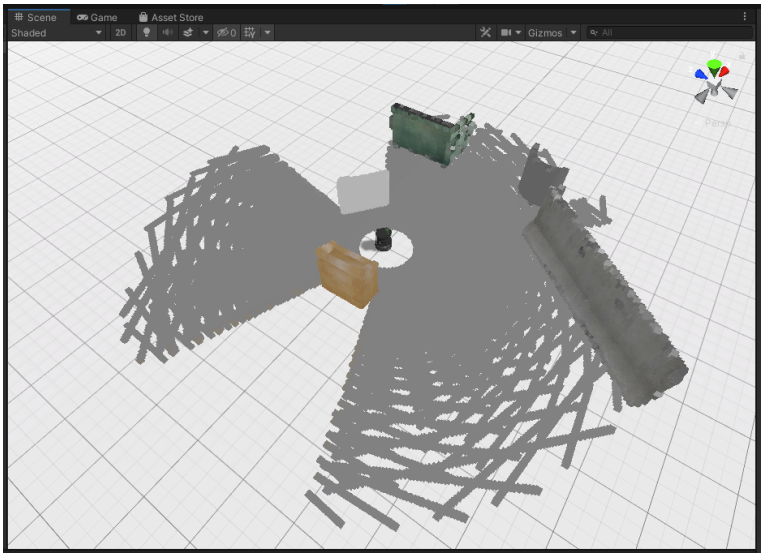


Figura 5.15: Visión del entorno virtual.

5.9 Realidad Virtual

En este momento podemos decir que ya hemos logrado implementar una interfaz de realidad virtual en la que se lleva a cabo el control del movimiento del robot y se traslada el mundo real al virtual, en tiempo real, y el entorno completo, de manera satisfactoria y fiel a la realidad. No obstante, para conseguir que esta interfaz de teleoperación se convierta en una interfaz inmersiva debemos introducirla en las gafas de realidad virtual y controlar al robot desde ellas. Una vez realizado todo lo explicado, esto se convierte en algo sencillo ya que pueden ser conectadas a Unity y hacerlas funcionales de manera simple y rápida.

En primer lugar, como hemos explicado en el apartado 5.1, debemos agregar a Unity el XR Plug-In desde su ”package manager” y el paquete para la integración de gafas Oculus (*Oculus Integration for Unity*, s.f.). Seguidamente, debemos crear una cámara compatible con XR para ser usada como la visión de las Oculus Quest 2. Par ello clickamos en la cámara que queramos convertir y le añadimos el componente ”XR Rig”, además de dos objetos con ”Tracked Pose Driver” para los controladores. Cabe decir que al añadir el paquete de Oculus se nos pregunta si queremos crear la XR Rig automáticamente, podemos decir que si y luego establecerla en la posición que queramos. Por último, debemos tener en cuenta que para conseguir controlar el movimiento con los joysticks de los controladores del sistema VR, tenemos que indicarlos en los scripts ”Joy Axis Reader” que se usan para leer los ejes de los joysticks (5.4). Cambiamos los nombres ”Horizontal” y ”Vertical” que aparecen como parámetros (referidos a los ejes horizontal y vertical de las teclas del ordenador), por ”Oculus_CrossPlatform_SecondaryThumbstickHorizontal” y ”Oculus_CrossPlatform_SecondaryThumbstickVertical” que reciben los datos de los ejes horizontal y vertical del joystick del controlador derecho de las Oculus (*Map Controllers*, s.f.).

En este momento ya está todo preparado para poner a funcionar nuestra interfaz de realidad

virtual. Encendemos las gafas VR y las conectamos mediante el cable de alta velocidad al ordenador. Previamente debe estar instalado el software explicado en 5.1, (*Oculus Link*, s.f.), para que el PC pueda detectarlas y hacerlas funcionar con Unity. Cabe destacar que en las últimas actualizaciones del software de las gafas se ha introducido una nueva funcionalidad que permite realizar este proceso de forma inalámbrica mediante una red wifi 5G, se ha probado de esta manera y también funciona correctamente por lo que puede llevarse a cabo de cualquiera de las dos formas. Entonces, accedemos a la configuración en las Oculus y damos en la opción Oculus Link si se está realizando con cable, o en la opción Air Link si es por wifi. En ese momento se conectará al ordenador y entrará en el entorno virtual Oculus Link desde el que se puede ver el escritorio del PC y controlar el ratón mediante los controladores VR. De esta forma, podemos acceder a Unity en este entorno y controlarlo desde ahí (figura 5.16). En la imagen vemos el mensaje de advertencia que indica que mi ordenador no cumple las especificaciones mínimas para la realidad virtual. Aunque para la realización de esta interfaz no ha sido un impedimento, en cuanto al rendimiento y eficiencia de la misma si que hay que tenerlo en cuenta como se explicará en la sección 6.2.

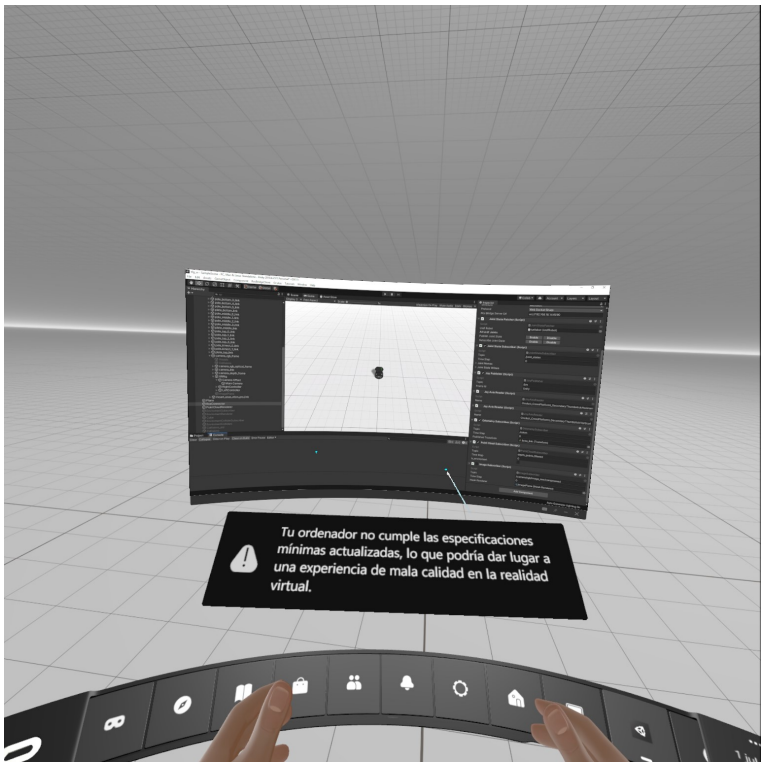
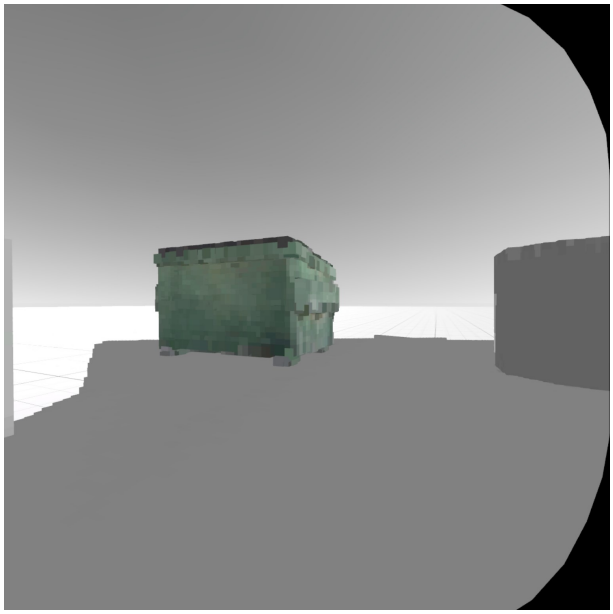


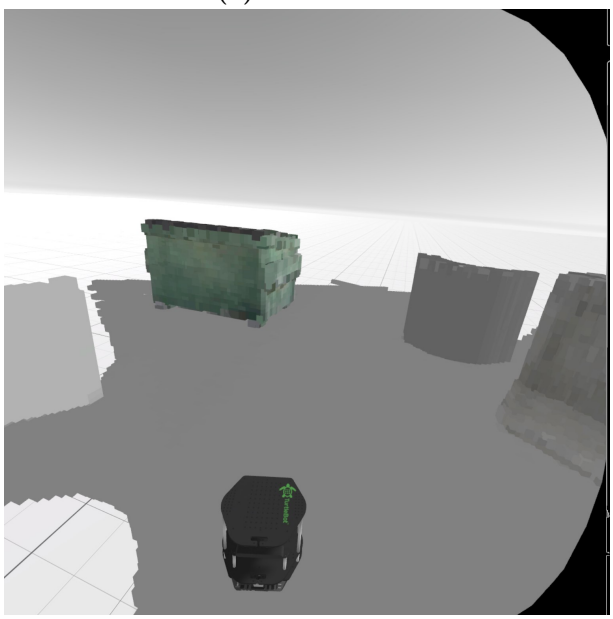
Figura 5.16: Entorno virtual Oculus Link para controlar el PC desde la realidad virtual.

Una vez tenemos el sistema VR preparado, podemos proceder a ejecutar la simulación del entorno del turtlebot en el esclavo (5.5), así como el scrit ”pov.cpp” (5.6) si queremos ver el punto de vista del robot en tiempo real, o los ficheros para el mapeado del entorno (5.7 y 5.8). Estas dos funcionalidades pueden ejecutarse a la vez y funcionar de manera conjunta. Entonces, desde la ventana de Unity en la realidad virtual (5.16) damos al botón play y se

ejecutará la interfaz donde veremos de manera inmersiva la percepción del robot y podremos controlar sus movimientos desde los controladores de las gafas. El resultado final de la interfaz desarrollada se muestra en la figura 5.17 donde vemos la visión del robot en 1ª y 3ª persona en la realidad virtual.



(a) 1ª Persona.



(b) 3ª Persona.

Figura 5.17: Visión final de la interfaz de realidad virtual desarrollada.

6 Experimentación

Una vez diseñada e implementada la interfaz de teleoperación inmersiva de realidad virtual, podemos trabajar en optimizar su funcionamiento, ajustando y mejorando ciertos aspectos. En esta sección se van a mostrar ciertas experimentaciones que se han llevado a cabo para lograr este objetivo, junto a sus resultados y conclusiones.

6.1 Reducción de la nube de puntos 3D

Como se ha explicado en el apartado 5.6.2, para realizar la reducción de la nube de puntos obtenida por la cámara 3D se utiliza un algoritmo de voxelgrid perteneciente a la PCL (*Point Cloud Library*, s.f.). Este algoritmo ha sido elegido ya que es con el que mejores resultados se obtuvo, pero se utilizaron y probaron otros métodos que vamos a explicar y comparar con este.

En esencia, todos los métodos usados se basan en el mismo análisis, el de voxelgrid. El vóxel (volumetric pixel), es la unidad cúbica que compone un objeto tridimensional. Constituye la unidad mínima procesable de una matriz tridimensional y es, por tanto, el equivalente del píxel en un objeto 2D. Los algoritmos de voxelgrid para reducir nubes de puntos pretenden crear una red de cubos 3D o vóxeles donde cada uno de ellos contenga un número definido de puntos vecinos entre sí, de manera que se pueda aproximar la posición del centro y reducir la nube a una nube más pequeña, con un menor número de puntos. Cabe decir que la aproximación del centro del cubo puede hacerse de dos maneras, simplemente extrayendo su centro geométrico, o aproximándola con el centroide de los puntos que lo forman. Esta última es más precisa y es la que se usa en este trabajo. En la figura 6.1 vemos el funcionamiento de este procedimiento, cada punto de la nube es agrupado con sus vecinos más cercanos en un vóxel grande y de este, se aproxima el centroide obtenidendo un sólo punto a partir de los demás. De esta forma se reduce la nube en gran medida.

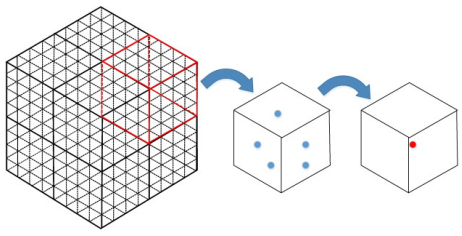


Figura 6.1: Funcionamiento de los algoritmos de voxelgrid.

Dentro de estos algoritmos para la reducción de nubes de puntos se encuentran los octrees. Son estructuras en ”árbol” en las cuales cada nodo interno tiene exactamente 8 ”hijos”. Las estructuras octree se usan mayormente para partir un espacio tridimensional, dividiéndolo recursivamente en ocho octantes pero también pueden ser usadas a la inversa, como es el caso, donde se crea un nodo grande a partir de sus 8 ”hijos”. Estos algoritmos también se basan en voxelgrid ya que necesitan crear una red de vóxeles a partir de la nube para después ir agrupando cada vóxel con sus 7 ”hermanos”. Este procedimiento pueden ser más preciso que utilizando únicamente voxelgrid por lo que se compararán entre sí. El funcionamiento de los octrees se muestra gráficamente en la figura 6.2 dónde vemos como 8 ”hijos” (amarillos) se agrupan en un vóxel ”padre” que a su vez puede ser agrupado de nuevo con sus 7 ”hermanos” (rojos) en otro vóxel más grande, reduciendo así la nube de puntos.

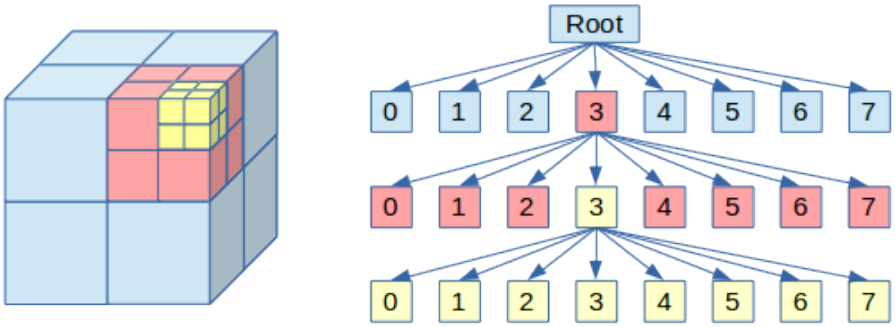


Figura 6.2: Funcionamiento de los algoritmos que utilizan estructuras octree.

A continuación indicaremos los algoritmos usados para la reducción de la nube de puntos. Explicaremos la programación seguida en cada método y compararemos sus resultados de forma analítica y visual, exponiendo los datos obtenidos y las imágenes con las nubes correspondientes para ciertos tamaños de hoja o resolución en el caso de los octrees. En estas imágenes se podrá apreciar cómo los algoritmos consiguen reducir la nube de puntos, en contraposición a la nube original de la Kinect mostrada en la figura 6.3, la cual cuenta con 307200 puntos.

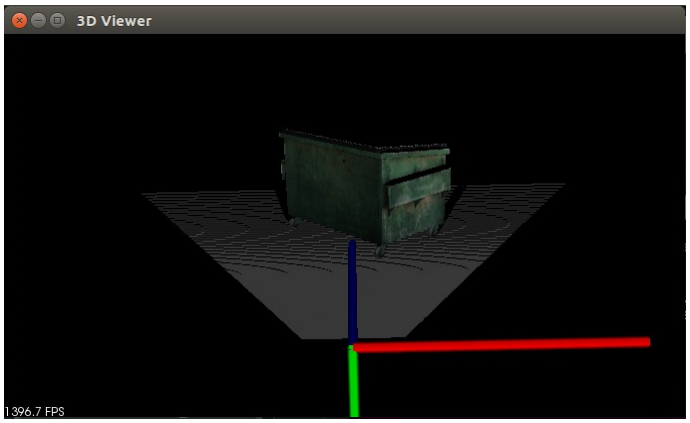


Figura 6.3: Nube de puntos original recibida del sensor 3D. 307200 puntos.

PCL Voxelgrid: en este caso se hará uso de la clase VoxelGrid de la PCL. Creamos un dato de este tipo y, tras haber convertido la nube proveniente del mensaje del topic a una PointCloud2 de la PCL, la introducimos a la clase VoxelGrid usando la función setInputCloud(). Entonces, establecemos el tamaño de las hojas (que será el tamaño de los vóxeles y por tanto, cuántos puntos caben en cada uno) y lo aplicamos a la nube, obteniendo la nube reducida.

PCL Octree: en este método se utilizará la clase OctreePointCloud de la PCL. De la misma forma que en el anterior, convertimos la nube de puntos recibida y la agregamos a esta clase. En ese momento la nube se clasifica en nodos ”hjos” y ”padres” siguiendo los criterios de los octrees explicados y podemos proceder a recorrer cada nodo del árbol. Para cada nodo pasamos un filtro VoxelGrid como el del algoritmo anterior de manera que se voxeliza nodo por nodo, logrando reducir la nube original.

Octomap Octree: estas reducciones también pueden lograrse mediante paquetes de ROS sin necesidad de acudir a librerías externas como la PCL. Octomap (*Octomap*, s.f.) es un paquete para mapeo basado en redes de ocupación 3D, el cual contiene clases como las comentadas en otros métodos, que pueden realizar el trabajo requerido. A la clase de Octomap OcTree podemos agregarle la nube de puntos y, estableciendo la posición del origen del sensor 3D, y la resolución correcta, obtenemos buenos resultados. Como se ve en la tercera fila de la tabla 6.1, la reducción de la nube de puntos es buena pero está en blanco y negro. Esto es debido a que la clase OcTree no cuenta con información RGB. Aunque existe otra clase llamada ColorOcTree que si la tiene, no la implementa en algoritmos como este de reducción de nubes de puntos, por lo que obtenemos el mismo resultado.

Octomap Octree + PCL: para resolver el problema del color del método anterior y seguir utilizando clases de Octomap, podemos realizar una combinación entre estas y un tratamiento de las nubes de puntos usando la PCL. Para ello creamos un dato de tipo ColorOcTree y, mediante funciones de la PCL, recorremos la nube de puntos añadiendo la información x, y, z, r, g, b, a cada nodo del OcTree. Según la resolución del OcTree que hayamos establecido obtendremos más o menos reducción.

Primero, comparamos los resultados gráficamente con las nubes mostradas en la tabla 6.1 para distintos parámetros de tamaño de hoja o resolución. En la primera fila se ven las nubes obtenidas por el algoritmo PCL Voxelgrid, en la segunda las de PCL Octree, en la tercera las dadas por el método Octomap Octree, y en la última fila, las nubes provenientes de Octomap Octree + PCL. En general, podemos decir que, aunque existen pequeñas diferencias en el número de puntos obtenido o la posición de estos, todos los métodos funcionan correctamente ya que consiguen cumplir con su objetivo, que es el de reducir la nube de puntos original sin perder información importante (a excepción de Octomap Octree que no procesa la información RGB). No obstante, estos resultados son solo visuales, en cuanto a términos de rendimiento hay algoritmos que funcionan mejor que otros. Para estudiar esto, en la tabla 6.2 comparamos la cantidad de puntos conseguida en la reducción, y el tiempo de procesamiento para cada método. De esta manera podremos comprobar la eficiencia y/o eficacia de cada algoritmo.

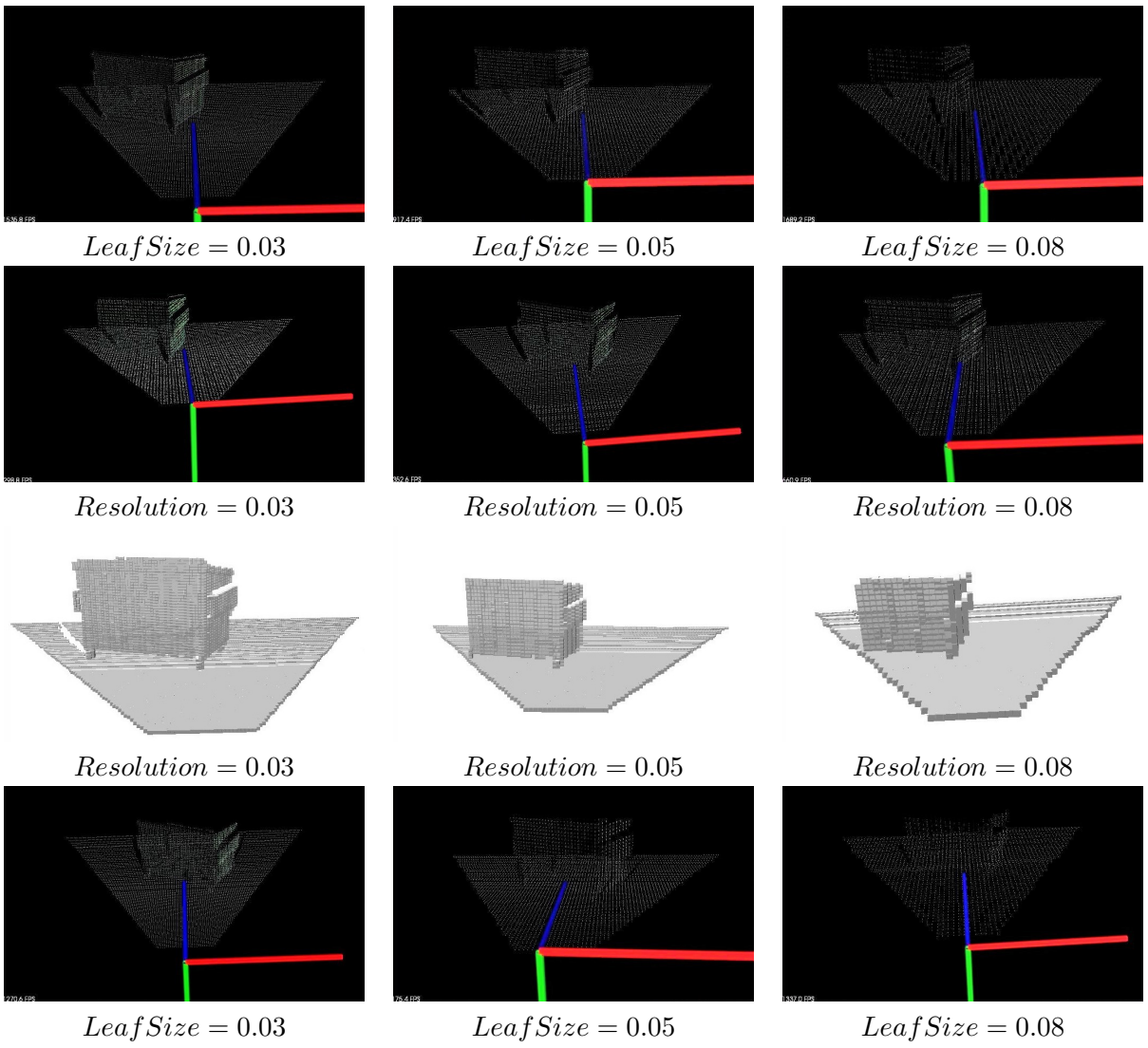


Tabla 6.1: Comparativa de las nubes de puntos obtenidas por cada método.

Analizando los resultados de la tabla 6.2, vemos que los dos peores son los obtenidos por los algoritmos de Octree de la PCL y de Octomap. En cuanto al número de puntos, el de Octomap sí consigue tener coherencia pero tiene la problemática del color blanco y negro y además los tiempos que necesita son demasiado elevados. El de la PCL no consigue reducir la nube de manera proporcional al parámetro introducido y sus tiempos son demasiado altos. Esto puede ser debido a que en este algoritmo se necesita el ajuste tanto del LeafSize como de la resolución del octree ya que, como se ha explicado, se utilizaban ambas clases. Esto puede proporcionar grandes tiempos de procesamiento y dificultad en la adecuación de los parámetros. Por otro lado, vemos que los métodos de VoxelGrid de la PCL y Octomap Octree combinado con procesamiento bajo la PCL, obtienen resultados muy buenos. Los dos reducen la nube de manera correcta pero, aunque en ambos se ven tiempos muy bajos, apreciamos una diferencia considerable a favor del algoritmo VoxelGrid. Dado que la reducción de la nube de

LeafSize / Resolution	Algoritmo utilizado			
	PCL VoxelGrid	PCL Octree	Octomap Octree	Octomap Octree + PCL
0.01	57290 puntos 0.054121 seg	7922 puntos 7.1974 seg	61267 puntos 6.347547 seg	54453 puntos 0.821943 seg
0.03	11655 puntos 0.04633 seg	5300 puntos 6.0235 seg	49413 puntos 5.87445 seg	11413 puntos 0.423117 seg
0.05	5318 puntos 0.040522 seg	5347 puntos 6.7591 seg	16732 puntos 3.36958 seg	5339 puntos 0.402089 seg
0.08	2489 puntos 0.046237 seg	4583 puntos 5.1166 seg	7416 puntos 1.92993 seg	2500 puntos 0.359051 seg
	Nube original: 307200 puntos			

Tabla 6.2: Resultados de la reducción de la nube de puntos original mediante los métodos comentados.

este último es muy buena, y que el tiempo en las tareas de teleoperación de robots es crucial para percibir el entorno de la forma más inmediata posible, decidimos usar este método en el trabajo de reducir la nube de puntos recibida por el sensor 3D, para poder adaptarla de una manera correcta y realista al mundo virtual.

6.2 Ajuste de la malla virtual

Como se explicó en la sección 5.7, las mallas que se obtinen en Unity a partir de la nube de puntos recibida, cuentan con un parámetro llamado ”Point Size” que sirve para determinar el tamaño del cubo que envolverá a cada uno de los puntos de la nube. Este parámetro es imprescindible ya que sin él tan sólo se verían puntos flotando en el espacio en vez de objetos 3D completos. Sin embargo, este dato depende mucho de la cantidad de puntos que tenga la nube, ya que la separación entre ellos determinará el tamaño del cubo que los envuelve, de manera que no queden huecos vacíos en el objeto.

Una vez establecido el algoritmo de reducción de la nube de puntos (que será el que utiliza la clase VoxelGrid de la PCL), debemos estudiar la relación de su tamaño de hoja ”LeafSize”, con el tamaño de los cubos de la malla virtual, ”Point Size”, y decidir qué valores son los más indicados. Para ello, comparamos ciertas combinaciones de estos parámetros con la visión del robot en el mundo real (simulado en gazebo) y establecemos cuáles de ellos generan una visión mas cercana a la realidad. Cabe destacar que en esta decisión también influye la capacidad de procesamiento de nuestro PC ya que, parámetros de tamaño de hoja muy bajos devuelven una gran cantidad de puntos, lo que hace que la representación de la realidad en el mundo virtual sea excelente pero inservible porque la teleoperación no es fluida ni sincronizada con el tiempo real.

En la siguiente imagen (6.4) vemos el entorno real del robot, mientras que en la figura 6.3 aparecen las distintas combinaciones de parámetros que se han ido probando para realizar la representación virtual de esa visión.

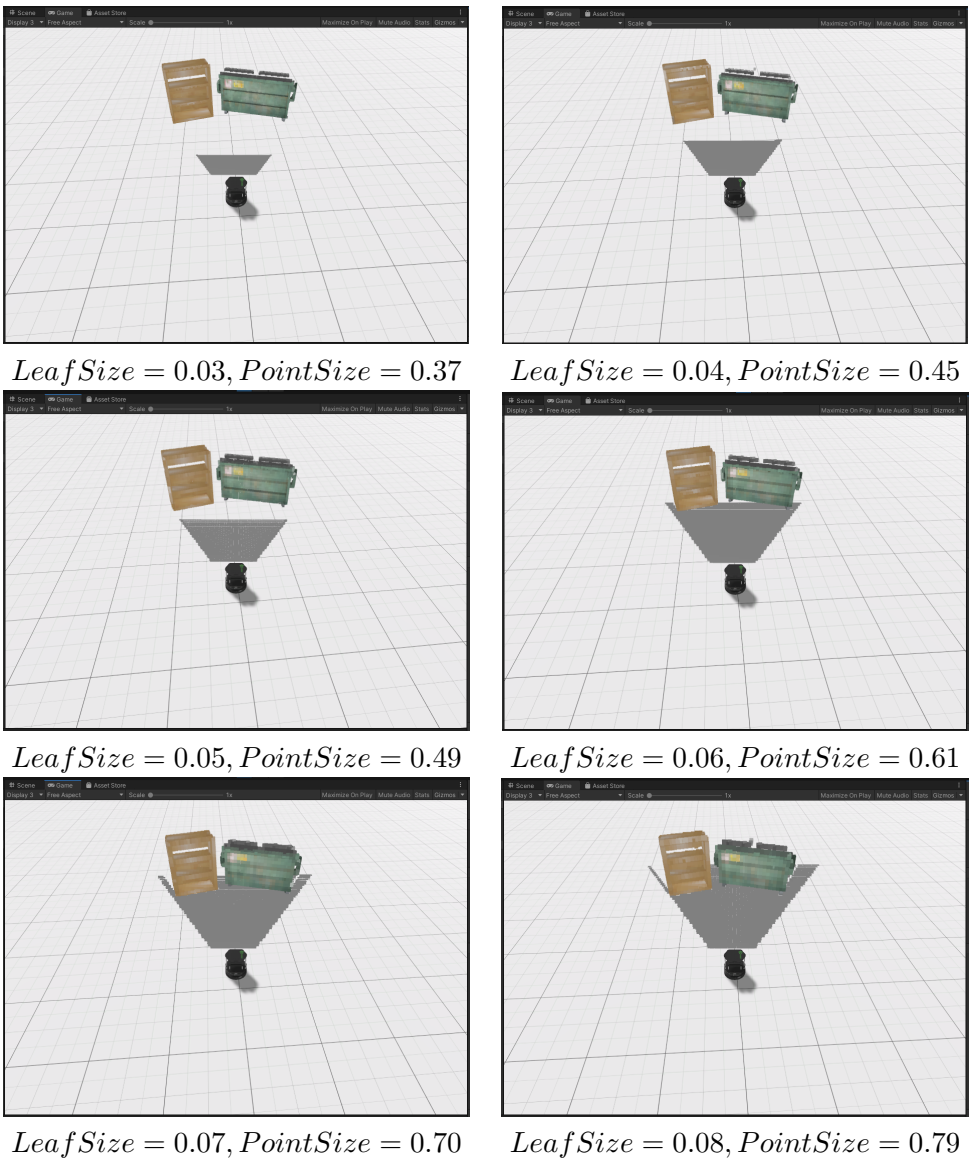
Figura 6.4: Percepción real del robot (simulada en gazebo).

Con esta experimentación hemos comprobado el gran desempeño que realiza el parámetro "Point Size". Como vimos en las imágenes de la tabla 5.1, este valor se puede ir ajustando hasta que el resultado final sea lo más parecido posible a la realidad. Por eso, lo que de verdad va a limitar y definir estos valores es la capacidad de procesamiento del ordenador y la fluidez de la teleoperación. En mi caso, a partir de los valores mostrados en la tercera imagen de la tabla 6.3 (Leaf Size= 0:05, PointSize= 0:49) pude comprobar como la teleoperación era correcta y sin casi delay. No obstante, los mejores resultados en relación semejanza a la realidad y fluidez, los obtuve con LeafSize=0.06 y PointSize=0.61, ya que partir de un LeafSize de 0.07 la representación virtual se distorsiona bastante. Con menos de un LeafSize=0.05 el resultado gráfico es excelente pero se pierda la fluidez por lo que no es viable. Como digo, estos resultados son referidos a mi computadora, en un equipo con mayor capacidad de procesamiento pueden ajustarse aún más estos parámetros y conseguir un parecido a la realidad casi idéntico. Aún así, vemos como las representaciones obtenidas en mi ordenador y mostradas en la tabla 6.3 son bastante buenas y perfectamente aplicables a una interfaz de realidad virtual como la de este proyecto.

6.3 Sincronización de movimiento y visión

Ya que tenemos claro cuál es el algoritmo de reducción de la nube de puntos más eficaz, y sabemos los mejores parámetros para la correcta traslación del mundo real al virtual de manera fiel y con una teleoperación inmersiva y fluida, podemos centrarnos en mejorar otros aspectos que pueden pasar desapercibidos. Haciendo pruebas con las gafas VR, comparando la visión y el movimiento que se obtienen en ellas con los que realiza el robot real, me di cuenta que, aunque el movimiento del Turtlebot se da de manera correcta en la realidad virtual, la visión del entorno correspondiente a esa posición iba ligeramente desincronizada con la real. Profundizando en este aspecto saqué las siguientes conclusiones y lo solucioné de la forma que se va a explicar.

La posición del robot es actualizada en el mundo virtual por el mensaje recibido del topic



$LeafSize = 0.03, PointSize = 0.37$	$LeafSize = 0.04, PointSize = 0.45$
$LeafSize = 0.05, PointSize = 0.49$	$LeafSize = 0.06, PointSize = 0.61$
$LeafSize = 0.07, PointSize = 0.70$	$LeafSize = 0.08, PointSize = 0.79$

Tabla 6.3: Comparativa de la vista del robot en el entorno virtual, dependiendo de los parámetros "LeafSize" y "PointSize".

"/odom" que proviene directamente del robot, mientras que la nube de puntos del sensor 3D requiere un procesamiento anterior a ser publicada para reducirla. Aunque hemos visto que el tiempo para realizar este proceso es mínimo (unos 0.04 segundos), este se suma al tiempo que necesita Unity para procesarla y mostrarla, que es mayor al que se requiere para modificar la posición del robot. Además, con ayuda del comando de ROS "rotopic hz" vemos que el Turtlebot publica ambos mensajes a distinta frecuencia, el topic "/odom" es publicado a una frecuencia máxima de 100 Hz, mientras que el topic "/camera/depth/points", con la nube de puntos de la Kinect, se publica a una frecuencia máxima de 20 Hz. Esta es una diferencia bastante grande y el motivo principal del problema, la posición se publica más rápidamente

que lo que el robot está viendo, de manera que en la realidad virtual el movimiento va adelantado a la visión. Para resolver esto, en el script que reduce y publica de nuevo la nube de puntos ”pov.cpp”, sincronizamos ambos topics para recibirlos y enviarlos al mismo tiempo. Esto se logra gracias al paquete de ROS ”message_filters” (*message_filters*, s.f.) y el funcionamiento de este se explica a continuación.

Se crean dos subscriptores de tipo ”message_filters::Subscriber” para subscribirse a ambos topics, y se sincronizan utilizando la función ”sync.registerCallback()”. Esta recibe por parámetros una función ”callback” que será llamada cuándo los dos mensajes se hayan recibido. Entonces, en esta última se realiza la reducción de la nube y al finalizar el proceso se crean dos publishers que publicarán los dos mensajes sincronizados. El topic ”sync_odom” con los datos de la posición del robot, y el topic ”depth_points_filtered” con la nube reducida. Además, haciendo uso del código ”ros::rate()” podemos establecer la frecuencia de publicación de manera que podemos modificarla para estudiar con cuál se obtienen mejores resultados (la dejamos en 20 Hz ya que es la frecuencia máxima de la nube y no podría publicarse más rápido). De esta manera, en Unity nos subscribiremos, además de al topic ”depth_points_filtered” que ya lo hacíamos, al topic ”sync_odom” en lugar de ”/odom” consiguiendo así una mayor sincronización y resolviendo el problema planteado. En la figura 6.5 mostramos el resultado que aparece en la terminal al comprobar la frecuencia de publicación de los dos topics con ”rostopic hz”, una vez sincronizados. Comprobamos que en ambos casos es de 20 Hz.

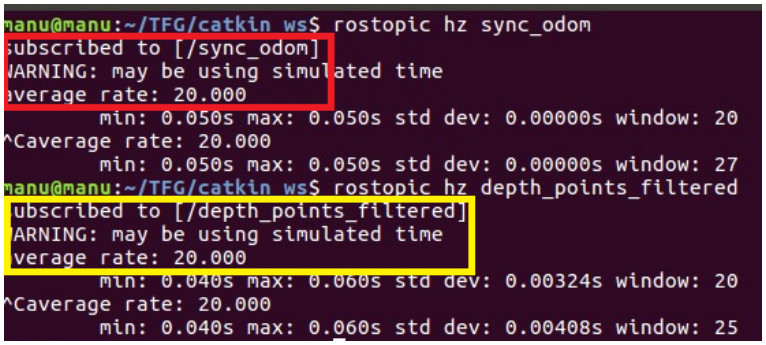


Figura 6.5: Mensaje mostrado por ”rostopic hz” con la frecuencia de publicación de los topics ”sync_odom” y ”depth_points_filtered”.

Aunque la diferencia de tiempo que existe en Unity entre el procesamiento de mostrar la nube de puntos y el de actualizar la posición del robot no se ha solucionado, y, se podría trabajar en ello sincronizando también esta representación virtual. Este desfase es mínimo y, habiendo solucionado el problema de la desincronización de los topics, que era el mayor causante del delay, el funcionamiento final es fluido, sin diferencias de tiempo perceptibles entre lo que sucede en la realidad virtual y el mundo real.

6.4 Funcionamiento en otros entornos

Finalmente, tras desarrollar el sistema de teleoperación y realizar todos los arreglos comentados, comprobamos el funcionamiento del mismo en ambientes más complejos como los que se podría encontrar en una situación real. En este caso utilizaremos el mapa "small-house-world" de Aws Robotics para Gazebo (*Small-House-World*, s.f.). Se trata de una casa en la que se pueden ver elementos típicos de estas, como un sofá, una mesa con sillas o una máquina para hacer pesas (6.6).

Figura 6.6: Mapa Small-House-World de Aws Robotics en gazebo.

Para poder utilizar este mapa en nuestro sistema, debemos añadir el paquete citado (*Small-House-World*, s.f.) a nuestro workspace de ROS, y en el fichero "gazebo_simulation_scene.launch" cambiaremos el mapa establecido por el "small-house" como se indica en 6.1. Debemos tener en cuenta que en este nuevo mapa no existe ningún robot por defecto por lo que debemos añadir los modelos de la estructura y sensores del Turtlebot2 al fichero "small_house.launch" para poder teleoperarlo.

Código 6.1: Introducción del mapa Small-House en el sistema.

```
1    <include file="$(find aws_robomaker_small_house_world)/launch/small_house.↩
         ↪ launch"/>
```

Una vez cambiado el mapa, podemos probar el funcionamiento ejecutando el sistema como se explica en 5.6 para tener una visión del punto de vista del robot, o como se explica en 5.8

par percibir el entorno de la casa completo. Este procedimiento puede realizarse con cualquier otro mapa y comprobar así el funcionamiento de nuestro sistema de realidad virtual en otros entornos y con elementos de características distintas.

6.5 Demostración

Por último, se adjunta un enlace a un vídeo demostrativo donde podemos ver el funcionamiento de este sistema de teleoperación:

`https://drive.google.com/file/d/1lGygR6IX3dSZLLLiXaBxDUIHfqUziJQq/view?usp=sharing`

En esta prueba se utiliza el mapa comentado en el apartado anterior (6.4), y se muestran simultáneamente el movimiento del robot por el entorno, al operador teleoperando al robot con el sistema VR, y la visión que tiene este en la realidad virtual (figura 6.7).

Figura 6.7: Video demostrativo.

Cabe decir que el modo de visión por parte del operador puede ser modificado: primera o tercera persona, distintos puntos de vista, cámara libre o fija, etc. Aunque la cámara fija puede ser interesante para percibir exactamente lo que está viendo el robot en ese momento, en este caso, para tener un mayor campo de visión, la cámara está en tercera persona y se trata de una cámara libre, es decir, el operador puede girar la cabeza en cualquier momento para mirar el entorno sin necesidad de que el robot se mueva. Esta funcionalidad puede apreciarse en momentos del vídeo como el mostrado en la figura 6.8, donde el robot permanece inmóvil pero el operador está mirando a otro punto del entorno para tener una percepción más clara

de su posición. En la imagen vemos como el robot está apuntado hacia la mesa con sillas azules, pero el operador puede girarse y mirar detrás de él, donde se encuentran las sillas naranjas y la máquina de pesas.

Figura 6.8: Visión del operador con cámara libre.

Además de esta funcionalidad, en el vídeo también se muestra el control del movimiento del robot por parte del operador utilizando el joystick del controlador derecho, y como resulta, a efectos finales, la traslación del mundo real al entorno virtual. En relación a esto, es importante decir que, como se ha comentado en otras secciones, la potencia de los ordenadores en los que se esté ejecutando el sistema es muy importante. En un PC más potente podría procesarse un mayor número de puntos por lo que la visión del mundo virtual sería mucho más parecida a la real.

7 Conclusiones

Para concluir este proyecto debemos recapitular hasta los objetivos establecidos al principio del mismo (3). En cuanto a los generales, podemos decir que se han conseguido ampliamente. Para el desarrollo de este trabajo hemos tenido que utilizar distintos lenguajes de programación como Python y C++ en la implementación del sistema esclavo, y C# y ShaderLab para la traslación del mundo real al virtual en Unity. Aunque gracias a tareas anteriores, el conocimiento de los dos primeros era bueno, se ha logrado aprender más con el uso de nuevas librerías y paquetes. En cuanto a los dos últimos, se han utilizado por primera vez en este proyecto y se han desarrollado nuevos conocimientos que, aunque están enfocados a la programación de videojuegos, son realmente útiles en sistemas de este tipo, aplicados a la robótica y la realidad virtual. Por último, de la misma manera que en Python y C++, en el uso de ROS ya se tenían conocimientos previos pero este proyecto ha servido para afianzarlos y crear nuevos, gracias a la utilización de paquetes y funciones que no se habían tratado hasta ahora.

Finalmente, centrándonos en los objetivos específicos, se ha llevado a cabo el diseño y la implementación de algoritmos que permiten llevar el mundo real al virtual, desarrollando técnicas para la representación del movimiento y las colisiones, además de las conseguidas para la reducción de nubes de puntos y su aplicación y exposición en la realidad virtual. De esta manera, se ha conseguido una muy buena reproducción virtual del comportamiento y percepción del robot, lo que ha permitido crear, satisfaciendo además las motivaciones personales expuestas, un sistema de teleoperación de robots móviles en el que se utiliza una interfaz inmersiva de realidad virtual, para trasladar al operador, sin moverse de su espacio de trabajo, al lugar en el que se encuentra el dispositivo.

8 Trabajos futuros

Una vez completado el proyecto, es interesante comentar los siguientes pasos que se podrían seguir en la continuación y perfeccionamiento de este. Algunos de ellos ya se han comenzado a investigar pero su aplicación no ha sido lo suficientemente profunda como para añadirlos en la memoria.

En primer lugar, de manera lógica e imprescindible, se debería probar el sistema con un robot real. Aunque su funcionamiento debe ser el mismo que en simulación, es necesario realizar pruebas con robots reales antes de continuar con el desarrollo de la interfaz. En segundo lugar, como se comenta en las experimentaciones (6.3), un posible trabajo futuro sería ahondar en la sincronización del movimiento en el entorno virtual y la representación de la visión del robot. Aunque, como se explica, se ha eliminado el problema prácticamente al completo, aún puede pulirse más para conseguir un resultado perfecto. También podría ser interesante la realización de un menú en la realidad virtual en el que el operador pudiese modificar ciertas variables que mejoren la percepción y/o el movimiento, cambiar de primera a tercera persona o activar y desactivar el entorno completo. Por último, como posibles tareas futuras, podría trabajarse en la introducción de nuevas funcionalidades en la interfaz, de manera que se facilite el trabajo al operador. Por ejemplo, profundizar en el estudio del paquete de ROS "rtabmap" y las técnicas de SLAM para estimar la trayectoria del robot en el entorno o la detección de objetos. O la agregación de nuevas características en la realidad virtual, como controlar el movimiento del robot simplemente con el movimiento de la cabeza.

Bibliografía

Allard, J., Gouranton, V., Lecointre, L., Limet, S., Melin, E., Raffin, B., y Robert, S. (2004). Flowvr: A middleware for large scale virtual reality applications. *Laboratoire ID, LIFO, Université d'Orléans*.

Atria innovation [La robótica colaborativa en entornos peligrosos]. (2020, 10 de Noviembre). AtriaInnovation. Descargado de `https://www.atriainnovation.com/la-robotica-colaborativa-en-entornos-peligrosos/`

Ballesteros, S. A. (2017). Sistema de teleoperación mediante una interfaz natural de usuario. *Trabajo Final de Grado (TFG), Universidad Carlos III de Madrid*.

Basañez, L., y Suárez, R. (2009). Teleoperation. En S. Y. Nof (Ed.), *Springer handbook of automation* (pp. 449–468). Berlin, Heidelberg: Springer Berlin Heidelberg. Descargado de `https://doi.org/10.1007/978-3-540-78831-7_27` doi: 10.1007/978-3-540-78831-7_27

Bourguet, M.-L. (2003). Designing and prototyping multimodal commands. *Human-Computer Interaction, IOS Press*, 717-720.

Chen, J. Y. C., Haas, E. C., y Barnes, M. J. (2007). Human performance issues and user interface design for teleoperated robots. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *37*(6), 1231-1245. doi: 10.1109/TSMCC.2007.905819

Cincodias [Emprendedores españoles crean una araña robótica para zonas de desastres]. (2015, 9 de Septiembre). CincoDias. Descargado de `https://cincodias.elpais.com/cincodias/2015/09/08/emprendedores/1441748986_582559.html`

Codd-Downey, R., Forooshani, P. M., Speers, A., Wang, H., y Jenkin, M. (2014). From ROS to unity: Leveraging robot and virtual environment middleware for immersive teleoperation. En *2014 ieee international conference on information and automation (icia)* (p. 932-936). doi: 10.1109/ICInfA.2014.6932785

Components of turtlebot 2. (s.f.). researchgate.net. Descargado de `https://www.researchgate.net/figure/Components-of-Turtlebot-2_fig2_322656676`

Cousins, S. (2010). Welcome to ros topics [ros topics]. *IEEE Robotics Automation Magazine*, *17*(1), 13-14. doi: 10.1109/MRA.2010.935808

Fang, H., Ong, S., y Nee, A. (2014). A novel augmented reality-based interface for robot path planning. *Int J Interact Des Manuf*.

Félix Díez, J. J. R. y. A. B. (2016). Modelado y anÁlisis de misiones multi-robot mediante minerÍa de procesos. *Actas de la XXXVII jornada de automática. Comité español de automática*.

Gaschler, A., Springer, M., Rickert, M., y Knoll, A. (2014). Intuitive robot tasks with

augmented reality and virtual obstacles. En *2014 ieee international conference on robotics and automation (icra)* (p. 6026-6031). doi: 10.1109/ICRA.2014.6907747

ROS kinetic [Software para la programación de robots]. (s.f.). Descargado de `http://wiki.ros.org/kinetic/Installation/Ubuntu`

Grifoni, P. (2009). Multimodal human computer interaction and pervasive services. *Information Science Reference*, 1959.

Hansberger, J. T. (2015). Development of the next generation of adaptive interfaces. *Human Research and Engineering Directorate*.

haptx.com. (s.f.). Haptx. Descargado de `https://haptx.com/virtual-reality/`

How to link unity3d and gazebo for robot control. (s.f.). Dvic. Descargado de `https://dvic.devinci.fr/resource/tutorial/unity-rosbridge/`

Lin, A., Milshteyn, A., Herman, G., Garcia, M., Liu, C., Rad, K., ... Boussalis, H. (2014). Virtual reality head-tracking observation system for mobile robot. En *2014 3rd mediterranean conference on embedded computing (meco)* (p. 152-157). doi: 10.1109/MECO.2014.6862681

Map controllers [Mapeo de los botones de los controladores Oculus Quest 2 para Unity]. (s.f.). Oculus. Descargado de `https://developer.oculus.com/documentation/unity/unity-ovrinput/`

message_filters [Paquete de ROS para la aplicación de filtros de tiempo a mensajes.]. (s.f.). Descargado de `http://wiki.ros.org/message_filters`

Montalvo, W., Garcia, C. A., Naranjo, J. E., Ortiz, A., y Garcia, M. V. (2019). Sistema de tele-operación para robots móviles en la industria del petróleo y gas. *Revista Ibérica de Sistemas e Tecnologias de Informação*.

Needleman, S. E. (2018). Virtual reality, now with the sense of touch. *THE FUTURE OF EVERYTHING*.

Octomap [Paquete de ROS para creación de mapas a partir de redes de ocupación 3D.]. (s.f.). Descargado de `http://wiki.ros.org/octomap`

Oculus integration for unity [Paquete oficial para la integración de las gafas de realidad virtual Oculus en proyectos de Unity]. (s.f.). Descargado de `https://assetstore.unity.com/packages/tools/integration/oculus-integration-82022`

Oculus link [Software oficial de Oculus para la conexión de gafas de realidad virtual con un ordenador]. (s.f.). Descargado de `https://www.oculus.com/setup/`

Oculus quest 2. (s.f.). Wikipedia. Descargado de `https://en.wikipedia.org/wiki/Oculus_Quest_2`

Oculus youtube. (s.f.). Oculus. Descargado de `https://www.youtube.com/watch?v=P_V68E-hnZc&t=50s`

Point cloud library [Librería de código abierto para el procesamiento de imágenes 2D, 3D y nubes de puntos.]. (s.f.). Descargado de `https://pointclouds.org/downloads/#linux`

Pointclound streaming from ros. (s.f.). github. Descargado de `https://github.com/inmo-jang/unity_assets/tree/master/PointCloudStreaming`

Raymond goertz. (s.f.). Wikipedia. Descargado de `https://en.wikipedia.org/wiki/Raymond_Goertz`

Robot operating system. (s.f.). Wikipedia. Descargado de `https://es.wikipedia.org/wiki/Robot_Operating_System`

rosbridge. (s.f.). ros.org. Descargado de `http://wiki.ros.org/rosbridge_suite`

roscomponents.com. (s.f.). roscomponents. Descargado de `https://www.roscomponents.com/es/robots-moviles/9-turtlebot-2.html`

ros.org. (s.f.). ros.org. Descargado de `https://www.ros.org/about-ros/`

Ros-sharp [Librería de código abierto para la comunicación de aplicaciones .NET en lenguaje C-Sharp (En especial Unity) con ROS]. (s.f.). Descargado de `https://github.com/siemens/ros-sharp`

rtabmap_ros [Paquete de ROS para proyectos de SLAM RGB-D con restricciones en tiempo real.]. (s.f.). Descargado de `http://wiki.ros.org/rtabmap_ros`

Small-house-world [Mapa de código abierto para su uso en gazebo y otras plataformas]. (s.f.). Aws-Robotics. Descargado de `https://github.com/aws-robotics/aws-robomaker-small-house-world`

Tapia, E. P. (2017). Interfaz inmersiva para misiones robóticas. *TFG, Universidad Politécnica de Madrid.*

Turtlebot2 datasheet. (s.f.). roscomponents. Descargado de `https://www.roscomponents.com/es/robots-moviles/9-turtlebot-2.html`

turtlebot.com. (s.f.). turtlebot. Descargado de `https://www.turtlebot.com/about/`

Understanding ros 2 topics. (s.f.). ROS. Descargado de `https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html`

Unity [Software para el diseño y la programación de videojuegos]. (s.f.). Descargado de `https://unity3d.com/es/get-unity/download`

Unity3d. (s.f.). Unity3d. Descargado de `https://unity3d.com/es/how-to/cutscenes-cinematics-with-timeline-and-cinemachine`

Unity (motor de videojuegos). (s.f.). wikipedia.org. Descargado de `https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego)`

Unity xr plug-in. (s.f.). Unity3d. Descargado de `https://docs.unity3d.com/Manual/XRPluginArchitecture.html`

Lista de Acrónimos y Abreviaturas

2D	Two-Dimensional.
3D	Tri-Dimensional.
API	Application Programming Interface.
AR	Augmented Reality.
GPU	Graphics Processing Unit.
IP	Internet Protocol.
JSON	JavaScript Object Notation.
MR	Mixed Reality.
OpenGL	Open Graphics Library.
OpenGL ES	OpenGL for Embedded Systems.
PC	Personal Computer.
PCL	Point Cloud Library.
POV	Point of View.
RGB	Red Green Blue.
RGB-D	Red Green Blue Depth.
ROS	Robot Operating System.
SDK	Software Development Kit.
SLAM	Simultaneous Localization and Mapping.
STAIR	Stanford Artificial Intelligence Robot.
TCP-IP	Transmission Control Protocol - Internet Protocol.
TFG	Trabajo Final de Grado.
URDF	Unified Robot Description Format.
USB	Universal Serial Bus.
VR	Virtual Reality.
XML	Extensible Markup Language.
XR	Extended Reality.