ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

Herramientas robóticas con tecnologías web en la plataforma Jderobot

Autor: Roberto Pérez González

Tutor: José María Cañas Plaza

Curso académico 2018/2019

Agradecimientos

Quien me conoce sabrá que esta probablemente sea una de las partes más difíciles para mí debido a mi personalidad, sin embargo esta ocasión merece que realice el esfuerzo.

En primer lugar agradecer a mi tutor José María por su dedicación y su ayuda durante los meses que ha durado el desarrollo de este proyecto.

Dar las gracias a mi hermano y, en especial a mis padres. A mi padre, por trabajar duro durante toda su vida para que yo pudiera tener una vida mejor y tener la mejor educación posible, y a mi madre, por siempre estar a mi lado y apoyarme hiciera lo que hiciera. Nunca me habéis presionado y habéis permitido que yo mismo tomara mis propias decisiones, fueran buenas o malas.

No quiero olvidar a mis compañeros de trabajo. A mi jefa Rocío por facilitarme compaginar el desarrollo de este proyecto y mi trabajo, además de aconsejarme. A Gonzalo por esas largas tardes que sin ti no hubieran sido lo mismo. Pasamos muchas horas juntos a lo largo del día y me habéis aguantado, que sé que no es tarea fácil, por ello os considero ya mis amigos.

Resumen

La plataforma JdeRobot nació como un paquete de softwares para el desarrollo de aplicaciones robóticas y visión artificial. El propósito de la plataforma es crear herramientas y controladores que faciliten la conexión con componentes hardware. La plataforma esta principalmente desarrollada en C++ y Python.

El propósito de este proyecto es facilitar el uso de las diferentes aplicaciones mediante el uso de tecnologías web en el front-end. Esta tecnología nos permite la ejecución de las diferentes herramientas con independencia del sistema operativo en que se está ejecutando. Además, utilizando el framework Electron, seremos capaces de crear aplicaciones de escritorio evitando de este modo problemas de compatibilidades entre los diferentes navegadores web que hay actualmente.

El desarrollo de este proyecto se realizará utilizando JavaScript, HTML5 y CSV en el cliente, y NodeJS en el servidor, sobre el cual está implementada la tecnología de Electron. Además me apoyare en los middleware ZeroC ICE y ROS para la interconexión cliente-servidor.

La primera parte del proyecto está centrada en la adaptación de los Visores existentes en la plataforma JdeRobot para que funcionen como aplicaciones de escritorio. Estos visores son Turtlebotviz, Droneviz y Camviz. Para finalizar esta parte, me centrare en este último visor para hacerlo funcionar con los dos principales middleware de comunicación: ZeroC ICE (como hasta ahora) y ROS.

La segunda parte, consiste en la creación de un nuevo visor web que mostrara objetos 3D, desde un simple punto o segmento, hasta objetos más complejos que se obtienen a partir de modelos 3D. Una de las funciones de este visor es la de ser utilizado en la práctica de JdeRobot Academy “Reconstrucción 3D”.

El último propósito de este proyecto es la elaboración de un nuevo componente para la plataforma JdeRobot. Este nuevo componente será un servidor de imágenes obtenidas a través de cámaras web y servidas a los diferentes visores. Este componente utiliza como middleware de comunicación ROS.

Índice general

1. Introducción **2**

1.1. Robótica 2

1.1.1. Productos róboticos 3

1.1.2. Software para robots 7

1.2. Tecnologías web 9

1.2.1. Arquitectura de las aplicaciones web 10

1.2.2. Tecnologías del lado del cliente 11

1.2.3. Tecnologías del lado del servidor 12

1.3. Tecnologías web en robótica 13

2. Objetivos **17**

2.1. Objetivos 17

2.2. Metodología 18

2.3. Plan de Trabajo 19

3. Infraestructura **21**

3.1. Robots reales y simulados 21

3.1.1. Robots simulados: Gazebo 22

3.1.2. Robots reales 22

3.2. La plataforma JdeRobot 23

3.3. El Middleware ICE 24

3.4. El Middleware ROS 25

3.5. Robot Web Tools 27

3.6. Node.js 28

3.7. WebGL y Three.js 29
3.8. WebRTC 30
3.9. El entorno Electron 31
3.9.1. Adaptación de una aplicación web a Electron 32
3.9.1.1. package.json 32
3.9.1.2. main.js 33

4. Visores web modificados **35**

4.1. CamVizWeb 35
4.1.1. Diseño 35
4.1.2. Interfaz gráfica 37
4.1.3. Conexiones 38
4.1.3.1. Establecimiento de la conexión 38
4.1.3.2. Estructura de los mensajes 39
4.1.3.3. Subscripción a los *Topic* y visualización de las imágenes . 40
4.1.4. Configuración 41
4.1.5. Experimentos 42
4.2. TurtleBotVizWeb 44
4.2.1. Diseño 44
4.2.2. Interfaz gráfica 46
4.2.3. Conexiones 46
4.2.3.1. Establecimiento de la conexión 47
4.2.3.2. Estructura de los mensajes 48
4.2.3.3. Subscripción a los *Topic* y tratamiento de la información . 50
4.2.3.3.1. Subscripción y tratamientos de las cámaras . . . 50
4.2.3.3.2. Subscripción y tratamiento del sensor láser 50
4.2.3.3.3. Subscripción y tratamiento de la odometría . . . 53

4.2.3.4. Publicación del mensaje con la información del teleoperador 55
4.2.4. Configuración . . . 56
4.2.5. Experimentos . . . 58
4.3. DroneVizWeb . . . 62
4.3.1. Diseño . . . 62
4.3.2. Interfaz gráfica . . . 64
4.3.3. Conexiones . . . 65
4.3.3.1. Establecimiento de las conexiones . . . 65
4.3.3.2. Estructura de los mensajes . . . 65
4.3.3.3. Subscripción a los *Topic* y tratamiento de la información . 67
4.3.3.4. Publicación del mensaje con la información del teleoperador 68
4.3.3.5. Arranque, Despegue y Aterrizaje mediante ROS Services . 71
4.3.4. Configuración . . . 73
4.3.5. Experimentos . . . 75

5. Visor web dinámico de objetos 3D **77**

5.1. Diseño . . . 77
5.2. Configuración . . . 79
5.3. Interfaz gráfica . . . 80
5.3.1. Escena base . . . 80
5.3.2. Visualización de puntos . . . 81
5.3.3. Visualización de segmentos . . . 83
5.3.4. Visualización de objetos 3D . . . 85
5.3.4.1. Creación y visualización de los objetos 3D . . . 85
5.3.4.2. Movimiento de los objetos 3D . . . 89
5.3.5. Borrado de elementos mostrados en el visor . . . 90
5.4. Estructura de los Mensajes . . . 91

5.4.1. Mensaje para visualizar los puntos 92
5.4.2. Mensaje para visualizar segmentos 94
5.4.3. Mensaje para visualizar un objeto 3D 95
5.4.4. Mensaje para mover los objetos 3D 98
5.5. Conexiones 100
5.5.1. Conexión y peticiones a la aplicación 100
5.5.2. Recepción y tratamiento de los mensajes recibidos 105
5.5.2.1. Tratamiento de los puntos 106
5.5.2.2. Tratamiento de los objetos 3D 107
5.5.2.3. Tratamiento del movimiento de los objetos 109
5.6. Experimentos 109

6. Servidor de imágenes con tecnologías web 112
6.1. Diseño 112
6.2. Adquisición y preparación de imágenes 114
6.3. Conexiones 116
6.3.1. Establecimiento de la conexión 116
6.3.2. Estructura de los mensajes 117
6.3.3. Creación de los mensajes y publicación 117
6.4. Configuración 119
6.4.1. Interfaz gráfica 119
6.5. Experimentos 120

7. Conclusiones 123
7.1. Contribuciones 123
7.2. Trabajos Futuros 125

Bibliografia 127

Capítulo 1

Introducción

En este primer capítulo se explica el contexto de las bases tecnológicas en las que se apoya este proyecto, que son principalmente la robótica y las tecnologías web. Para empezar se habla sobre el estado actual de la robótica y la gran expansión de la misma en nuestros días. Se continúa señalando el contexto de las tecnologías web y cómo de importantes son en la sociedad actual, su arquitectura y las tecnologías más relevantes. Para finalizar este capitulo, se introducen proyectos previos que combinan tecnologías web con componentes robóticos.

1.1. Robótica

El diccionario de la Real Academia Española define robótica como *"la técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales"*.

Desde que Asimov acuñara sus tres leyes la sociedad ha temido lo que los robots puedan hacer al ser humano, desde esclavizarnos hasta quitarnos el trabajo, siendo este miedo el más reciente. Sin embargo, con el paso del tiempo, nos hemos dado cuenta que la robótica tiene una gran utilidad para hacer avanzar la sociedad. Ya no solo pensamos en robótica como en un humanoide que pueda reemplazar al ser humano, sino que vemos robótica en nuestro día a día, desde un brazo mecánico en una cadena de montaje hasta un aspirador robótico en casa. Tareas que hasta hoy conllevaban riesgos para la salud humana como la desactivación de artefactos explosivos o trabajos con altas temperaturas,

o tareas pesadas y repetitivas se pueden realizar de manera más eficiente y fácil gracias a su robotización.

1.1.1. Productos róboticos

El uso de la robótica en la actualidad está muy extendido, tareas tales como montajes en cadena en una fábrica, trabajos repetitivos o manipulación de objetos peligrosos ya están robotizados de manera que el ser humano no se ponga en peligro o malgaste tiempo de manera innecesaria. A continuación se muestran varios ejemplos de uso de la robótica en nuestros días:

En cirugía los robots se están empezando a usar en procedimientos quirúrgicos, ya que compensan las deficiencias y limitaciones del ser humano en exactitud y precisión. Además, posibilitan la "telecirugía", la realización de intervenciones quirúrgicas de manera remota mediante brazos robóticos que imitan al milímetro los movimientos de un cirujano situado en cualquier lugar. La primera "telecirugía" fue realizada en el año 2001 por cirujanos estadounidenses que operaron a un paciente en Francia y ha supuesto una revolución en la medicina, al poder realizar intervenciones quirúrgicas separadas por miles de kilómetros. Actualmente, ya se comercializan sistemas como el Da Vinci (Figura 1.1) pensado para mejorar la laparoscopia, que permite al cirujano operar sentado y visualizando en un monitor la imagen del paciente, proporcionando una mayor precisión al evitar los temblores, mejor visualización y mayor destreza.

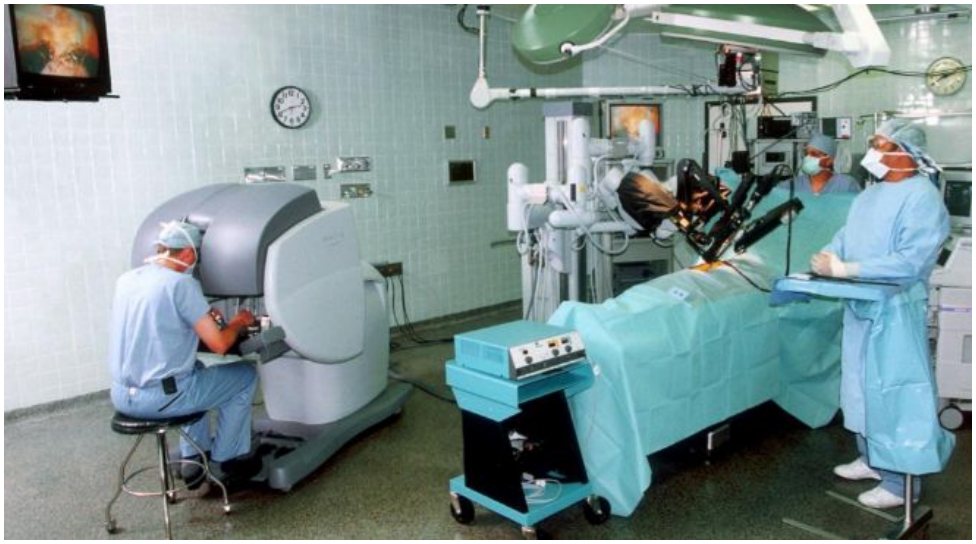


Figura 1.1: Sistema Da Vinci

En industria es el sector donde más extendido está el uso de la robótica (Figura 1.2), desde mover una pieza de posición hasta cargar y descargar máquinas. Hay una donde sobresale por encima del resto: el sector del automóvil. En España, según datos de la Asociación Española de Robótica, seis de cada diez robots pertenecen a este sector. Los robots se encargan de distribuir por toda una fábrica los componentes necesarios, así como montarlos. Gracias a la robotización de la industria, la producción ha aumentado en los últimos años de manera significativa, disminuyendo costes y errores.

Figura 1.2: Fábrica de Seat en Martorell

El sector militar es donde más dinero se invierte y se investiga. Actualmente hay multitud de robots usados militarmente como los vehículos aéreos no tripulados Hermes y Predator, el Goalkeeper CIWS, que es un sistema de armamento defensivo por proximidad holandés, o Samsung SGR-A1, que es un robot centinela utilizado para la vigilancia de la zona desmilitarizada entre Corea del Sur y Corea del Norte. Se está investigando la elaboración de un robot humanoide capaz de caminar por terrenos irregulares y zonas catastróficas para realizar tareas de rescate y ayuda humanitaria. En este sentido, entre el año 2012 y 2015, el Departamento de Defensa de Estados Unidos, a través de su Agencia de Proyectos de Investigación Avanzada en Defensa (DARPA), creó una competición con el objetivo de desarrollar robots terrestres semiautónomos capaces de realizar tareas complejas en entornos peligrosos y degradados[1]. El premio fue ganado por el robot humanoide desarrollado por el Instituto Avanzado de Ciencia y Tecnología de Corea, "DRC-Hubo" (Figura 1.3).

[1] `https://www.darpa.mil/program/darpa-robotics-challenge`

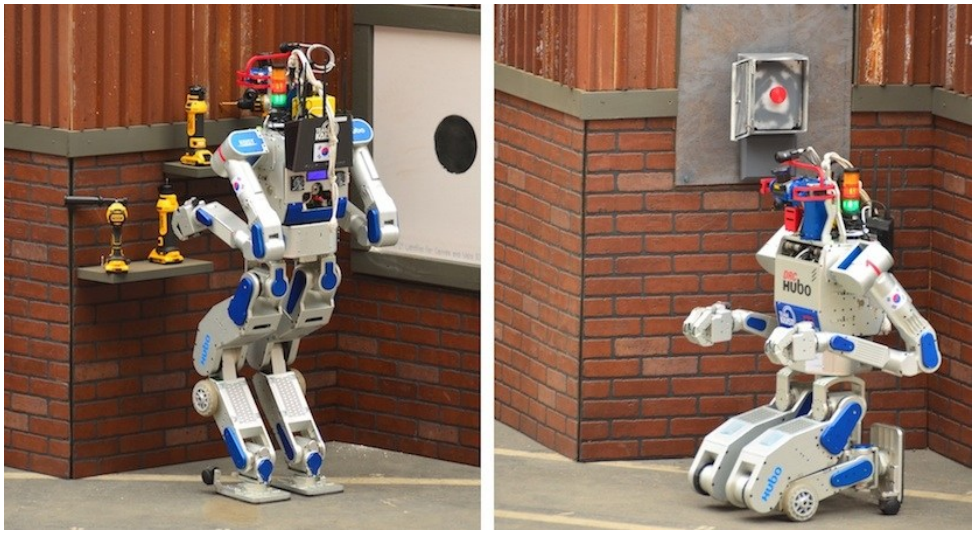


Figura 1.3: Robot DRC-Hubo durante el DARPA Robotics Challenge

Por último, cabe destacar el sector doméstico donde la dómotica está comenzando a despuntar gracias a los altavoces inteligentes como Google Home o Amazon Echo. Sin embargo, si hay un elemento robótico por encima del resto que destaca en el día a día de una vivienda son los robot aspiradores (Figura 1.4). Estos aspiradores son robots autónomos que mejoran la calidad de vida de los ser humanos ayudando a realizar las tareas del hogar, liberando de trabajo a unas vidas lo suficientemente recargadas.

Figura 1.4: Ejemplo de robot aspirador

1.1.2. Software para robots

Todo robot tiene una parte hardware (el componente físico ya sea real o simulado) y una parte software (la lógica). El software proporciona al robot la inteligencia y autonomía necesaria para que realice las funciones o acciones que deseamos. Para facilitar esta tarea, se han creado *middleware* y bibliotecas especificas.

El *middleware* proporciona las herramientas que se necesitan para facilitar el desarrollo de software robótico, ya que permite estructurarlo separando las diferentes tareas (lectura de sensores, extracción de datos, especificar la velocidad, etc.) y haciendo exportable el uso de este software con cualquier otro sistema robótico al tener un marco común de comunicación. Actualmente hay una gran cantidad de *middleware* que permiten esta abstracción, siendo algunos de los más interesantes:

- Robot Operating System (ROS)[2]: Provee de la funcionalidad que cabría esperar de un sistema operativo (abstracción del hardware, control de dispositivos de bajo nivel, la transferencia de mensajes entre procesos, administración de paquetes, etc.). Su principal objetivo es permitir la reutilización del código en la investigación y desarrollo de robótica, publicando paquetes de software que están listos para ser usados por cualquier desarrollador. ROS cuenta con una gran comunidad de colaboradores que comparten sus paquetes y proyectos, gracias a su sitio web y repositorios.

- Open Robot Control Software project (Orocos)[3]: Se trata de un proyecto de software libre cuyo objetivo es la creación de un paquete de software para el control de robots. Desde el principio nace con tres objetivos a respetar: que este basado en componentes, que tenga proveedores múltiples dado que un robot no está construido con piezas de un único proveedor y convertirse en el mejor entorno de software libre para el control en tiempo real de robots y máquinas del mercado.

- JdeRobot[4]: Se trata de una plataforma de código abierto creada en la Universidad Rey Juan Carlos para el desarrollo de aplicaciones de visión artificial y robóticas, compatible con middlewares de comunicación ICE y ROS y desarrolladas principalmente en Python y C++.

[2] `http://www.ros.org/`
[3] `http://www.orocos.org/`
[4] `https://jderobot.org/`

Además del *middleware*, hay multitud de bibliotecas para el desarrollo de sistemas robóticos: bibliotecas para procesar imágenes, interactuar con el robot, procesar elementos 3D, etc. Algunas de las bibliotecas más utilizadas son:

- OpenCV[5]: Es la biblioteca de visión artificial por excelencia. Creada originalmente por Intel, es de código abierto y desarrollada en C++. Actualmente dispone de interfaces en C++, C, Python, Java y están empezando a desarrollar para JavaScript.

- Point Cloud Library (PCL)[6]: Es una biblioteca de código abierto que proporciona algoritmos para el procesamiento de nubes de puntos y geometría 3D.

- Bibliotecas de herramientas y nodos de ROS: ROS proporciona bibliotecas para cada lenguaje de programación al que da soporte, ofreciendo una serie de funciones y algoritmos que permite crear aplicaciones que interactúan rápidamente. Las bibliotecas más utilizadas son `rospy`[7], desarrollada para Python y `roscpp`[8], desarrollada para ser usada con C++. Cabe destacar, también, la biblioteca `roslibjs`, que ofrece funcionalidades para JavaScript.

Finalmente, cabe mencionar los simuladores robóticos, que imitan el funcionamiento de un sistema robótico para probar las aplicaciones, evitando que surjan fallos críticos a la hora de hacerlo funcionar con el sistema real. Estos fallos pueden conllevar averías muy costosas en tiempo y dinero, que conllevarían retrasos importantes a la hora de realizar el desarrollo.

Uno de los simuladores más usados es Gazebo[9] gracias a su fácil manejo y su intuitiva interface. Es un simulador de código abierto que ofrece múltiples motores de físicas, motores de renderizado avanzado, soporte para plug-ins y programación en la nube. Además, dispone de un gran número de robots, sensores y cámaras para simular, lo que permite realizar las pruebas de nuestras aplicaciones de forma bastante realista y, así poder llevar una aplicación robótica al sistema físico sin miedo a que sufra daños.

[5] `https://opencv.org/`
[6] `http://pointclouds.org/`
[7] `http://wiki.ros.org/rospy`
[8] `http://wiki.ros.org/roscpp`
[9] `http://gazebosim.org/`

Existen otros simuladores como Stage[10] para la simulación en 2D o Webots[11].

1.2. Tecnologías web

Desde que en el año 1992 Tim Berners-Lee ideara y desarrollara las primeras herramientas para compartir información entre los científicos del CERN desde cualquier parte del mundo, dando lugar World Wide Web. La sociedad actual no se puede entender sin la existencia de la web. Sin embargo, pese a la gran evolución, el propósito de la web no ha cambiado, es que acceder a la información sea lo más fácil posible. Sí lo ha hecho la manera en que se utiliza. La aparición de aplicaciones web como las redes sociales, los servicios de streaming o los comercios electrónicos ha supuesto un impulso importante en el uso de la web y la necesidad de idear nuevas herramientas que faciliten el desarrollo de webs.

A continuación se muestran algunos ejemplos de aplicaciones y páginas web muy utilizadas:

- Spotify[12]: Aplicación creada para la reproducción de música vía streaming, actualmente cuenta con más de 75 millones de usuarios activos siendo una de las principales formas de escuchar música en la actualidad. La aplicación está desarrollada mediante JavaScript (lado del cliente) y Python (lado del servidor).
- Netflix[13]: Comenzó como un videoclub donde se alquilaban DVDs a través de una página web y se enviaban por correo postal, para pasar a proporcionar un servicio de video bajo demanda por internet. Cuenta con más de 110 millones de usuarios suscritos a su servicio y ha supuesto una revolución al cambiar el modo en que la sociedad ve contenidos audiovisuales, gracias a su facilidad de uso mediante su aplicación web. Este auge ha llevado a Netflix a realizar sus propias producciones.
- Facebook[14]: La red social más conocida y que provocó el auge de las mismas, actualmente cuenta con más de 2200 millones de usuarios. Originalmente nació como

[10] `http://rtv.github.io/Stage/index.html`
[11] `https://www.cyberbotics.com/`
[12] `https://www.spotify.com/es/`
[13] `https://www.netflix.com/es/`
[14] `https://es-es.facebook.com/`

una página web desarrollada mediante HTML y JavaScript para el lado del cliente y PHP para el lado del servidor, ha evolucionado hasta disponer una aplicación que puede ser utilizada en nuestros teléfonos móviles.

- Google maps[15]: Un servicio de mapas que permite desplazarse por el mundo, ver fotografías por satélite o recorrer una ubicación a pie de calle. Actualmente es utilizada para indicar ubicaciones, a modo de GPS o para encontrar lugares de interés. La base de esta aplicación es JavaScript, usando HTML y CSS para la interfaz gráfica.

- YouTube[16]: Página web dedicada a la compartición de vídeos entre usuarios. Actualmente cuenta con más de 1900 millones de usuarios al mes y cada minuto se suben más de 120 horas de vídeo, llegando a convertirse en un oficio para miles de personas que ganan dinero publicando sus propios vídeos comentando videojuegos, consejos o críticas de móviles, películas, juguetes, etc.. YouTube está desarrollado mediante JavaScript y HTML en el lado del cliente y Python en el lado del servidor, incluso proporcionó el primer reproductor de video incrustado en el HTML, que posteriormente fue incluido por el World Wide Web Consortium.

- Whatsapp Web[17]: Es una aplicación de mensajería instantánea que permite el intercambio entre usuarios de mensajes a través de internet sin coste adicional. Cuenta con más de mil millones de usuarios. Ha supuesto una revolución al provocar la extinción del servicio de mensajes cortos (SMS) que conllevaban un coste alto. Tras ser usado como aplicación en teléfonos móviles, en el año 2015 se lanzó la versión web de Whatsapp para su uso en un ordenador, provocando que la aplicación de mensajería de Google, Hangouts, sufriera una reducción importante de usuarios. Es uno de los factores que ha llevado a Google a anunciar el cierre del servicio en el año 2020.

1.2.1. Arquitectura de las aplicaciones web

Desde su creación, el modelo para que un sitio o aplicación web funcione no ha variado (Figura 1.5).

[15] `https://www.google.es/maps`
[16] `https://www.youtube.com/`
[17] `https://www.whatsapp.com/`

- Cliente: Realiza las peticiones de recursos a diferentes servidores web a través de un localizador uniforme de recursos (URL). Generalmente, la función de cliente la realizada un navegador.
- Servidor: Almacena la información de la aplicación web y sirve los contenidos acorde a las peticiones realizadas por el navegador.
- Http: Es el protocolo permite el intercambio de información entre el cliente y el servidor.

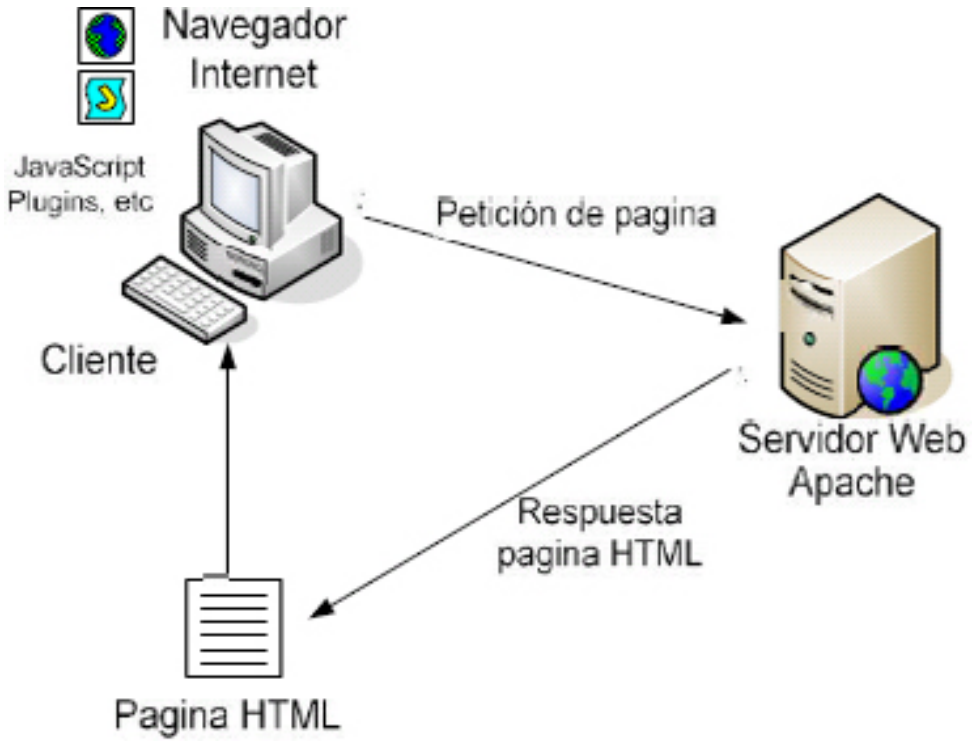


Figura 1.5: Arquitectura de una aplicación web

1.2.2. Tecnologías del lado del cliente

Son las encargadas de dar forma a la interfaz de usuario y de establecer la comunicación con el servidor. El navegador es capaz de leer e interpretar la información recibida. Las más utilizadas son:

- Hyper-Text Markup Lenguage (HTML): Es el lenguaje de descripción de contenidos web que permite especificar las características visuales. Actualmente se utiliza la

quinta revisión llamada HTML5, que aporta una gran mejora como las etiquetas para reproducir contenido multimedia.

- Hojas de estilo en cascada (CSS): Es el lenguaje utilizado para describir la presentación (el aspecto y el formato) de un documento en lenguaje HTML.
- JavaScript: Es un lenguaje de script orientado a objetos y guiado por eventos que permite realizar acciones en el cliente e interactuar con el servidor u otras aplicaciones web.

1.2.3. Tecnologías del lado del servidor

Son las encargadas de dar forma al servidor web de manera que permita el acceso a bases de datos, conexiones de red, recursos compartidos, en definitiva, de realizar todas las tareas necesarias para crear la aplicación web que se visualizará en el cliente. Las tecnologías más utilizadas son:

- PHP: Creado en 1994, se trata de un lenguaje de programación de uso general de script que originalmente fue diseñado para proporcionar contenido web dinámico. Su código está empotrado en el código HTML y es interpretado por un servidor web para generar el documento HTML finalal vuelo, evitando la necesidad de acceder a un archivo externo.
- Django: Se trata de un entorno programado en Python que proporciona un conjunto de componentes en el lado del servidor para ayudar a la hora de desarrollar una aplicación web. Sigue el diseño de Modelo-Vista-Controlador, que separa la lógica y datos de la interfaz gráfica y de las comunicaciones y eventos. Cuando un cliente solicita una URL al servidor, esta pasa por Django que analizará la URL solicitada y pasará la petición a la función correspondiente llamada *vista*. En esta función se ejecutará lo necesario para proporcionar al cliente lo solicitado con su URL.
- Rails: Al igual que Django, Ruby on Rails se trata de un entorno para facilitar el desarrollador web programado en Ruby que sigue el diseño de Modelo-Vista-Controlador. El funcionamiento de Django y Rails es muy similar, siendo la principal diferencia el lenguaje en el que están programados, siendo necesario menos código en el caso de Rails.

- Node.js: Es un entorno de ejecución multiplataforma de código abierto para el lado del servidor basándose en el lenguaje JavaScript. Se basa en eventos y gestiona todas las operaciones con una programación asíncrona. Todo esto facilita el desarrollo de aplicaciones web escalables de manera sencilla y robusta.

1.3. Tecnologías web en robótica

La utilización de tecnologías web en robótica es aún un campo con poco desarrollo. Sin embargo, debido a las ventajas que ofrece frente a otras tecnologías (el mismo código funciona en cualquier plataforma, no es necesario realizar ninguna instalación, etc.) es un campo prometedor de cara al futuro.

Uno de los desarrollos más importante de tecnologías web en robótica son las bibliotecas y herramientas de código abierto para su utilización con el middleware ROS, elaboradas por la comunidad de Robot Web Tools[18]. Por ejemplo:

- `Rosbridge Suite`: Proporciona una interfaz usando JSON para ROS que permite a cualquier cliente enviar JSON para conectar con el robot, mediante las capas de comunicación WebSockets, UDP y TCP. Realmente podría entenderse como un servidor intermedio que recibe o envía la información a un cliente web.

- `roslibjs`: Se trata de la biblioteca que da soporte a la interactuación entre una aplicación web desarrollada en JavaScript con ROS.

- `ros2djs` y `ros3djs`: Son las bibliotecas elaboradas para gestionar la visualización de elementos en dos y tres dimensiones, respectivamente. Están elaboradas utilizando `roslibjs` y proporcionan funciones que son estándares en ROS como la elaboración de mapas, el procesado de nubes de puntos o del escaneo laser.

Adicionalmente Robot Web Tools, ofrece una serie de herramientas como un servidor de video web, una herramienta para mostrar e interactuar con la navegación autónoma del robot, una herramienta para la creación de teleoperadores de robots, etc.

Otro ejemplo de desarrollos robóticos con tecnologías web son los TFGs realizados en el Grupo de Robótica de la URJC. Una parte de este proyecto está basado en el TFG

[18] `http://robotwebtools.org/`

elaborado por Aitor Martínez[19]. En este trabajo se crearon tres aplicaciones capaces de conectarse con varios sistemas robóticos: `CameraViewjs`, `KobukiViewerjs` y `UavViewerjs`.

`CameraViewjs` es un visualizador de imágenes, desarrollado utilizando JavaScript, HTML5 y CSS3 en el lado del Cliente y NodeJS en el lado del servidor, e ICE como middleware. Permite la visualización de las imágenes recibidas desde un servidor de imágenes, ya sean obtenidas desde una webcam, una cámara conectada a un robot o almacenadas en el dispositivo. En la Figura 1.6 se puede ver el uso de esta aplicación

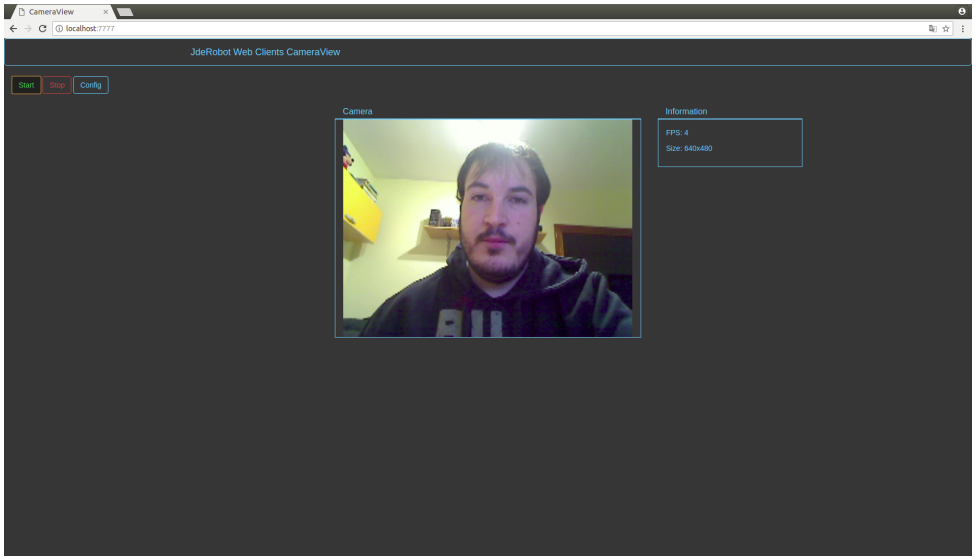


Figura 1.6: CameraViewjs

`KobukiViewerjs` es un visualizador y teleoperador de robots con ruedas del tipo Turtlebot, desarrollado utilizando JavaScript, HTML5 y CSS3 en el lado del Cliente y NodeJS en el lado del servidor, e ICE como middleware. Consta de varias partes, la primera de ellas es mostrar las imágenes obtenidas a través de las dos cámaras de las que dispone el robot (izquierda y derecha), otra parte donde muestra el escaneo láser, una representación tridimensional del robot y el movimiento del mismo, y por último, el teleoperador, que envía al robot una velocidad lineal y otra angular para indicar tanto el movimiento en línea recta como la orientación del mismo. En la Figura 1.7 se puede ver un ejemplo de uso de esta aplicación

[19] `https://jderobot.org/Aitormf-tfg`

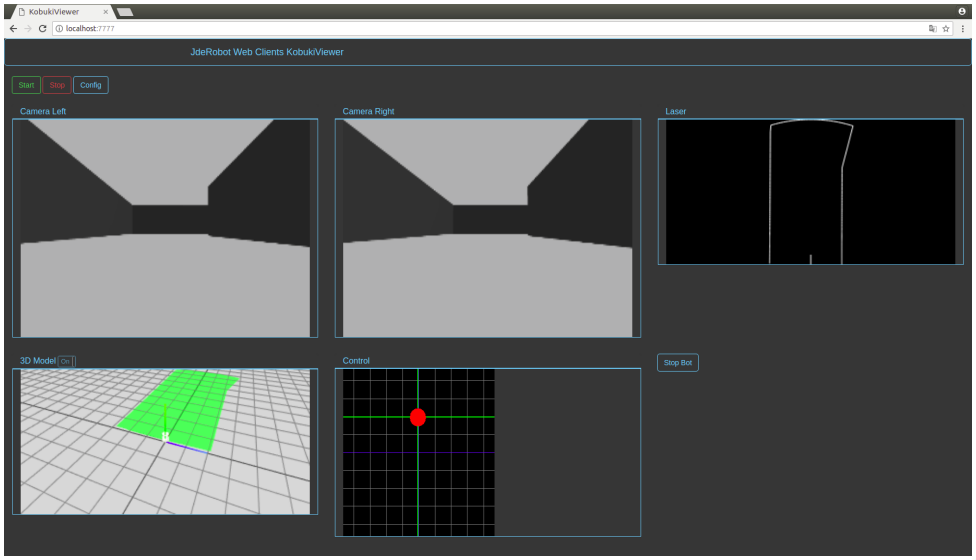


Figura 1.7: KobukiViewerjs

`UavViewerjs` es un visualizador y teleoperador de drones, desarrollado utilizando JavaScript, HTML5 y CSS3 en el lado del Cliente y NodeJS en el lado del servidor, e ICE como middleware. La aplicación tiene integrada sobre la misma pantalla el teleoperador y la imagen obtenida de la cámara, pudiéndose elegir si deseamos mostrar la cámara frontal o la inferior del dron. Consta también de una representación tridimensional del dron y de su movimiento. Desde la aplicación se teleopera mediante el envío de una velocidad lineal y otra angular, además se le indica cuándo se desea aterrizar y despegar. En la Figura 1.8 se muestra una prueba de esta aplicación.

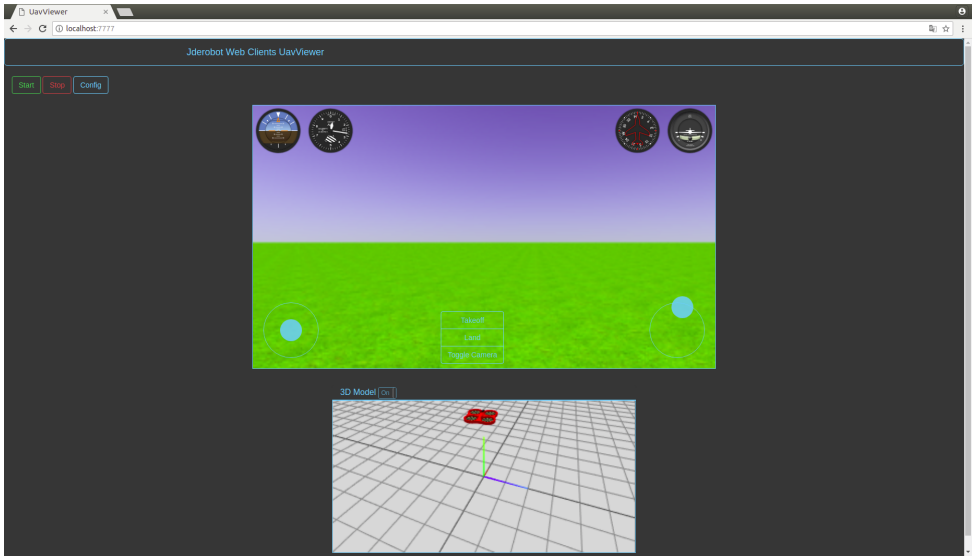


Figura 1.8: UavViewerjs

Capítulo 2

Objetivos

En este capitulo se presentan los propositos planteados para este proyecto y la metodología que se ha utilizado para alcanzarlos.

2.1. Objetivos

La principal meta de este trabajo es la de enriquecer con tecnologías web las herramientas robóticas existentes en la plataforma JdeRobot. Para alcanzar este objetivo, se ha dividido el trabajo en tres partes:

1. Modificar los tres visores existentes elaboradas mediante tecnologías web en la plataforma JdeRobot (`CameraViewjs`, `KobukiViewerjs` y `UavViewerjs`) para su utilización con el middleware ROS, además de ICE como hasta ahora.

2. Creación mediante tecnologías web de un nuevo visor que permitirá la visualización de elementos 3D (puntos, líneas y objetos 3D) usando WebGL.

3. Elaborar un nuevo driver robótico de imágenes utilizando tecnologías web y WebRTC, cuyo cometido será el de servir imágenes a los distintos visores existentes en JdeRobot.

Además de desarrollar estas piezas software para su ejecución dentro del navegador, se utilizará el entorno web Electron en todos ellos para posibilitar que se puedan ejecutar como una aplicación de escritorio sobre diferentes sistemas operativos.

2.2. Metodología

La metodología elegida para la ejecución del trabajo ha sido el desarrollo en espiral. Se trata de uno de los modelos más utilizados en el desarrollo de software y consiste en la realización de una serie de ciclos o iteraciones que se repiten en forma de espiral.

Figura 2.1: Modelo de desarrollo en espiral

Cada iteración o ciclo está formado por cuatro etapas.

1. Determinación de los objetivos a alcanzar para que el ciclo sea finalizado de manera exitosa.

2. Analizar los riegos que conlleva las elecciones tomadas para realizar el desarrollo, y establecer alternativas para solventar los posibles inconvenientes.

3. Desarrollar y probar los objetivos establecidos en la primera fase.

4. Planificar las siguientes etapas del proyecto, teniendo en cuenta los resultados obtenidos en esta interacción.

De cara a llevar un mejor del trabajo, se han tenido reuniones semanales con el tutor, en las que se marcaban los objetivos para la siguiente semana, se exponían dudas o posibles problemas así como se establecen posibles alternativas, y se revisaba el trabajo previo. Con estas reuniones semanales se consigue tener flujo de trabajo constante y fluido, disminuyendo la posibilidad de quedar bloqueado en algún punto durante un largo periodo de tiempo.

Además, se ha elaborado un bitácora en la mediawiki[1] de JdeRobot, donde quede reflejado todos los progresos así como videos demostrativos de los avances. También se dispone de un repositorio en GitHub[2] donde se ha ido subiendo todo el código para su verificación y prueba por terceras personas.

2.3. Plan de Trabajo

El plan de trabajo seguido es el siguiente:

1. Familiarización con las tecnologías robóticas que se van a utilizar en este trabajo: La plataforma JdeRobot, el simulador Gazebo y los middleware ICE y ROS

2. Entender el funcionamiento de las tecnologías web que se utilizarán en el desarrollo de las herramientas: El entorno Electron, WebGL, WebRTC y RobotWebTools para la compatibilidad de las tecnologías web con ROS.

3. Ampliación de las herramientas web existentes en la plataforma JdeRobot: Se adaptarán para ser usadas con Electron, y posteriormente se añadirá la compatibilidad con el middleware ROS, de modo que puedan ser usadas con ambos middleware de comunicación.

4. Creación de un nuevo visor de objetos 3D: Primero se revisará el visor utilizado hasta ahora y desarrollado con C++. Una vez conocido su funcionamiento, se creará el nuevo visor 3D adaptando las conexiones ICE del visor antiguo para que funcionen con el nuevo. Finalmente, se ampliará el visor para que pueda mostrar más primitivas además de los puntos que podía mostrar el antiguo, concretamente segmentos y modelos 3D de objetos cualesquiera.

[1] `https://jderobot.org/Rperez-tfg`

[2] `https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez`

5. Para terminar el trabajo, se implementará un nuevo driver que proporcionará un servidor de imágenes utilizando tecnologías web y WebRTC.

Capítulo 3

Infraestructura

En este capítulo se describen las tecnologías sobre las que se cimenta este trabajo. Se empezará haciendo una introducción al mundo de los robots, ya sean reales o simulados, a los que las herramientas que se van a desarrollar en este trabajo podrán conectarse. Posteriormente se indican las diferentes infraestructuras de robótica y comunicación utilizadas en el desarrollo de este trabajo, destacando la plataforma JdeRobot y los *middleware* de comunicación ICE y ROS. Finalmente, se explican diferentes infraestructuras de tecnologías web, haciendo especial hincapié en el entorno Electron y Node.js, ya que se dotará a todas las herramientas desarrolladas la posibilidad de ser ejecutadas por estas dos vías.

3.1. Robots reales y simulados

Los robots pueden ser reales o simulados. Mientras que los robots reales son más costosos y más difíciles de manejar, los robots simulados permiten que cualquier persona desde un ordenador pueda probar sus desarrollos sin riesgo y que posteriormente sean capaces de funcionar con un robot real.

Las herramientas desarrolladas en este trabajo son capaces de conectarse a los robots, tanto simulados como reales.

3.1.1. Robots simulados: Gazebo

Se trata del principal programa de simulación de robots existente[1], de código abierto y bajo la licencia Apache 2.0, que lleva largo tiempo utilizándose en laboratorios de investigación de robótica e inteligencia artificial. Con Gazebo es posible simular desde una cámara a robots más complejos de una forma sencilla gracias a su interfaz simple e intuitiva. Cada robot cuenta con drivers simulados como pueden ser la cámara, los sensores o los actuadores, de manera que la experiencia sea lo más parecida a un robot real.

En este trabajo se va a hacer un uso destacable de este simulador en su version 7.14, conectando cada uno de los visores a un robot simulado. En concreto, las herramientas son capaces de conectarse a un Turtlebot (Figura 3.1) y un drone simulados obteniendo la información o enviando órdenes a los diferentes drivers del robot: la cámara, el láser y los motores.

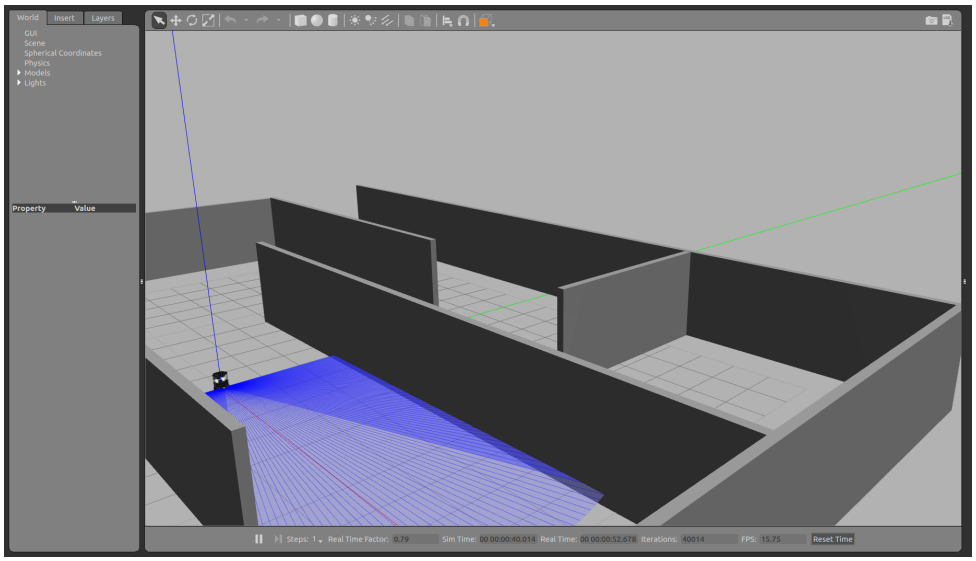


Figura 3.1: Robot Turtlebot simulado con Gazebo

3.1.2. Robots reales

Gracias a que las herramientas elaboradas se han probado con los robots simulados durante el desarrollo, es posible probarlas con robots reales sabiendo que las herramientas

[1] `http://gazebosim.org/`

van a funcionar correctamente. Tanto el robot Turtlebot real (Figura 3.2) como el drone real, cuentan con los mismos drivers que sus versiones simuladas, por lo que la información obtenida o las órdenes transmitidas será la misma y las herramientas podrán conectarse sin ningún tipo de problema.

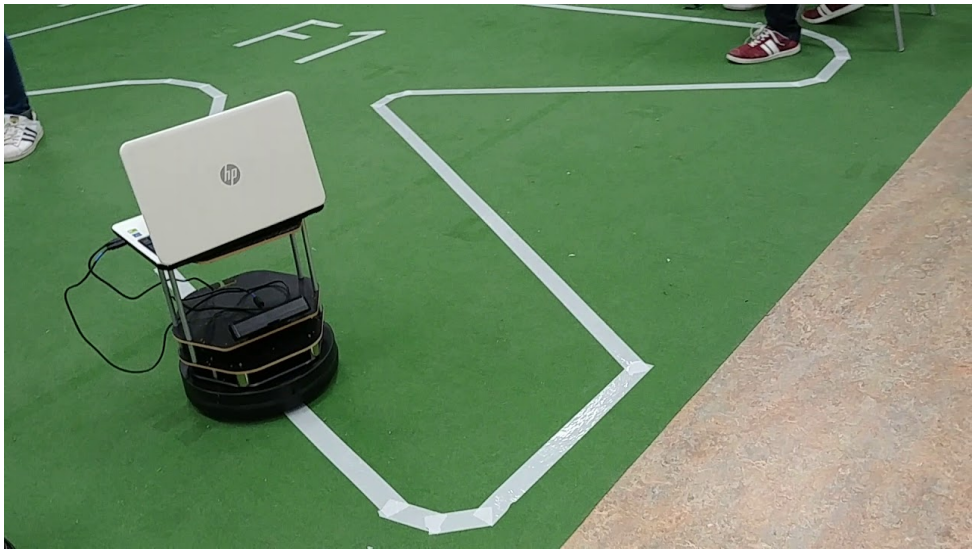


Figura 3.2: Robot Turtlebot real

3.2. La plataforma JdeRobot

JdeRobot[2] es un entorno de software libre para robótica y visión artificial creado por el grupo de robótica de la Universidad Rey Juan Carlos y licenciado bajo GPL v3[3]. Su desarrollo principalmente está realizado mediante C, C++ y Python, incorporando últimamente desarrollos en JavaScript, como en este trabajo.

JdeRobot está basado en componentes que son interconectados mediante el uso de middlewares como ICE o ROS, facilitando el acceso a los dispositivos hardware. Estos componentes obtienen mediciones de los sensores u órdenes del motor a través de llamadas a funciones locales. JdeRobot conecta esas llamadas a drivers conectados a sensores (para la recepción de las mediciones) o actuadores (para las órdenes), ya sean reales o simulados. Estas funciones locales forman la API de la capa de abstracción del hardware (HAL-API). La plataforma también ofrece una serie de herramientas para facilitar la teleoperación o

[2] `https://jderobot.org/Main_Page`

[3] `http://www.gnu.org/licenses/gpl-3.0-standalone.html`

el procesamiento de las mediciones de los sensores, y bibliotecas.

Para el desarrollo de este proyecto, se ha usado la versión de JdeRobot 5.6.4

3.3. El Middleware ICE

Se trata de una biblioteca orientada a objetos que ayuda a crear aplicaciones distribuidas fácilmente[4]. ICE se ocupa de todas las interacciones con las interfaces de programación de bajo nivel de red (ahorra al desarrollador la tarea de apertura de puertos, conexiones de red o serialización de datos). El objetivo principal de ICE es facilitar el desarrollo de aplicaciones, de modo que en muy poco tiempo se pueda aprender a utilizarlo. Toda conexión ICE cuenta con un cliente y un servidor, de modo que el cliente siempre lleva la iniciativa de la conexión, enviando peticiones al servidor, el cual responde a esas peticiones. En la Figura 3.3 puede verse esta estructura de cliente-servidor.

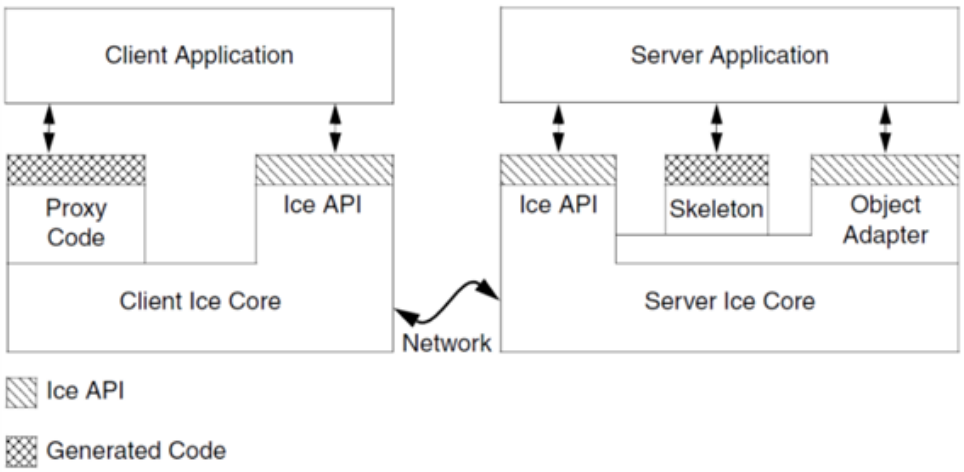


Figura 3.3: Estructura de Cliente - Servidor con Ice

ICE tiene un lenguaje de especificación propio llamado Slice (Specification Language for ICE) que permite la abstracción fundamental para separar interfaces de objetos de sus implementaciones. Este lenguaje especifica las interfaces, operaciones y tipos de parámetros utilizados por la aplicación. Cada una de las aplicaciones que se deseen que se comuniquen entre sí deben compartir la misma descripción Slice. Esta descripción es

[4] `https://zeroc.com/`

independiente del lenguaje en el que está desarrollado nuestro cliente o servidor, de modo que es posible su utilización con clientes y servidores escritos en diferentes lenguajes de programación.

Las interfaces `Slice` pueden verse como un contrato firmado entre un cliente y un servidor para compartir los mismos tipos, funciones y elementos, dando igual el lenguaje de programación en el que estén escritos, ya que posteriormente se traducen usando el compilador correspondiente al lenguaje de programación respectivo. Si el cliente y el servidor no compartiesen la misma interface `Slice`, la conexión no podría llevarse a cabo al no conocer las funciones o tipos que maneja cada uno.

Para este trabajo se va a utilizar su versión para JavaScript. El soporte para este lenguaje es relativamente reciente, por lo que muchas de las funcionalidades que ofrece para otros lenguajes de programación como C++ o Python, no están aún disponibles para JavaScript. Por ejemplo no hay soporte para la creación de un servidor completo mediante JavaScript. Por ello durante este trabajo se ha optado por utilizar ICE únicamente como cliente, con las ventajas e inconvenientes a los que se hará referencia en próximos capítulos.

La versión de ICE que se usa en este trabajo para conectar las herramientas web con los drivers robóticos es la 3.6.4.

3.4. El Middleware ROS

Se trata de un ecosistema para el desarrollo de software robótico que ofrece las funcionalidades de un sistema operativo (abstracción del hardware, control de dispositivos de bajo nivel, mantenimiento de paquetes, etc.)[5]. ROS ofrece una serie de herramientas, bibliotecas y convenciones para simplificar la tarea de crear complejas y robustas aplicaciones para robots. Nace con la idea de fomentar el desarrollo colaborativo, es decir que cualquier persona que realice un desarrollo puede subir a los repositorios que ofrece ROS para ello, de modo que otros puedan reutilizarlo en sus proyectos con relativa facilidad.

El elemento fundamental del funcionamiento de ROS es el nodo. Un nodo es un proceso que realiza cálculos y se comunica con otros mediante: (a) un sistema de publicación - subscripción para conexiones asíncronas y anónimas, o (b) servicios para las conexiones

[5] `http://www.ros.org/`

síncronas.

Típicamente cada nodo se ocupa de una parte del sistema de control de robot, por ejemplo un nodo se ocupa de los motores, otro de realizar la localización, etc. Este elemento proporciona mayor tolerancia a los fallos al ser bloques separados y aislados, y se reduce la complejidad del código.

En el sistema de comunicación asíncrona (Figura 3.4), un nodo actúa como publicador que transmite un mensaje con una etiqueta llamada *topic* por el canal para que sea recibido por cualquier otro nodo que se subscriba a esta etiqueta o *topic*. El mensaje enviado es una estructura de datos simple, que comprende campos tipados. ROS ofrece una serie de formatos de mensajes estándar que cubren la mayoría de necesidades de uso común (mensajes para sensores, cámaras, movimiento, láseres, nubes de puntos, etc).

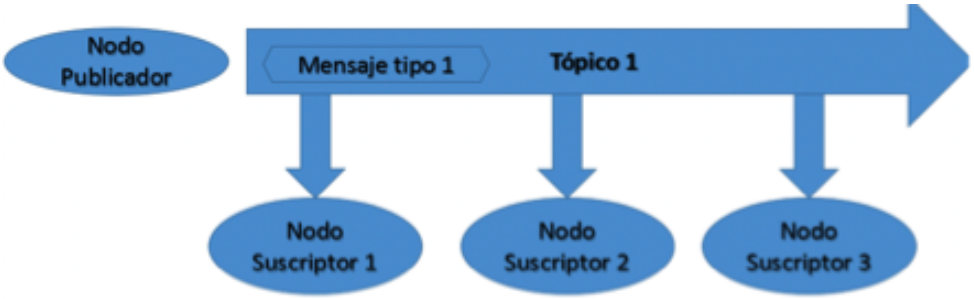


Figura 3.4: Estructura del sistema de comunicación Publicador - Subscriptor

El sistema de comunicación síncrona (Figura 3.5) es el típico sistema de llamada a procedimiento remoto, a través del cual un nodo ROS es el prestador del servicio que permanece en escucha continua y el resto de nodos le envían mensajes de solicitud. Cada servicio está definido por la cantidad y tipo de datos que necesita, tanto para recibir la petición, como para enviar la respuesta.

Figura 3.5: Estructura del sistema de comunicación por Servicios ROS

Un elemento muy útil para este proyecto es que ROS proporciona total compatibilidad con Gazebo mediante un conjunto de paquetes llamados *gazebo_ros_pkgs* [6]. En ROS, los paquetes son aquellos donde se incluye todo el código fuente, las librerías usadas y cualquier otro recurso necesario para que funcione el nodo.

La versión de ROS que se utilizará para el desarrollo de este trabajo es ROS Kinetic.

3.5. Robot Web Tools

Robot Web Tools[7] es una comunidad nacida a partir de la de ROS, cuyo software permite conectar aplicaciones web a elementos robóticos gracias a un intermediario llamado `Rosbridge`. Este intermediario es a la vez una especificación de JSON para interactuar con ROS y una capa de transporte para que los clientes se comuniquen mediante WebSockets.

Por otro lado, han creado una serie de bibliotecas livianas y fáciles de utilizar de JavaScript que proporcionan una abstracción de la funcionalidad principal de ROS. Estas bibliotecas son `roslibjs`, `ros2djs` y `ros3djs`. En este trabajo solo se utiliza `roslibjs`.

La biblioteca `roslibjs` se encarga de ofrecer las funcionalidades necesarias para conectar, enviar o recibir mensajes ya sea mediante publicación y subscripción o mediante servicios. La conexión se realiza a un servidor intermedio (`Rosbridge Server`), el cual se encarga de gestionar las conexiones, los `topic` publicados o los servicios activos en

[6] `http://wiki.ros.org/gazebo_ros_pkgs`
[7] `http://robotwebtools.org/`

cada momento, de modo que cuando un nodo se subscriba o realice la petición de un servicio, este servidor contiene la capa de transporte para encaminar la petición mediante WebSockets. La funcionalidad de este servidor intermedio y las interacciones se pueden apreciar en la figura 3.6.

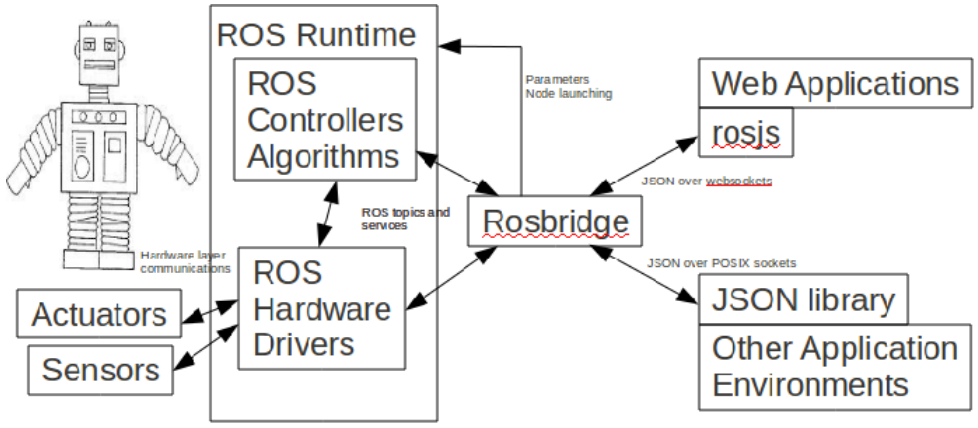


Figura 3.6: Estructura de una aplicación con Rosbridge

Dado que para este trabajo se va a utilizar JavaScript como lenguaje, se utilizará *Rosbrige* en todas las partes relacionadas con ROS, las bibliotecas, servidores y protocolo indicado en esta sección.

3.6. Node.js

Node.js[8] es un entorno multiplataforma del lado del servidor. Concebido con la intención de facilitar la creación de programas de red escalables, como puede ser un servidor web, permite ejecutar código JavaScript fuera de un navegador y aprovechar las ventajas que proporciona su programación orientada a eventos. Su ejecución se lleva a cabo en un único hilo, usando entradas y salidas asíncronas que pueden ejecutarse de manera concurrente, provocando que cada una de ellas necesite de un *callback* para manejar los eventos.

Node.js proporciona una serie de módulos básicos que permiten realizar funciones esenciales como puede ser la programación en red asíncrona, el manejo de archivos del sistema, etc. Al ser de código abierto, existe una gran comunidad de desarrolladores que

[8] `https://nodejs.org/es/`

crean nuevos módulos para que cualquiera pueda utilizarlos. Estos módulos son fácilmente compartidos gracias al manejador de paquetes npm[9], permitiéndo compilar, instalar y manejar las dependencias de cualquier módulo de terceros que deseemos usar en nuestro proyecto.

Por lo general, un proyecto Node.js contendrá al menos dos elementos, un archivo .js que contendrá la lógica del programa escrita en JavaScript y un segundo archivo llamado Package.json que definirá nuestro programa. Este segundo archivo es donde se indican las dependencias que usará nuestro programa y facilitará su instalación conjunta usando el comando npm install, así como se referenciará al fichero .js indicado anteriormente.

La versión de Node.js utilizada es la 8.9.1 y la versión del manejador de paquetes npm es la 6.4.0.

3.7. WebGL y Three.js

WebGL [10] es un API multiplataforma utilizado para crear gráficos 3D utilizando tecnologías web, está basado en OpenGL y utiliza parte de su API. WebGL se ejecuta dentro del elemento HTML Canvas, lo que proporciona una completa integración con la interfaz DOM. Ofrece ventajas como la compatibilidad con distintos navegadores y plataformas, no es necesario compilar para su ejecución o la interacción con otros elementos del HTML. Sin embargo, debido a que se trata de un API de bajo nivel es complejo de utilizar.

Three.js [11] nace como remedio a la complejidad de usar WebGL. Se trata de una biblioteca desarrollada en JavaScript que permite crear y mostrar gráficos 3D en un navegador web usando un API de alto nivel, proporciona funciones y objetos para facilitar la creación, interacción y visualización de entornos con gráficos 3D. Utilizando secuencias de código tan simples como `object = new THREE.Mesh( new THREE.SphereBufferGeometry(75, 20, 10), new THREE.MeshBasicMaterial(color:0xFF0000))`, permite crear una esfera (Figura 3.7) siendo la primera parte donde se define la geometría y la segunda donde se establece el material (puede ser desde un color básico como en este caso, hasta una

[9] `https://www.npmjs.com/`
[10] `https://get.webgl.org/`
[11] `https://threejs.org/`

textura obtenida desde una imagen)

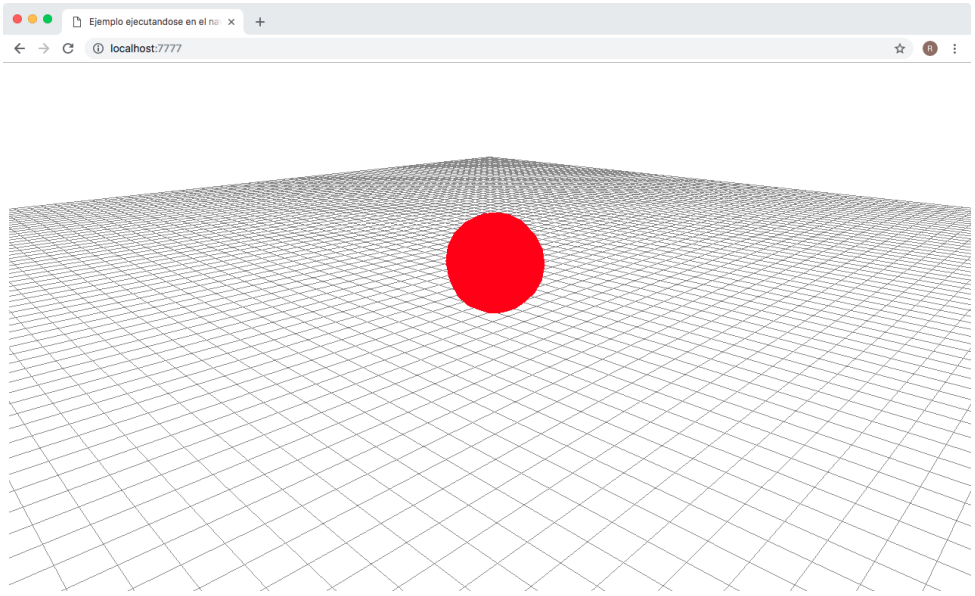


Figura 3.7: Esfera creada con la biblioteca Three.js y mostrada en un navegador

3.8. WebRTC

WebRTC [12] una tecnología que permite a una aplicación web capturar y transmitir audio y video, así como intercambiar datos audiovisuales con otros navegadores sin necesidad de intermediarios. Este intercambio se realiza de igual a igual (*peer-to-peer*) sin necesidad de instalaciones o software adicionales. WebRTC proporciona varios API y protocolos interrelacionados para dar a los navegadores y aplicaciones móviles la capacidad de intercambiar elementos multimedia en tiempo real (*Real Time Communications*). Esta tecnología esta soportada en los principales navegadores y tiene el soporte de Google.

En este proyecto se usa WebRTC para la adquisición del video obtenido de una cámara web. Para lograr este objetivo, se utiliza el API de WebRTC Media Stream, que proporciona la descripción de los flujos de datos de audio y video, los métodos para trabajar con ellos, la conexión con los dispositivos para adquirirlos, las limitaciones asociadas a cada tipo de datos o los eventos asociados al proceso.

[12] `https://webrtc.org/`

3.9. El entorno Electron

Electron[13] es un entorno de código abierto desarrollado por GitHub. Comenzó su desarrollo en 2013, en el mismo grupo de trabajo del editor Atom[14]. Fue concebido con la idea de permitir la creación de aplicaciones de escritorio multiplataforma con tecnologías web. Electron es la combinación de NodeJS y Chromium[15] en una misma ejecución.

La arquitectura de una aplicación que utiliza Electron está formada por dos procesos: Principal y Renderizador.

El proceso principal es el encargado de generar la interfaz de usuario mediante la creación de páginas web y las administra de modo que es posible mostrar más de una página web al mismo tiempo. Esta labor se realiza mediante la instancia al objeto `BrowserWindow` de Electron, ejecutándose una página web cada vez que se instancia. Cuando se destruye una de estas instancias, se está cerrando esa página web. Cada aplicación con Electron debe constar de un único proceso principal, y corresponderá al `script main` del archivo `package.json`.

El proceso renderizador es cada instancia del objeto `BrowserWindow` y la ejecución de la página web correspondiente. Una aplicación con Electron puede tener multitud de procesos renderizadores, siendo cada uno independiente del resto. Cada proceso solo se preocupa de la página web que se está ejecutando en él.

Electron es totalmente compatible con NodeJS, tanto en el proceso principal como en el renderizador, por lo que todas las herramientas disponibles para Node.js, también lo están para Electron. Así mismo, es posible utilizar módulos Node.js alojados en el repositorio de paquetes *npm* mencionado anteriormente. Éste nos aporta un gran número de ventajas como una mayor seguridad al cargar contenido remoto, tener siempre actualizadas las aplicaciones o tener un gran número de bibliotecas disponibles.

Todas las aplicaciones que se desarrollarán en este trabajo podrán ser ejecutadas utilizando Electron, lo que permite utilizarlas en cualquier plataforma o, incluso, empaquetarlas usando *npm* o mediante un archivo Asar[16].

[13] `https://electronjs.org/docs`
[14] `https://atom.io/`
[15] `https://www.chromium.org/Home`
[16] `https://github.com/electron/asar`

3.9.1. Adaptación de una aplicación web a Electron

Típicamente una aplicación que utiliza Electron está formada por:

1. Un archivo HTML.
2. Un fichero `main.js`, que define la ventana donde se mostrará el fichero HTML y se creará.
3. El archivo `package.json`, que define la aplicación en Electron (nombre, versión, descripción, etc.), se indican en él las dependencias a módulos externos y el fichero main.js para que crear la aplicación.

Una vez que se cuenta con estos tres elementos, se puede instalar las dependencias mediante `npm install` (igual que para Node.js) y ejecutar la aplicación mediante `npm start` o `npm test` (dependerá de como se haya definido `package.json`).

3.9.1.1. package.json

El cometido de este archivo es definir las dependencias de paquetes de terceros que tiene la aplicación web, las características de la aplicación (nombre, versión, autor, etc.) y, lo que es más importante, de qué manera se lanzará la aplicación y el archivo que se debe ejecutar al lanzarla. El cuadro 3.1 muestra un ejemplo de un archivo `package.json`

```
{"name": "Ejemplo",
  "version": "0.1.0",
  "main": "main.js",
  "scripts": {
    "start": "electron ."
  },
  "dependencies": {
     "electron": "^1.8.4",
  }
}
```

Como se puede apreciar, la versión de Electron que se utilizará será la 1.8.4.

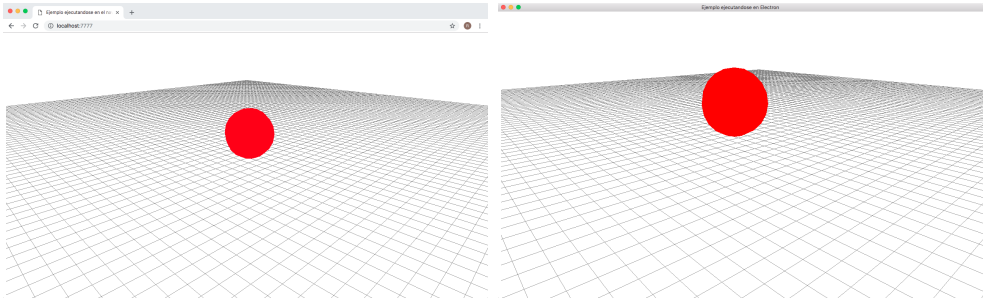
3.9.1.2. main.js

El archivo `main.js`, contiene muy pocas líneas de código al tratarse de un elemento totalmente externo a la aplicación y no tener ningún papel en su funcionamiento. Únicamente es necesario para ejecutar la aplicación utilizando Electron. El cuadro 3.2 muestra un ejemplo del contenido de este archivo.

```
const app = require('electron')
const path = require('path')
const url = require('url')

let win;

function createWindow () {
  win = new BrowserWindow({width: 1800, height: 1000})
  win.loadURL(url.format({
    pathname: path.join(__dirname, 'camserver.html'),
    protocol: 'file:',
  }))

app.on('ready', createWindow)
```

Lo primero que se hace es definir los módulos que se requieren para que funcione correctamente y que se utilizarán en el resto del código. Posteriormente, se define el tamaño de la ventana, en este caso la ventana será de 1800 pixeles de ancho y 1000 de alto. Mediante `win.loadURL` se simula el funcionamiento de un navegador y como gestionan las diferentes URL. A este método se le pasarán como parámetros el archivo HTML principal de la aplicación web y el protocolo que este caso es `file:`, al estar queriendo mostrar directamente de un archivo HTML (si quisiéramos mostrar mediante Electron el contenido de una página web ya existente, el protocolo tomaría el valor `http:`). Finalmente, la última línea de código llamará a la función que crea la ventana y muestra el HTML en el momento que Electron haya terminado de inicializarse y está preparado para la creación de la ventana.

En la figura 3.8 se muestra el mismo HTML ejecutado con Node.js y con Electron. Como se puede apreciar, el resultado es el mismo.



(a) Node.js (b) Electron

Figura 3.8: Ejemplo de la esfera anterior ejecutado con Node.js y con Electron

Capítulo 4

Visores web modificados

En este capítulo se va a explicar cómo se han modificado los visores con tecnologías web existentes en la plataforma JdeRobot y desarrollados por Aitor Martínez en su TFG[1], para que utilicen el *middleware* de comunicación ROS, además de ICE. Por otro lado, se les añade la posibilidad de ser ejecutados mediante el entorno Electron, además de en un navegador web. Estos tres visores son `CamVizWeb`, `TurtleBotVizWeb` y `DroneVizWeb`.

4.1. CamVizWeb

Se trata de un visor de imágenes desarrollado con tecnologías web, que usa los middleware de comunicación ICE y ROS y puede ser ejecutado con Electron o en un navegador web[2].

4.1.1. Diseño

El visor previamente solo contaba con el *middleware* ICE para realizar la comunicación, por lo que su funcionalidad estaba limitada a los drivers que son capaces de comunicarse de esta forma. Sin embargo, con esta extensión, se amplía el abanico de drivers que pueden conectarse al permitir la utilización tanto de ICE como de ROS. La Figura 4.1 muestra el diagrama de bloques tras desarrollar la comunicación mediante las dos vías.

[1] `https://jderobot.org/Aitormf-tfg`

[2] `https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez/tree/master/camVizWeb`

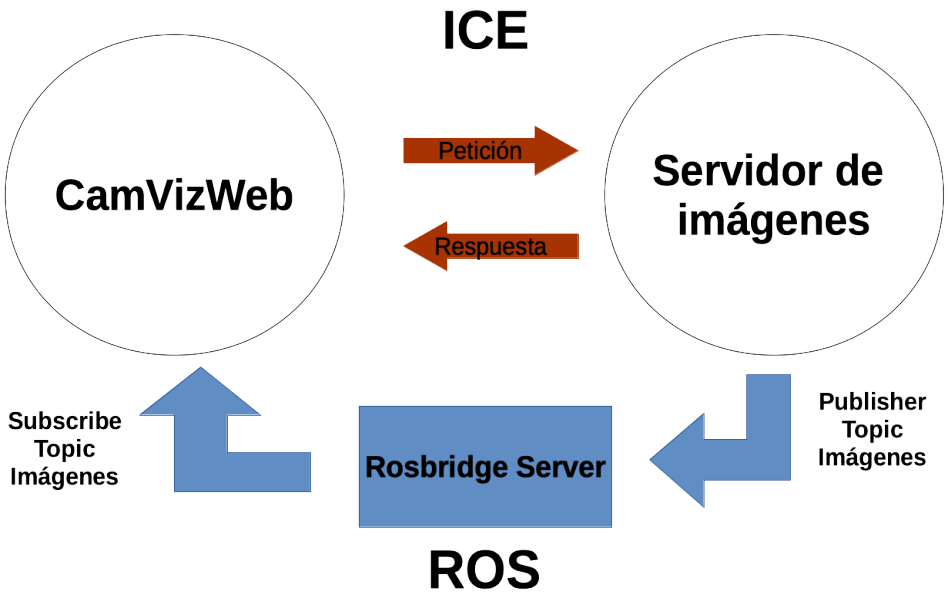


Figura 4.1: Diagrama de bloques actual del sistema

La Figura 4.2 muestra la estructura del visor tras realizar la modificación.

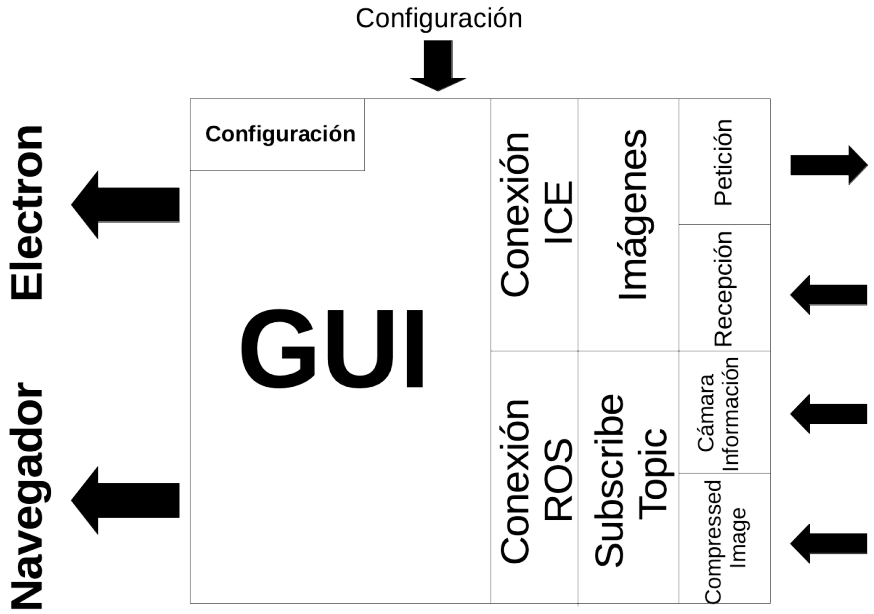


Figura 4.2: Estructura actual del visor

Internamente el visor está dividido en dos partes: la interfaz gráfica y las conexiones.

La interfaz gráfica contiene los elementos necesarios para visualizar las imágenes recibidas, además de un menú para realizar la configuración en tiempo de ejecución.

Las conexiones están formadas a su vez por otras dos partes. La primera corresponde a la conexión mediante ICE para realizar las peticiones y recibir las imágenes, siendo esta comunicación la ya existente en el visor, por lo que no se va a profundizar en su funcionamiento. La segunda corresponde a la conexión vía ROS y se obtienen los mensajes mediante los *Subscribe* de ROS. Para obtener la misma información que se obtiene utilizando ICE es necesario realizar dos *Subscribe*. El primero se utiliza para obtener las imágenes mediante el tipo de mensaje de ROS `CompressedImage`, el segundo es el encargado de suministrar la información de la cámara (anchura, altura, etc) utilizando el mensaje de ROS `CameraInfo`.

Por otro lado, el visor nuevo tiene dos novedades más. La primera corresponde al modo de ejecutarlo, ya que puede utilizarse Electron o un navegador web. Esto proporciona al visor ser ejecutado como una aplicación de escritorio, haciendolo multiplataforma y sin el problema de compatibilidad entre navegadores. La otra novedad es la posibilidad de realizar la configuración previa al arranque del visor, un fichero de formato `YAML`.

4.1.2. Interfaz gráfica

Se ha mantenido la interfaz gráfica existente realizando las modificaciones necesarias para especificar la conexión mediante el middleware ROS. Esta interfaz está formada por un elemento `Canvas` de HTML5, donde se mostrarán las imágenes, un cuadro donde irá la información acerca de la cámara y tres botones. El botón de *Start* comienza la recepción y visualización de imágenes, el botón de *Stop* detiene la recepción y visualización, y el botón menú de configuración abre una ventana emergente con las configuración.

Esta modificación se ha realizado sobre el menú de configuración para que cuando se abra la ventana emergente, se pueda seleccionar ROS o ICE. Para lograr esto, se ha añadido al archivo `index.html` un formulario de HTML que contiene dos botones, uno para ICE y otro para ROS.

En la Figura 4.3 se muestra la interfaz gráfica del visor y en la Figura 4.4 se muestra el menú de configuración.

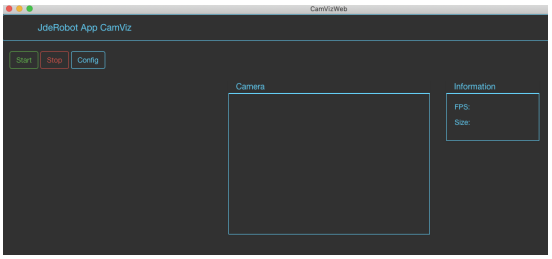


Figura 4.3: Interfaz gráfica del visor

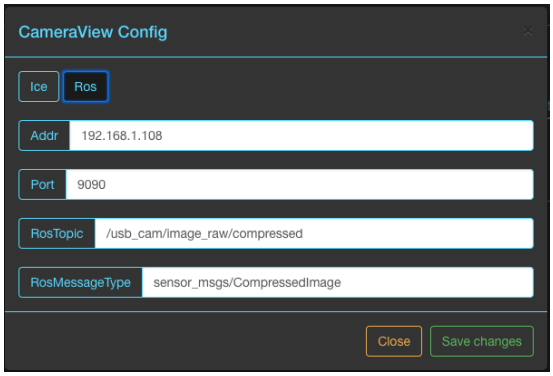


Figura 4.4: Menú de configuración del visor

4.1.3. Conexiones

En esta subsección se explica cómo se ha añadido la comunicación mediante el *middleware* ROS.

4.1.3.1. Establecimiento de la conexión

La conexión se realiza utilizando la biblioteca `roslibjs` que proporciona todo lo necesario. En el cuadro 4.1 se muestra cómo se realiza la conexión.

```
var ros = new ROSLIB.Ros({
           url : "ws://IP:Puerto"
        });

ros.on('connection', function() {
            console.log("Connect websocket")
        });
ros.on('error', function(error) {
           console.log("Error to connect websocket")
});
```

Cuadro 4.1: Establecimiento de la conexión ROS

Lo primero que se hace es definir la conexión mediante el objeto `ROSLIB.Ros` e indicando que se va a realizar mediante WebSockets y definiendo la IP y puerto de escucha el servidor intermedio Rosbridge Server. Creado este objeto, se establece la conexión con

el servidor intermedio, llamando al método `ros.on` y pasando por parámetro el *string* "Connect" y la función que se desea que se ejecute cuando se establezca la conexión. Si durante la fase de conexión ocurre algún error, se invocará al mismo método `ros.on`, pero en esta ocasión pasando por parámetro el *string* "error" y la función que se ejecutará en caso de error.

4.1.3.2. Estructura de los mensajes

Se necesitan dos mensajes ROS para obtener la misma información que se obtiene mediante ICE. Para ello, se utilizan dos mensajes de los tipos predeterminados de ROS y definidos en su documentación[3].

El mensaje con el que se recibirán las imágenes es del tipo `sensor_msgs/CompressedImage`, que son imágenes comprimidas en formato jpeg. El motivo de esta elección es por la fácil compatibilidad entre este formato y los elementos `Canvas` de HTML5, lo que permite la visualización directa de las imágenes reduciendo al mínimo el retardo. Sin embargo, este tipo de mensaje no contiene información acerca de las propiedades de la cámara, por este motivo es necesario complementar este mensaje con otro.

El mensaje para recibir las propiedades de la cámara es del tipo `sensor_msgs/CameraInfo` que da la información de la anchura y altura de cada imagen.

El cuadro 4.2 muestra cómo se definen estos mensajes. En ambos casos se utiliza el objeto de roslibjs `ROSLIB.Topic`, indicándole la conexión ROS establecida anteriormente, el nombre del *Topic*, que para el caso del mensaje con las imágenes se establecerá a través de la configuración del visor y para el caso de la información de la cámara será predeterminado, y el formato del mensaje, que son los indicados anteriormente para cada caso.

```
var roscam = new ROSLIB.Topic({
          ros : ros,
          name : conf.topic,
          messageType : "sensor_msgs/CompressedImage"
});

var rosdescrip = new ROSLIB.Topic({
```

[3] `http://wiki.ros.org/common_msgs`

```
        ros: ros,
        name : "/usb_cam/camera_info",
        messageType: "sensor_msgs/CameraInfo"
})
```

Cuadro 4.2: Definición de los mensajes ROS

4.1.3.3. Subscripción a los *Topic* y visualización de las imágenes

Una vez que se han definido los mensajes, es hora de subscribirse a los *Topic* para recibirlos. Para esto se ha definido la función `startStreaming` que se muestra en el cuadro 4.3.

```
var startStreaming = function(){
       var canvas = document.getElementById("camView");
       var fps = document.getElementById('fpsid');
       var size = document.getElementById(('sizeid');
       var ctx = canvas.getContext("2d");
       var imagedata = new Image();

       rosdescrip.subscribe(function(message){
         size.html(message.width + "x" + message.height);
       })

       roscam.subscribe(function(message){
              if (message.format != null){
                  imagedata.src = "" + message.data;
                  imagedata.onload = function(){
                  ctx.drawImage(imagedata,0,0,canvas.width,canvas.height);
              }
           } else {
                  console.log(message);
           }
           var fps_2 = calculateFPS();
           if (fps_2){
                  fps.html(Math.floor(fps_2));
```

```
            }
        });
}
```

Cuadro 4.3: Subscripción y visualización de las imágenes

Lo primero que se hace es establecer los elementos HTML donde se van a mostrar los datos recibidos. Son el `Canvas` y el cuadro donde va la información acerca de las propiedades de la cámara y los fotogramas por segundo, siendo `fps` el elemento que contendrá la información sobre los fotogramas por segundo, y `size`, el elemento que contendrá la información sobre el tamaño de cada fotograma. Posteriormente se da el contexto al elemento `Canvas` y se define la imagen que se va a mostrar como un objeto del tipo `Image`.

Una vez realizadas estas tareas previas, se realiza la subscripción a los *Topics* indicados anteriormente. Se utiliza el método de roslibjs `subscribe` pasando por parámetro la función con el código que se desea ejecutar una vez recibido los datos. Primero se subscribe a la información de la cámara y los datos recibidos, se indica que se deben mostrar en el elemento HTML donde se muestra el tamaño de cada fotograma. La segunda subscripción es para recibir las imágenes. Al mensaje recibido se le añade el encabezado para indicar que se trata de imágenes comprimidas del tipo jpeg y posteriormente mediante el método `drawImage` que proporciona el elemento `Canvas`, se muestran las imágenes recibidas en este elemento pasando por parámetros la imagen con el encabezado y el tamaño del elemento `Canvas` .

Finalmente se calculan los fotogramas por segundo que se reciben, para ello se ha creado el método `calculateFPS()`, este método resta a la marca de tiempo en la que se recibe el mensaje actual la marca de tiempo del anterior mensaje recibido. El resultado de esta función se muestra en el elemento `fps`.

4.1.4. Configuración

El visor puede ser configurado por dos vías: mediante un fichero de configuración externo para hacerlo antes de arrancar o a través del menú de configuración de la interfaz gráfica en tiempo de ejecución.

Para la primera se ha decidido usar un archivo externo con formato “YAML” que

es leído y su información cargada por el visor al arrancar. “YAML” es un formato de serialización de datos muy sencillo de usar y de leer por una aplicación. Usando una única línea de código, se puede cargar y guardar en una variable la información que contiene un archivo de este tipo.

```
const yaml = require('js-yaml');
const fs = require('fs');
config = yaml.safeLoad(fs.readFileSync('config.yml', 'utf8'))
```

Cuadro 4.4: Leer el fichero “YAML” con la configuración y guardar la información en una variable

El código del cuadro 4.4 muestra cómo realizar la lectura del fichero en la variable `config`.

Por otro lado, la configuración mediante el menú de la interfaz gráfica, permite realizarla en tiempo de ejecución. Este menú permite indicar el *middleware* de comunicación que se desea utilizar y mostrará las opciones de configuración correspondientes. Los parámetros configurables para este visor son los siguientes:

- *Middleware* utilizado para realizar la comunicación. Por defecto es ICE.
- Dirección IP. Por defecto es “localhost”
- Puerto. Por defecto es “9090”.
- Endpoint. Este parámetro solo se utiliza para realizar la comunicación mediante ICE, por lo que si se indica ROS, se ignora. Por defecto es “cameraA”
- Topic. Este parámetro solo se utiliza para realizar la comunicación mediante ROS, por lo que se indica ICE, se ignora. Por defecto es “/usb_cam/image_raw/compressed”

4.1.5. Experimentos

Como se ha indicado en la sección 4.1.1, el visor puede ser ejecutado a través de un navegador web o mediante Electron. Para realizar los experimentos, se ha utilizado el driver de ROS `usb_cam`[4].

[4] `http://wiki.ros.org/usb_cam`

En un primer terminal o consola se debe ejecutar el servidor intermedio de ROS que se puede ver en el esquema.

```
#>roslaunch rosbridge_server rosbridge_websocket.launch
```

Cuadro 4.5: Ejecución del servidor intermedio

En un segundo terminal se ejecuta el driver.

```
#>roslaunch usb_cam usb_cam.launch
```

Cuadro 4.6: Ejecución del driver de ROS label

En el tercer terminal se ejecuta CamVizWeb

- Como aplicación web utilizando Node.js[5]

```
#>node run.js
Se arranca el navegador y se introduce la URL http://localhost:7778/
```

Cuadro 4.7: Ejecución con Node.js

- Como aplicación de escritorio con Electron[6]

```
#>npm install
#>npm start
```

Cuadro 4.8: Ejecución con Electron

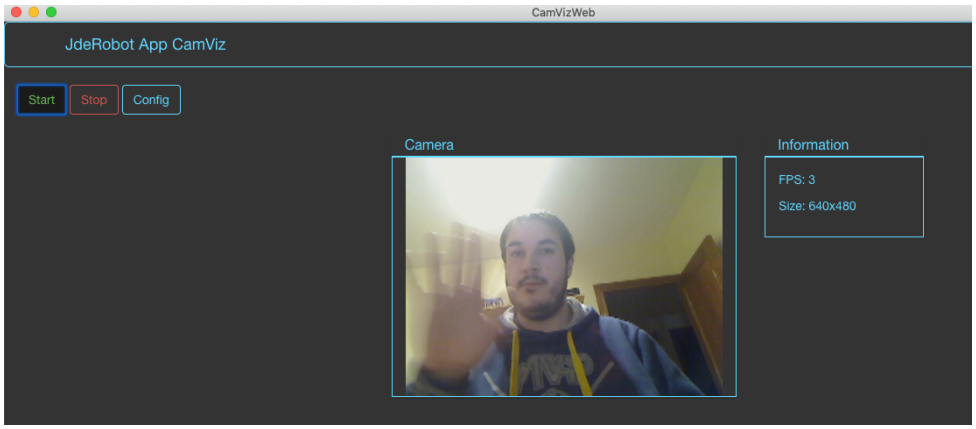


Figura 4.5: CamVizWeb ejecutado en Electron

[5] https://www.youtube.com/watch?v=cly8QOtxbMw
[6] https://www.youtube.com/watch?v=VeHj-CuIznM

4.2. TurtleBotVizWeb

Se trata de un visor y teleoperador para robots del tipo TurtleBot, desarrollado con tecnologías web, que usa los middleware de comunicación ICE y ROS y puede ser ejecutado con Electron o en un navegador web[7].

4.2.1. Diseño

La versión previa de esta herramienta solo contaba con ICE como *middleware* de comunicación, esto limitaba el uso de esta herramienta a TurtleBots que fueran capaces de comunicarse con ICE. Sin embargo, ahora será capaz también de conectarse con aquellos que se comuniquen mediante ROS. La figura 4.6 muestra el diagrama de bloques del visor tras la mejora realizada en este trabajo.

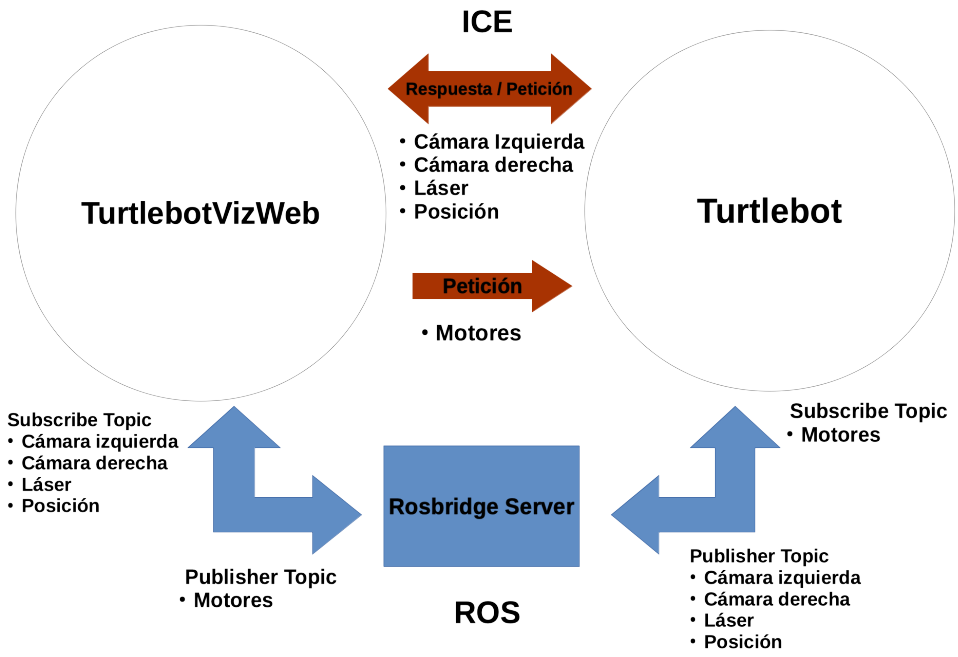


Figura 4.6: Diagrama de bloques actual del sistema

En la figura 4.7 muestra la estructura interna de la aplicación de manera detallada tras realizar la modificación.

[7] `https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez/tree/master/TurtlebotVizWeb`

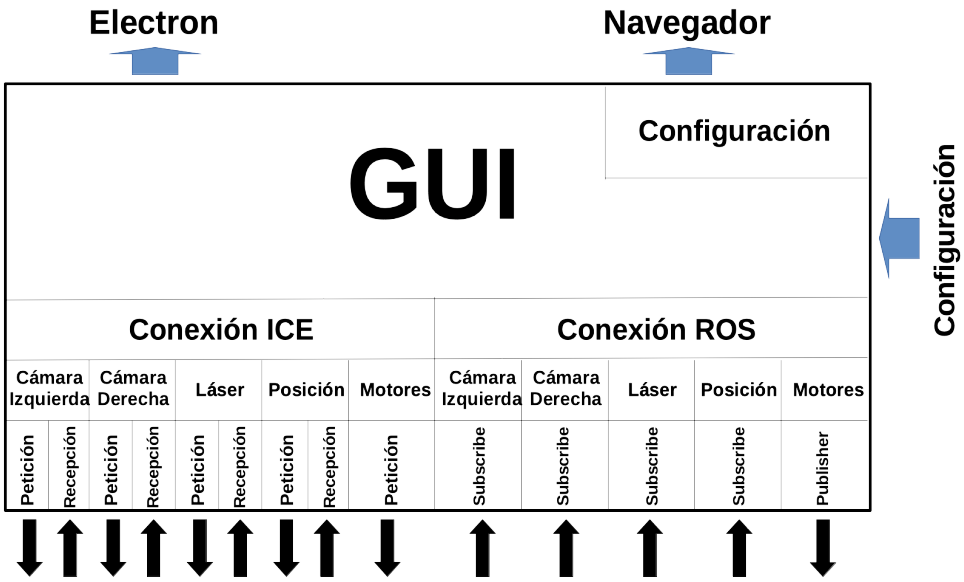


Figura 4.7: Estructura actual de la herramienta

La herramienta está dividida en dos partes: la interfaz gráfica y las conexiones. La interfaz gráfica contiene los elementos necesarios para visualizar los datos recibidos y teleoperar el TurtleBot, además de un menú para configurar en tiempo de ejecución del visor.

La conexión se realiza mediante ICE o ROS. La conexión mediante ICE es la existente previamente en la aplicación, por lo que no se profundiza en la explicación de su funcionamiento, únicamente se indica que se solicitan cuatro mensajes al TurtleBot: imagen de la cámara izquierda, imagen de la cámara derecha, datos del sensor láser y posición actual. Por otro lado, desde la propia herramienta se envía un mensaje con la información de los motores para teleoperar el robot (velocidad lineal y velocidad angular).

En el caso de ROS, se utilizan los *subscribe* para recibir los mensajes procedentes del robot y los *publisher* para transmitir la información de los motores. Teniendo en cuenta la información que se necesita, se realizan cuatro *subscribe* y un *publisher*.

Por otro lado, al igual que para el visor CamVizWeb, se le añade la posibilidad de ejecutarse mediante el entorno Electron y la configuración previa mediante un archivo externo en formato “YAML”.

4.2.2. Interfaz gráfica

La interfaz gráfica se mantiene la existente previamente realizando las modificaciones necesarias para realizar la conexión mediante el middleware ROS. La interfaz la forman cinco elementos `Canvas` de HTML5 y cuatro botones. Para poder configurar la conexión mediante ROS, se vuelve a añadir un formulario con dos botones al menú de configuración para permitir elegir en tiempo de ejecución.

Esta interfaz se puede visualizar en la figura 4.8 y en la figura 4.9 se muestra el nuevo menú de configuración.

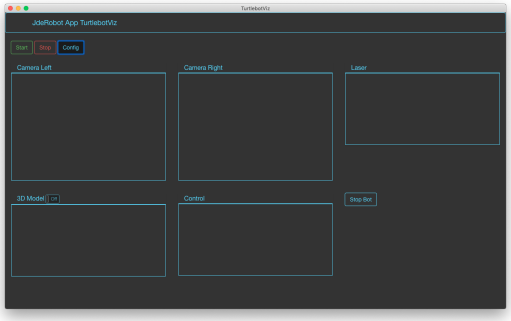


Figura 4.8: Interfaz gráfica de TurtleBotVizWeb

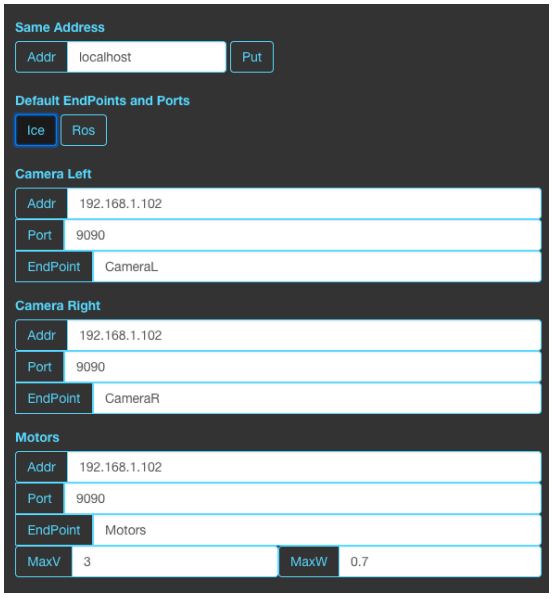


Figura 4.9: Menú de configuración de TurtleBotVizWeb

4.2.3. Conexiones

La herramienta se ha modificado para que además de conexiones ICE, sea capaz de aceptar conexiones ROS, de manera que el funcionamiento sea igual de eficiente en ambos casos.

En su versión ICE, se desarrolló un *API* (interfaz de programación de aplicaciones) con los métodos y objetos necesarios para realizar las conexiones ICE, la petición de cada mensaje (o el envío del mensaje para el caso de los motores) y el manejador de las respuestas desde el TurtleBot. Por tanto, para el caso de la comunicación con ROS, se va

a utilizar también el mismo *API* incorporando los métodos y objetos necesarios para su funcionamiento y que se explican a continuación.

4.2.3.1. Establecimiento de la conexión

Lo primero que se realiza es una conexión por cada tipo de mensaje que se desea recibir o transmitir. Cada conexión es un objeto de la *API* con su constructor y métodos correspondientes, a excepción de la información de las cámaras que se utiliza el mismo objeto tanto para la cámara izquierda como para la derecha. Por tanto, se tendrán cuatro objetos diferentes dentro de la API que son `API.CameraRos, API.LaserRos, API.MotorsRos y API.Pose3DRos`, correspondiendo a las cámaras, al sensor láser, a los motores y a la posición del robot respectivamente.

Estos objetos se crean pasándoles por parámetro la información acerca de la conexión (dirección IP, puerto y *topic* al que se debe subscribir), para que cuando se llame al método `connect` de cada objeto, se pueda realizar la conexión satisfactoriamente.

En el cuadro 4.9 se muestra cómo se define uno de estos objetos y el método para establecer la conexión. El resto de objetos se realiza de la misma manera.

```
API.CameraRos = function (config) {
    var conf = config || {};
    this.ros;
    var self = this;
    this.connect = function (topic){
          self.ros = new ROSLIB.Ros({
              url : "ws://"+config.server.dir+":"+config.server.port
          });
          self.ros.on('connection', function() {
              console.log("Connect websocket Camera");
          });
          self.ros.on('error', function(error) {
              console.log("Error to connect websocket");
          });
     }
```

Cuadro 4.9: Establecimiento de la conexión ROS

Como se puede ver, la conexión se realiza de la misma forma que la conexión ROS del visor CamVizWeb. La única diferencia es que en esta ocasión no se usa una variable global para toda conexión ROS mediante `self.ros`, ya que como se ha indicado anteriormente, cada objeto tendrá su propia conexión. Por otro lado, el método `connect` se define como `this.connect = function(config)` por la misma razón, es decir cada objeto tiene su propio método `connect` que se invocará cuando se inicialice cada conexión. Esta inicialización se realiza mediante el código del cuadro 4.10.

```
...
laser= new API.LaserRos(configlaser);
motors= new API.MotorsRos(configmotors);
cameraleft = new API.CameraRos (configcaml);
cameraright = new API.CameraRos(configcamr);
pose3d = new API.Pose3DRos(configpos);

laser.connect();
motors.connect();
cameraleft.connect();
cameraright.connect();
pose3d.connect();
...
```

Cuadro 4.10: Creación de cada uno de los objetos de la *API* y su posterior inicialización

Las cinco primeras líneas de código corresponden a la creación de los objetos. Su número coincide con los mensajes que se van a recibir y enviar. Las siguientes cinco líneas corresponden a la invocación del método para inicializar la conexión.

4.2.3.2. Estructura de los mensajes

Se van a utilizar cuatro mensajes de los tipos predeterminados de ROS y definidos en su documentación[8]. Para obtener las imágenes de las cámaras, al igual que en el visor CamVizWeb, se va a usar el tipo `sensor_msgs/CompressedImage` y se incluye dentro del método `connect` de `API.CameraRos`.

El mensaje para obtener el sensor láser es del tipo `sensor_msgs/LaserScan`, que

[8] `http://wiki.ros.org/common_msgs`

ofrece la información obtenida mediante el driver incorporado en el TurtleBot. En el cuadro 4.11 se muestra la definición de este mensaje y forma parte del método `connect` de `API.LaserRos`.

```
self.roslaser = new ROSLIB.Topic({
          ros : self.ros,
          name : config.topic,
          messageType : "sensor_msgs/LaserScan"
});
```

Cuadro 4.11: Definición del mensaje para obtener la información del sensor láser

Para definir el mensaje que proporciona la posición del robot en cada momento se va a utilizar `nav_msgs/Odometry`, que proporciona la posición mediante la clase "Pose3D". Esta clase está formada por las coordenadas "X", "Y" y "Z" para ubicar el robot, la coordenada homogénea "h" y el cuaternión "q0", "q1", "q2" y "q3" para obtener su orientación. En el cuadro 4.12 se muestra la definición de este mensaje, que como en los casos anteriores se define dentro del método `connect` del objeto `API.Pose3DRos`.

```
self.rospose = new ROSLIB.Topic({
       ros : self.ros,
       name : config.topic,
       messageType : "nav_msgs/Odometry"
});
```

Cuadro 4.12: Definición del mensaje para obtener el posicionamiento del TurtleBot

Ahora solo falta por definir el mensaje que se transmite desde el visor mediante *Publisher*. Este mensaje es el que enviará el movimiento indicado con el teleoperador al TurtleBot y es del formato de ROS `geometry_msgs/Twist`. Este tipo de ROS permite transmitir tanto la velocidad lineal en tres dimensiones como la velocidad angular en las mismas tres dimensiones. En el cuadro 4.13 se muestra cómo se define este mensaje dentro del método `connect` del objeto `API.MotorsRos`.

```
self.rosMotors = new ROSLIB.Topic({
       ros : self.ros,
       name : config.topic,
       messageType : "geometry_msgs/Twist"
});
```

Cuadro 4.13: Definición del mensaje para transmitir el movimiento proporcionado por el teleoperador

4.2.3.3. Subscripción a los *Topic* y tratamiento de la información

Cada tipo de mensaje recibido es tratado de manera independiente. Se ha definido el método `startStreaming` para cada uno de los objetos de la API.

4.2.3.3.1. Subscripción y tratamientos de las cámaras

Para el caso de las imágenes obtenidas de las cámaras, el tratamiento y subscripción es muy similar al explicado en la sección 4.1.3.3, por lo que no se vuelve a profundizar en ello, únicamente hacer hincapié en que un robot TurtleBot cuenta con dos cámaras (izquierda y derecha), por lo que se tiene un *Subscribe* para cada cámara y, por consiguiente, se tratan las imágenes recibidas de manera independiente. La figura 4.10 muestra la visualización de ambas cámaras.

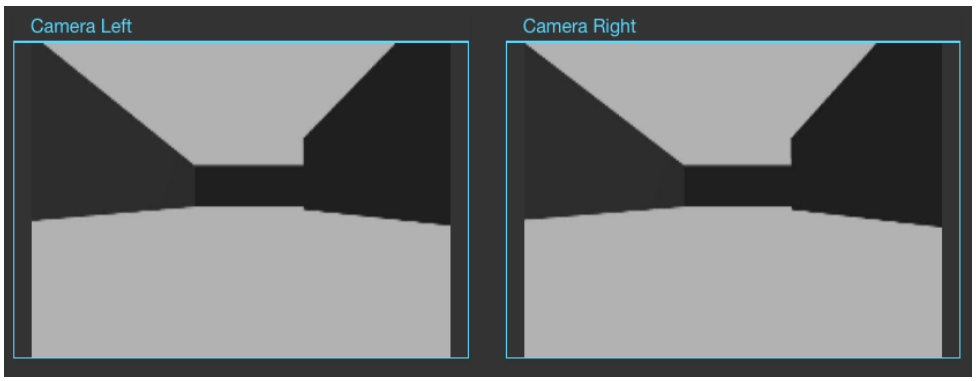


Figura 4.10: Visualización de la información de las cámaras

4.2.3.3.2. Subscripción y tratamiento del sensor láser

La subscripción del mensaje con el sensor láser se realiza de la misma forma que la cámara, sin embargo en este caso el tratamiento se realiza de manera diferente. El mensaje transmite la información acerca del ángulo mínimo y máximo del escaneo, el valor del rango mínimo y máximo, y el rango de datos. Con esta información se debe recrear el

escaneo tanto en 2D como en 3D. Cabe indicar que el escaneo 3D solo se muestra en caso de que así se desee, ya que únicamente se mostrará en el momento que se active la representación 3D del TurtleBot en el visor.

Para visualizar el escaneo en 2D, se debe tener en cuenta el rango del barrido del escaner incorporado en el TurtleBot, que es la resta entre el máximo ángulo barrido y el mínimo, el número de ángulos que ha escaneado y por último el valor del escaner en cada ángulo, que da lugar a un `array` de datos. Con esta información, ya es posible representar obtenida por el sensor láser en 2D.

Para representar el TurtelBot en 3D y el escaneo en 3D se utiliza `WebGL` y la biblioteca `Three.js`. Como se explicará en el método `drawLaser` se utiliza el objeto `new THREE.BufferGeometry()` para representar el escaneo. A este objeto hay que indicarle las componentes de cada vector, luego divide el *array* de datos pasado en vectores y dibuja triángulos utilizando tres vectores consecutivos del *buffer*. En este caso, el vector va a contar con 3 componentes ya que es en 3D ("x", "y" y "z"), y cada triangulo está dibujado mediante tres vértices que son dos puntos escaneados por el láser y el origen de coordenadas.

Tras obtener todos los datos necesarios para representar la información obtenida por el sensor láser se almacenan en la variable `laser` todos los datos necesarios para realizar la representación y se invoca al método `drawLaser`, tal y como se puede ver en el método `startStreaming` que se muestra en el cuadro 4.14.

```
this.startStreaming = function(convertUpAxis,canv2dWidth, scale3d, model){
    self.roslaser.subscribe(function(message){
        ...
        ...
        var laser = {}
        laser.distanceData = dist;
        laser.numLaser = numlaser;
        laser.maxAngle = maxAngle;
        laser.minAngle = minAngle;
        laser.maxRange = maxRange;
        laser.minRange = minRange;
        laser.canv2dData = array2d;
        laser.array3dData = array3d;
```

```
        drawLaser(laser,model,canv2dWidth);
    });
};
```

Cuadro 4.14: Subscripción y tratamiento del mensaje para su visualización 3D label

En el cuadro 4.15 se muestra el método para dibujar el escaneo del láser. El escaneo en 2D se dibuja sobre el elemento de HTML5 `Canvas` utilizando sus métodos para representar gráficos y se puede visualizar en la figura 4.11. El escaneo en 3D se realiza, como se ha indicado anteriormente, mediante la biblioteca de `WebGL Three.js` y el objeto `THREE.BufferGeometry()`, pero únicamente se mostrará en caso de que este activa la visualización del modelo 3D, tal y como muestra la figura 4.12.

```
function drawLaser(laser, model){
    var dist = laser.canv2dData;
    var lasercanv = document.getElementById("laser");
    var ctx = lasercanv.getContext("2d");
    ctx.beginPath();
    ctx.clearRect(0,0,lasercanv.width,lasercanv.height);
    ctx.fillRect(0,0,lasercanv.width,lasercanv.height);
    ctx.strokeStyle="white";
    ctx.moveTo(dist[0], dist[1]);
    for (var i = 2;i<dist.length; i = i+2 ){
        ctx.lineTo(dist[i], dist[i+1]);
    }
    ctx.moveTo(lasercanv.width/2, lasercanv.height);
    ctx.lineTo(lasercanv.width/2, lasercanv.height-10);
    ctx.stroke();

    if (model.active){
        var geometry = new THREE.BufferGeometry();
        geometry.addAttribute( 'position',
            new THREE.BufferAttribute(new Float32Array(laser.array3dData),3));
        var material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
        material.transparent = true;
        material.opacity=0.5;
        material.side = THREE.DoubleSide;
```

```
        var las = new THREE.Mesh( geometry, material );
        if (model.laser){
            model.robot.remove(model.laser);
        };
        model.robot.add(las);
        model.laser = las;
    }
}
```

Cuadro 4.15: Método para dibujar el escaneo del sensor láser

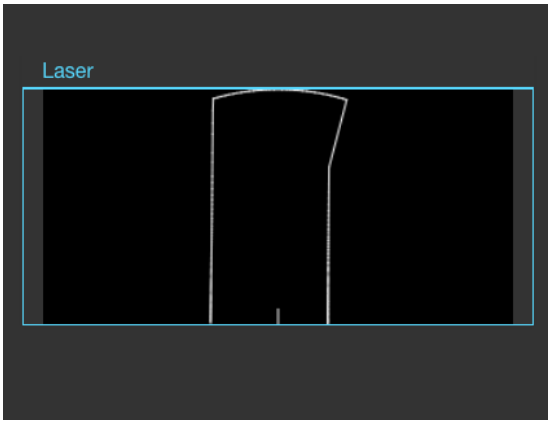


Figura 4.11: Visualización de la información del láser en 2D

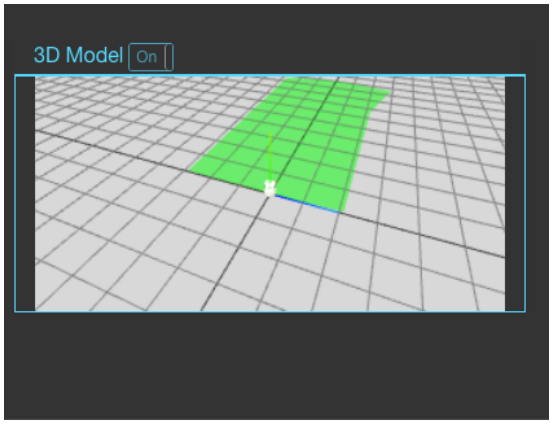


Figura 4.12: Visualización de la información del láser en 3D

4.2.3.3.3. Subscripción y tratamiento de la odometría

El último *topic* al que se subscribe es el encargado de proporcionar el mensaje con la posición y orientación del TurtleBot mediante la clase "Pose3D". Esta clase proporciona la posición a través de las coordenadas "x", "y" y "z", y la orientación mediante los cuaterniones "q0", "q1", "q2" y "q3" tras realizar las conversiones oportunas.

Únicamente se utiliza este mensaje en caso de que se desee mostrar la representación en 3D del TurtleBot mediante WebGL (aunque si se realiza la subscripción al *topic* para que cuando se desee mostrar no haya retardos). La subscripción se realiza de la misma forma que en los casos anteriores, es decir mediante el método `startStreaming`

```
this.startStreaming = function(model){
```

```
    self.rospose.subscribe(function(message){
        if (lastPose != message.pose){
            if (model.active) {
                var orientation = getPose3D(message.pose.pose.orientation);
                model.robot.position.set(message.pose.pose.position.x,
                    message.pose.pose.position.z, -message.pose.pose.position.y);
                model.robot.rotation.y=(orientation.yaw);
                model.robot.updateMatrix();
            }
        }
        lastPose = message.pose;
    });
};
```

Cuadro 4.16: Subscripción y tratamiento del mensaje para obtener la posición

En el cuadro 4.16 se puede visualizar cómo se realiza la subscripción. La variable `lastpose` se define en el constructor de objeto `API.Pose3DRos` y toma el valor de la posición obtenida en el anterior mensaje recibido. Con esta variable se detecta cuándo la posición ha sufrido cambios de modo que se desechen los mensajes que no envíen información sobre un cambio en la posición. En caso que el mensaje recibido transmita información sobre una nueva ubicación, se comprueba si está activa la representación 3D del robot. Si así lo fuera, se realiza la conversión de los cuaterniones para obtener la orientación del robot que se explicará en la sección 5.5.2.3. Tras esto, se pinta la representación del TurtleBot en la nueva ubicación, teniendo en cuenta que el sistema de coordenadas utilizado por WebGL no es el mismo que el del robot, por lo que la coordenada "y" de WebGL corresponde a la coordenada "z" del robot, y la coordenada "z" de la representación corresponde a la "-y" del robot. Posteriormente se establece la nueva orientación, que en el caso del TurtleBot solo rotará alrededor del eje vertical, por lo que solo es necesario indicar la orientación en el eje de coordenadas "y". Para finalizar, se actualiza la visualización de la representación del TurtleBot y se almacena esta posición en la variable `lastPose` para indicar que es el último movimiento realizado.

En la figura 4.12 se puede ver la representación del TurtleBot que se visualiza en el visor.

4.2.3.4. Publicación del mensaje con la información del teleoperador

Tras describir cómo realizar la subscripción a los mensajes publicados por el TurtleBot y posteriormente tratarlos para mostrar lo que se recibe, hay que explicar cómo se crea y posteriormente se publica el mensaje con la información de los motores.

Los motores se controlan mediante una interfaz creada usando WebGL y la biblioteca `Three.js`. Esta interfaz cuenta con una línea vertical y otra horizontal que se cruzan formando una cruz, y un circulo rojo para indicar el movimiento, de manera que la línea vertical marca el movimiento hacia delante y hacia atras del robot (la velocidad lineal) y la línea horizontal hacia donde debe girar (la velocidad angular). En la figura 4.13 se muestra esta interfaz.

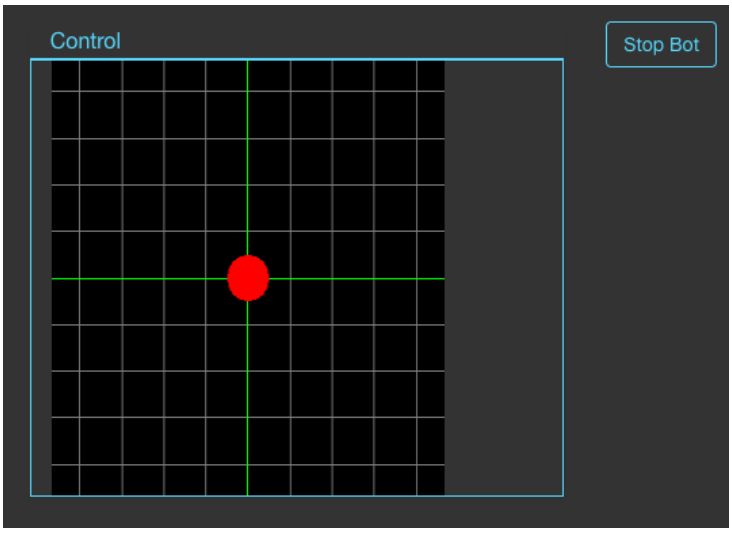


Figura 4.13: Interfaz para el control de los motores

Para transmitir la información de los motores se ha creado el método `setMotors` dentro del objeto `API.MotorsRos`. A este método se le introducen como parámetros las información de la velocidad lineal y la velocidad angular, y prepara el mensaje para que tenga la estructura del tipo predeterminado de ROS `geometry_msgs/Twist`[9] para posteriormente publicarlo mediante los *Publisher* de ROS.

```
this.setMotors = function(v,w){
    var motorsMessage = new ROSLIB.Message({
        linear: v,
```

[9] `http://docs.ros.org/melodic/api/geometry_msgs/html/msg/Twist.html`

```
        angular: w
    });
    self.rosMotors.publish(motorsMessage);
    }
```

Cuadro 4.17: Preparación y publicación del mensaje con el control de los motores

El código del cuadro 4.17 realiza la preparación y la publicación de los mensajes para transmitir la información de los motores al TurtleBot. Primero se prepara el mensaje mediante la utilización del objeto de *roslibjs* `ROSLIB.Message`, definiendo la velocidad lineal y angular con los parámetros introducidos en la invocación al método. Definido el mensaje a transmitir, se utiliza el método de *roslibjs* `publish` introduciendo como parámetro el mensaje definido con la información de los motores.

Sin embargo, con este método solo se transmite un mensaje con el control de los motores, y lo que se desea es estar continuamente enviando esta información para que el TurtleBot sepa en todo momento cómo se debe comportar. Para esto se utiliza el método de HTML `setInterval`[10] que proporciona la ejecución de código cuando ha transcurrido el periodo de tiempo indicado.

```
setInterval(function (){
        motors.setMotors(v,w)
},1);
```

Cuadro 4.18: Invocación cada milisegundo al método para publicar el mensaje con los motores

En el cuadro 4.18 se puede ver que cada milisegundo se invoca al método `setMotors` introduciendo la velocidad lineal y angular establecida en ese momento para que se publique y el TurtleBot pueda recibirla.

4.2.4. Configuración

Al igual que en el visor CamVizWeb, la configuración puede ser realizada via un fichero externo del tipo "YAML" o en tiempo de ejecución con el menu configuración de la interfaz gráfica. Los parámetros configurables para este visor son los siguientes:

[10] `https://www.w3schools.com/jsref/met_win_setinterval.asp`

- *Middleware* utilizado para realizar la comunicación. Por defecto es ICE.
- Dirección IP para conectar el visor a la cámara izquierda del TurtleBot. Por defecto "localhost".
- Puerto para conectar el visor a la cámara izquierda del TurtleBot. Por defecto es "9090".
- Endpoint para conectar el visor a la cámara izquierda del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ICE, por lo que si se indica ROS, se ignora. Por defecto es "CameraL".
- Topic para conectar el visor a la cámara izquierda del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ROS, por lo que se indica ICE, se ignora. Por defecto es "/TurtleBotROS/cameraL/image_raw/compressed".
- Dirección IP para conectar el visor a la cámara derecha del TurtleBot. Por defecto "localhost".
- Puerto para conectar el visor a la cámara derecha del TurtleBot. Por defecto es "9090".
- Endpoint para conectar el visor a la cámara derecha del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ICE, por lo que si se indica ROS, se ignora. Por defecto es "CameraR".
- Topic para conectar el visor a la cámara derecha del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ROS, por lo que se indica ICE, se ignora. Por defecto es "/TurtleBotROS/cameraR/image_raw/compressed".
- Dirección IP para conectar el visor con los motores del TurtleBot. Por defecto "localhost".
- Puerto para conectar el visor con los motores del TurtleBot. Por defecto es "9090".
- Endpoint para conectar el visor con los motores del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ICE, por lo que si se indica ROS, se ignora. Por defecto es "Motors".

- Topic para conectar el visor con los motores del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ROS, por lo que se indica ICE, se ignora. Por defecto es "/TurtleBotROS/mobile_base/commands/velocity".
- Dirección IP para conectar el visor a la odometría del TurtleBot. Por defecto "localhost".
- Puerto para conectar el visor a la odometría del TurtleBot. Por defecto es "9090".
- Endpoint para conectar el visor a la odometría del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ICE, por lo que si se indica ROS, se ignora. Por defecto es "Pose3D".
- Topic para conectar el visor a la odometría del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ROS, por lo que se indica ICE, se ignora. Por defecto es "/TurtleBotROS/odom".
- Dirección IP para conectar el visor al sensor láser del TurtleBot. Por defecto "localhost".
- Puerto para conectar el visor al sensor láser del TurtleBot. Por defecto es "9090".
- Endpoint para conectar el visor al sensor láser del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ICE, por lo que si se indica ROS, se ignora. Por defecto es "Laser"
- Topic para conectar el visor al sensor láser del TurtleBot. Este parámetro solo se utiliza para realizar la comunicación mediante ROS, por lo que se indica ICE, se ignora. Por defecto es "/TurtleBotROS/laser/scan".
- Máxima velocidad lineal permitida.
- Máxima velocidad angular permitida.

4.2.5. Experimentos

El visor puede ser ejecutado a través de un navegador web o mediante Electron. Para realizar los experimentos, se va a utilizar el simulador Gazebo[11] junto con la simulación

[11] `http://wiki.ros.org/gazebo_ros`

del robot TurtleBot de ROS[12].

En un primer terminal o consola se debe ejecutar el servidor intermedio de ROS que se puede ver en el esquema.

```
#>roslaunch rosbridge_server rosbridge_websocket.launch
```

Cuadro 4.19: Ejecución del servidor intermedio

En un segundo terminal se ejecuta Gazebo con la simulación del TurtleBot.

```
#>roslaunch gazebo_ros kobuki-simple-ROS.launch
```

Cuadro 4.20: Ejecución del driver de ROS label

En el tercer terminal se ejecuta TurtleBotVizWeb.

- Como aplicación web utilizando Node.js[13]

```
#>node run.js
Se arranca el navegador y se introduce la URL http://localhost:7777/
```

Cuadro 4.21: Ejecución con Node.js

- Como aplicación de escritorio con Electron[14]

```
#>npm install
#>npm start
```

Cuadro 4.22: Ejecución con Electron

[12] http://wiki.ros.org/action/show/Robots/TurtleBot?action=show&redirect=TurtleBot
[13] https://www.youtube.com/watch?v=aK68GONhsbw
[14] https://www.youtube.com/watch?v=ko9he24Uvcw

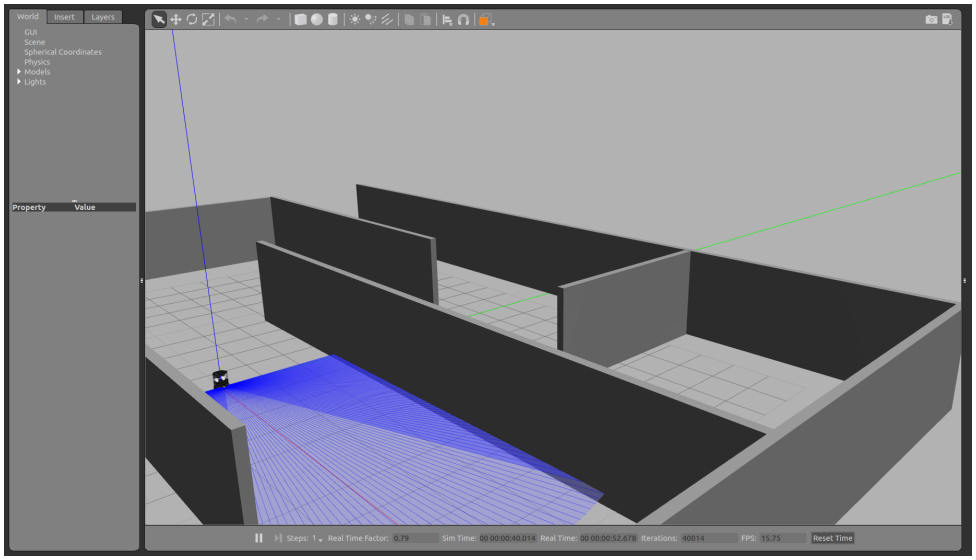


Figura 4.14: Simulación del robot TurtleBot con Gazebo

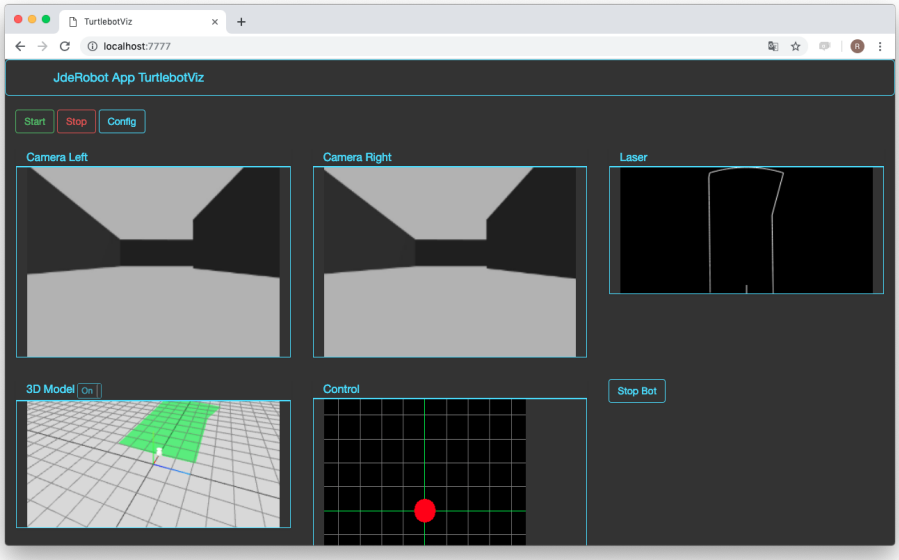


Figura 4.15: TurtleBotVizWeb ejecutado en un navegador web

Este driver también se ha probado con un TurtleBot real[15]. Para ello se han seguido los siguientes pasos:

En un primer terminal o consola se debe ejecutar el servidor intermedio de ROS que se puede ver en el esquema.

[15] `https://www.youtube.com/watch?v=U71PYPtEZIo`

```
#>roslaunch rosbridge_server rosbridge_websocket.launch
```

Cuadro 4.23: Ejecución del servidor intermedio

En un segundo terminal se ejecuta los drivers de ROS necesario para controlar los motores del TurtleBot y obtener el sensor láser.

```
#>roslaunch turtlebot-hokuyo.launch
```

Cuadro 4.24: Ejecución de los drivers para los motores y el láser

En un tercer terminal se ejecuta los drivers de ROS para obtener las imágenes de las cámaras incorporadas en el TurtleBot.

```
#>roslaunch usb_cam usb_cam.launch
```

Cuadro 4.25: Ejecución del driver de ROS para las cámaras

En el cuarto terminal se ejecuta TurtleBotVizWeb.

```
#>npm install
#>npm start
```

Cuadro 4.26: Ejecución con Electron

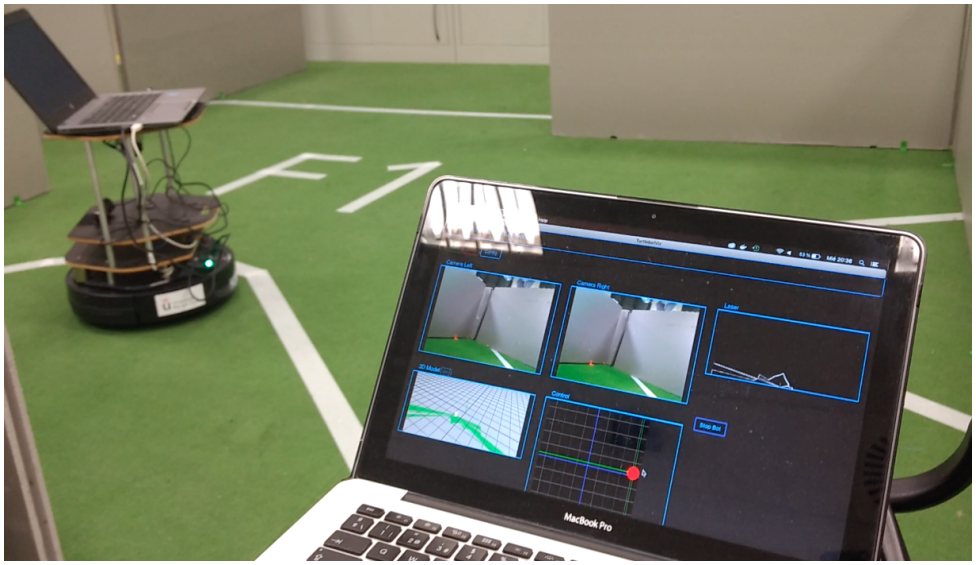


Figura 4.16: TurtleBotVizWeb conectado a un TurtleBot físico

4.3. DroneVizWeb

Se trata de un visor y teleoperador para Drones desarrollado con tecnologías web, que usa los middleware de comunicación ICE y ROS y puede ser ejecutado con Electron o en un navegador web[16].

4.3.1. Diseño

Al igual que en los visores anteriores, la versión previa de esta herramienta solo contaba con ICE como *middleware* de comunicación. Ahora es capaz también de conectarse con Drones que se comuniquen mediante ROS y no solo mediante ICE. Para lograr la conectividad con el drone mediante ROS ha sido necesaria la utilización de MavROS[17] que es un paquete que proporciona una interfaz para el control de drones utilizando el protocolo MAVLink[18]. La figura 4.17 muestra el diagrama de bloques del visor tras la mejora realizada en este trabajo.

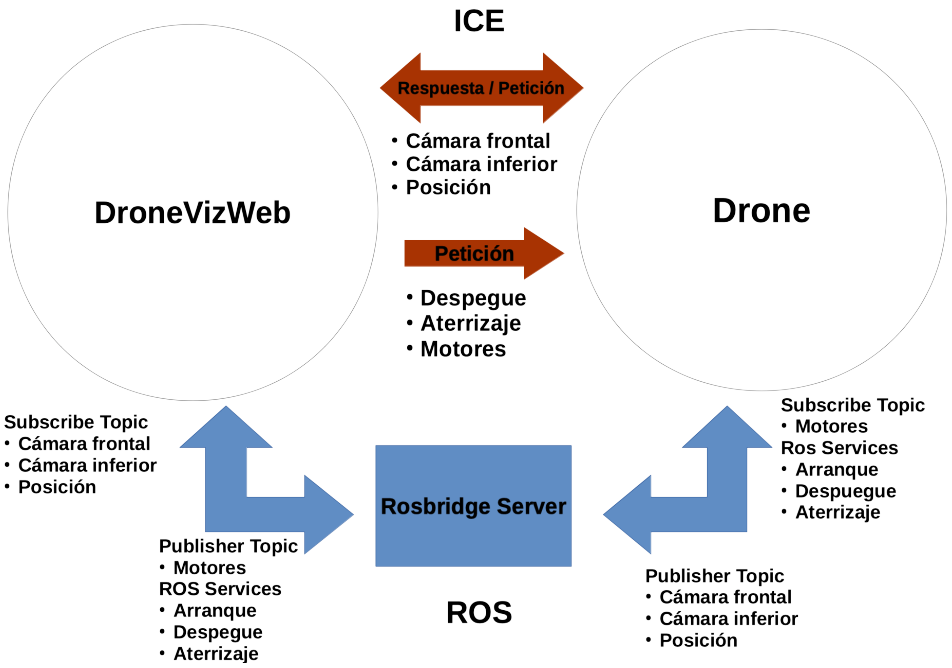


Figura 4.17: Diagrama de bloques actual del sistema

[16] https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez/tree/master/DroneVizWeb

[17] http://wiki.ros.org/mavros

[18] https://mavlink.io/en/

En la figura 4.18 muestra la estructura interna de la aplicación previa de una manera detallada tras realizar la modificación.

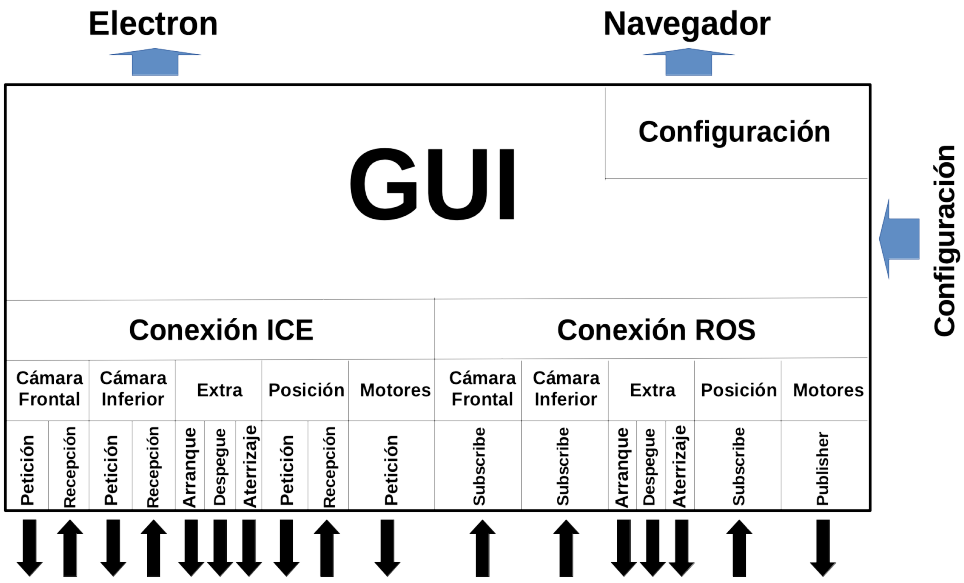


Figura 4.18: Estructura actual de la herramienta

Al igual que en las herramientas anteriores, está dividida en dos partes: la interfaz gráfica y las conexiones. La interfaz gráfica contiene los elementos necesarios para visualizar los datos recibidos y teleoperar el drone, además de contener un menú para configurar en tiempo de ejecución el visor.

Para realizar las conexiones se utiliza los *middleware* de comunicación ROS y ICE. La versión de ICE es la existente de manera previa por lo que no es objeto de este trabajo profundizar en su funcionamiento. El visor recibe dos mensajes simultáneamente procedentes del drone, la imagen de la cámara (ya sea la cámara frontal o inferior) y la posición del drone. El visor envía la información sobre el control de los motores y las peticiones de arranque, despegue y aterrizaje del drone a la que se llama "Extra" ya que se realiza mediante la misma conexión.

En el caso de ROS, se utiliza los *subscribe* para recibir los mensajes procedentes del robot, los *publisher* para transmitir la información de los motores y los servicios de ROS para solicitar el arranque, despegue y aterrizaje al drone.

Por otro lado, al igual que para el resto de herramientas que componen este trabajo, se le añade la posibilidad de ejecutarse mediante el entorno Electron y la configuración

previa mediante un archivo externo en formato “YAML”.

4.3.2. Interfaz gráfica

Se mantiene la interfaz gráfica existente en la versión anterior de la herramienta. Esta interfaz cuenta con un elemento `Canvas` de HTML5 donde se muestra la imagen procedente de la cámara, sobre este elemento se incorporan tres botones, el primer botón es el utilizado para indicar el arranque y despegue, el segundo botón es el que se utiliza para indicar el aterrizaje y el ultimo botón se utiliza para solicitar el cambio de cámara a mostrar (si en ese momento se muestra la cámara frontal, se cambia a la inferior y viceversa). Por otro lado, dentro de este elemento también se incorporan dos *joysticks* para controlar el drone, estos *joysticks* están creados usando también elementos `Canvas`, siendo el *joystick* izquierdo el que controla la velocidad y el derecho el que controla la altura del drone. La interfaz gráfica cuenta con otro elemento `Canvas` donde se muestra la representación tridimensional del drone y su movimiento, así como tres botones para comenzar la conexión, para parar la conexión y para abrir la ventana con el menú para realizar la configuración en tiempo de ejecución.

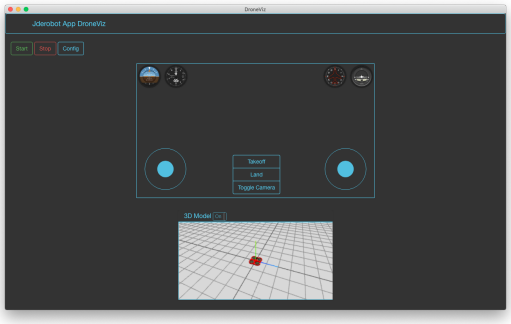


Figura 4.19: Interfaz gráfica de DroneVizWeb

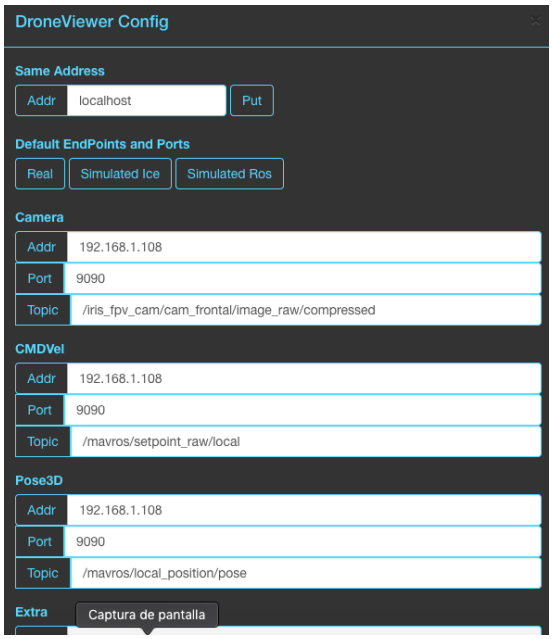


Figura 4.20: Menú de configuración de DroneVizWeb

En la figura 4.19 se muestra la interfaz gráfica utilizada. Para poder configurar la

conexión mediante ROS, se vuelve a utilizar lo indicado en la sección 4.2.2. En la figura 4.20 se muestra el menú de configuración para realizar la configuración en tiempo de ejecución.

4.3.3. Conexiones

Con la modificación realizada la herramienta permite el uso de los middleware ICE y ROS para realizar las comunicaciones con un funcionamiento igual de eficiente.

Al igual que en el visor TurtleBotVizWeb, se reutilza el *API* de ICE para definir los métodos y objetos necesarios para realizar la conexión mediante ROS.

4.3.3.1. Establecimiento de las conexiones

Por cada tipo de mensaje que se envía o se recibe se realiza una conexión, en el caso de este visor simultáneamente se tendrán activas 5 conexiones (dos cámaras, posición, motores y extra) como se ha podido ver en la figura 4.18.

Dado que se establece la conexión de la misma forma que la explicada en la sección 4.2.3.1, no se va a profundizar más al respecto, simplemente indicar que los objetos creados en la `API` para realizar las conexiones en esta ocasión son `API.CameraRos, API.MotorsRos, API.ExtraRos y API.Pose3DRos` que corresponden con la conexión para la cámara, para el control de los motores, para el arranque, despegue y aterrizaje, y para la posición del drone respectivamente, con el método `connect` asociado a cada objeto y explicado en la sección anteriormente referenciada.

4.3.3.2. Estructura de los mensajes

Para este visor se van a utilizar los mensajes predeterminados de ROS para obtener las imágenes de la cámara y obtener la posición. Por otro lado, para transmitir las peticiones de arranque, despegue, aterrizaje y el control de los motores se utilizan los mensajes facilitados en el paquete MavROS[19]. Para definir los mensajes se va a utilizar los mismos métodos que se usan en las herramientas anteriores.

El mensaje que transmite las imágenes de las cámaras son del tipo `sensor_msgs/CompressedImage`

[19] `http://docs.ros.org/melodic/api/mavros_msgs/html/index-msg.html`

y se define con el código que se muestra en el cuadro 4.27. Esta definición está incluida dentro del método `connect` del objeto `API.CameraRos`.

```
self.roscam = new ROSLIB.Topic({
          ros : self.ros,
          name : config.topic,
          messageType : "sensor_msgs/CompressedImage"
});
```

Cuadro 4.27: Definición del mensaje para la información de las cámaras

La información sobre el posicionamiento del drone se transmite mediante `nav_msgs/Odometry`, que transmite la posición mediante la clase "Pose3D". En el cuadro 4.28 se muestra la definición de este mensaje que forma parte del método `connect` del objeto `API.Pose3DRos`

```
self.rospose = new ROSLIB.Topic({
       ros : self.ros,
       name : config.topic,
       messageType : "nav_msgs/Odometry"
});
```

Cuadro 4.28: Definición del mensaje para obtener el posicionamiento del drone

La transmisión del control de los motores se envía mediante el tipo facilitado en el paquete MavROS `mavros_msgs/PositionTarget`. Este mensaje transmite la información acerca de la posición para la coordenada "x", "y" y "z", la velocidad lineal en cada una de estos ejes de coordenadas, la aceleración en cada eje y la rotación entorno al eje "y". Con este mensaje se puede enviar toda la información para teleoperar un drone. Además este mensaje incluye una marca de tiempo para sincronizar el drone con el teleoperador. Como en los casos anteriores, esta definición se incluye en el método `connect` del objeto `API.MotorsRos` y se puede ver en el cuadro 4.29.

```
self.rosMotors = new ROSLIB.Topic({
   ros : self.ros,
   name : config.topic,
   messageType : "mavros_msgs/PositionTarget"
});
```

Cuadro 4.29: Definición del mensaje para controlar los motores del drone

Finalmente, los mensajes parta solicitar el arranque, despegue y aterrizaje se definen en el método `connect` del objeto `API.ExtraRos` y se muestran en el cuadro 4.30. El arranque se transmite mediante el mensaje del tipo `mavros_msgs/CommandBool`, que envía únicamente un *booleano* siendo el valor *True* el que indica que arranque y *False* el que indica que se apague el drone. Por otro lado, para el despegue y aterrizaje se utiliza el mismo tipo de mensaje contenido en el paquete de MavROS que es `mavros_msgs/CommandTOL`. En este mensaje se envía información sobre la latitud, longitud, altitud y la rotación sobre el eje "y".

```
self.rosArming = new ROSLIB.Service({
    ros : self.ros,
    name : config.topic_arming,
    messageType : "mavros_msgs/CommandBool"
 });

self.rosTakeoff = new ROSLIB.Service({
    ros : self.ros,
    name : config.topic_takeoff,
    messageType : "mavros_msgs/CommandTOL"
});

self.rosLand = new ROSLIB.Service({
    ros : self.ros,
    name : config.topic_land,
    messageType : "mavros_msgs/CommandTOL"
});
```

Cuadro 4.30: Definición de los mensajes para arrancar, despegar y aterrizar

4.3.3.3. Subscripción a los *Topic* y tratamiento de la información

Mediante los *subscribe* de ROS se obtienen las imágenes de las cámaras, así como la odometría del drone en cada momento. Para ello, los objetos `API.CameraRos` y `API.Pose3DRos` cuentan con el método `startStreaming`, sobre el cual se realiza la subscripción al *topic* y se tratan los mensajes recibidos. Dado que el funcionamiento es el mismo que el explicado en la sección 4.1.3.3 para el caso de las cámaras y en la sección

4.2.3.3.3 para la información odometrica, no se profundiza de nuevo.

En la figura 4.21 se puede ver la representación del drone que se muestra en el visor.

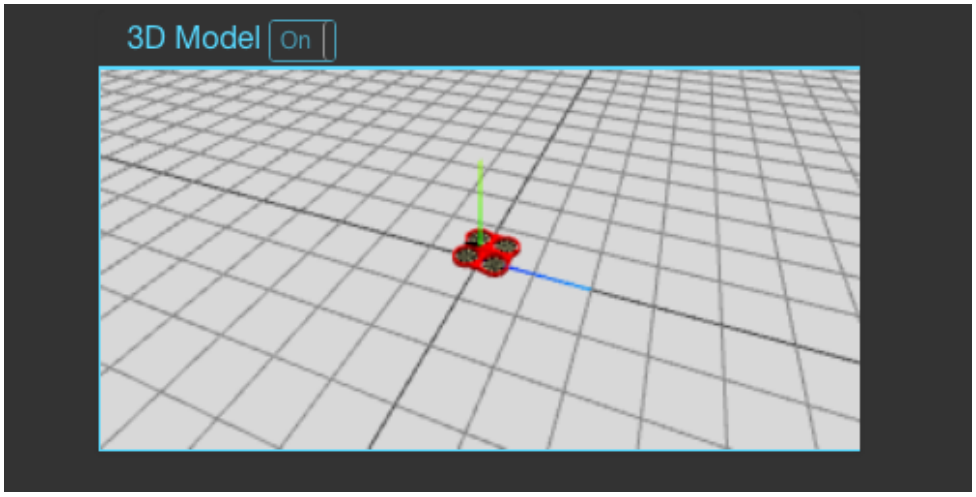


Figura 4.21: Representación 3D del drone

4.3.3.4. Publicación del mensaje con la información del teleoperador

Para enviar la información acerca del control de los motores del drone, se utilizan los *publisher* de ROS. Los motores se controlan mediante dos *joysticks* que se proyectan sobre el elemento `canvas` donde se muestra la imagen recibida por la cámara, tal y como se ha indicado en la sección 4.3.2. El *joystick* izquierdo maneja la velocidad lineal (adelante, atrás, izquierda y derecha), mientras que con el *joystick* derecho se indica la altura del drone.

Una vez se ha indicado esto, lo siguiente es crear el mensaje con la información de los motores y publicarlo. Para ello se ha definido el método `setMotors` dentro del objeto `API.MotorsRos` y que se muestra en el cuadro 4.31. Se va a utilizar el mensaje del tipo `mavros_msgs/PositionTarget` proporcionado en el paquete MavROS.

```
self.setMotors = function(vel){
    var msg = {};
    var currentTime = new Date().getTime()-timestamp;
    var secs = Math.floor(currentTime/1000);
    var nsecs = Math.round(1000000000*(currentTime/1000-secs));
    msg.header = {};
```

```
    msg.header.stamp = {};
    msg.header.stamp.secs = secs;
    msg.header.stamp.nsecs = nsecs;
    msg.header.frame_id = "";
    msg.coordinate_frame = 8;
    msg.type_mask = 1475;
    msg.position = {};
    msg.position.x = 0;
    msg.position.y = 0;
    msg.position.z = vel.linearZ;
    msg.velocity = {};
    msg.velocity.x = vel.linearY;
    msg.velocity.y = vel.linearX;
    msg.velocity.z = 0;
    msg.acceleration_or_force = {};
    msg.acceleration_or_force.x = 0;
    msg.acceleration_or_force.y = 0;
    msg.acceleration_or_force.z = 0;
    msg.yaw = 0;
    msg.yaw_rate = 0;
    var motorsMessage = new ROSLIB.Message({
        header: msg.header,
        coordinate_frame: 8,
        type_mask: 1475,
        position: msg.position,
        velocity: msg.velocity,
        acceleration_or_force: msg.acceleration_or_force,
        yaw: 0,
        yaw_rate: 0
    });
    self.rosMotors.publish(motorsMessage);
}
```

Cuadro 4.31: Publicar el mensaje con la información del teleoperador

Sobre el código del cuadro 4.31, la información del teleoperador se le introduce al

método por parámetro. Dentro del método se crea la marca de tiempo para incorporarla a la cabecera del mensaje y se definen el resto de parámetros de la cabecera. Posteriormente se definen el resto de parámetros que forman parte del tipo de mensaje escogido, sin embargo para transmitir el control de los motores únicamente se utilizan los parámetros correspondientes a la velocidad en su componente "x" e "y" para transmitir la velocidad lineal, y la posición en su componente "z" para manejar la altura del drone. El resto de parámetros del mensaje no se utilizan por lo que se da el valor "0". Posteriormente se crea el mensaje que se va a transmitir mediante `new ROSLIB.Message` y se publica.

Con este método se transmite un mensaje, pero se desea estar publicando mensajes con la información del teleoperador de manera continua. Al igual que se explica en la sección 4.2.3.4, se va a utilizar el método de HTML `setInterval` como se muestra en el cuadro 4.32.

```
var cmdvelInterval = setInterval(function(){cmdvel.setMotors(cmdSend);},1);
```

Cuadro 4.32: Publicación continua del mensaje con la información del teleoperador

En la figura 4.22 se muestra el visor de las cámaras y los *joysticks* para controlar los motores.

Figura 4.22: Visualización de las cámaras y control de los motores

4.3.3.5. Arranque, Despegue y Aterrizaje mediante ROS Services

Para arrancar, despegar y aterrizar el drone es necesario usar los servicios de ROS proporcionados por el paquete MavROS. Estos servicios son complementarios entre si, es decir para despegar primero se ha debido arrancar los motores, para aterrizar ates se ha debido despegar y para apagar los motores antes se ha debido aterrizar. Teniendo en cuenta esto, es evidente que las llamadas a los métodos que realizan estas funciones deben realizarse de manera correlativa. Para lograr estos controles se han creado tres métodos dentro del objeto `API.ExtraRos` `Arming, takeoff y land`. Adicionalmente hay un cuarto método que es el encargado de cambiar el modo de pilotaje del drone y que de nuevo usa servicios de ROS, y se solicita una vez haya realizado el despegue del drone.

```
self.arming = function(arranque){
    var request = new ROSLIB.ServiceRequest({
            value: arranque
    })
    self.rosArming.callService(request, function(result){
        if (arming){
            self.takeoff();
        }
    });
}
```

Cuadro 4.33: Método para arrancar o apagar los motores del drone

En el código del cuadro 4.33 se muestra el método para realizar el arranque o apagado de los motores dependiendo lo que se envíe por parámetro al propio método. Si el valor del *booleano* es verdadero, entonces se solicita el arranque mediante el servicio de ROS, si por el contrario es falso, se solicita su apagado. Para realizar la solicitud se utiliza el método de `roslibjs callService` y se le pasa por parámetro la petición creada mediante el objeto `new ROSLIB.ServiceRequest` y un manejador para el resultado. En caso de que el resultado sea positivo y se solicite el arranque, se llama al método que realiza la solicitud de despegue, si se trata del apagado de los motores no se realiza nada, ya que el drone queda a la espera de una nueva solicitud de arranque.

```
self.takeoff = function(){
    var request = new ROSLIB.ServiceRequest({
```

```
        altitude:5.5,
        latitude: 0,
        longitude: 0,
        min_pitch: 0,
        yaw: 0
    });
    self.rosTakeoff.callService(request, function(result) {
        console.log("TakeOff Done");
        self.setmode();
    });
}
```

Cuadro 4.34: Método para arrancar o apagar los motores del drone

En el cuadro 4.34 se muestra el método para realizar el despegue. En la petición de despegue se debe enviar información acerca de la latitud, longitud, altitud, orientación entorno al eje "y" y el ángulo de vision. Dado que solo interesa indicar la altura, se dejan todos los valores a "0", menos la altitud que se fija un valor de "5.5". Posteriormente se vuelve a enviar la solicitud del mismo modo que para el arranque o apagado. En caso de que el resultado sea positivo, se llama al método para cambiar el modo de pilotaje del drone.

```
self.setmode = function(){
    var request = new ROSLIB.ServiceRequest({
        custom_mode: "OFFBOARD"
    });
    self.rosSet_Mode.callService(request, function(result){
    })
}
```

Cuadro 4.35: Método para cambiar el modo de pilotaje

En el cuadro 4.35 se muestra el método para cambiar el modo de pilotaje del drone. Una vez el drone ha despegado se le solicita que cambie el modo de piloto automático por el modo "OFFBOARD", de modo que se puedan controlar los motores mediante el teleoperador del visor. En este caso, una vez el resultado de la solicitud es afirmativo, no se realiza ninguna nueva acción al estar el drone ya listo para ser pilotado, por tanto el

drone queda a la espera de la información con el control de los motores o la petición de aterrizaje.

```
this.land = function(){
    var request = new ROSLIB.ServiceRequest({
        min_pitch: 0,
        yaw: 0,
        latitude: 0,
        longitude: 0,
        altitude:0,
    });
    self.rosLand.callService(request, function(result){
        self.arming(false);
    })
}
```

Cuadro 4.36: Método para cambiar el modo de pilotaje

El cuadro 4.36 contiene el código del método que realiza la solicitud de aterrizaje. Se puede apreciar que la petición es igual a la realizada para indicar el despegue, salvo que la altitud que se envía ahora es "0" y no "5.5" como en el despegue. Una vez que se envía la solicitud de aterrizaje y el resultado es positivo, se llama al método del cuadro 4.33 pasando por parámetro el *booleano false* para indicar que se solicita el apagado de los motores.

4.3.4. Configuración

Al igual que los dos visores anteriores, se puede realizar la configuración mediante la utilización de un archivo del tipo "YAML" o a través de la configuración en tiempo de ejecución mediante el menú de configuración de la interfaz gráfica que se explica en la sección 4.3.2.

Los parámetros configurables para este visor son los siguientes:

- *Middleware* utilizado para realizar la comunicación. Por defecto es Ros.
- Dirección IP para conectar el visor a la cámara frontal del drone. Por defecto "localhost".

- Puerto para conectar el visor a la cámara frontal del drone. Por defecto es “9090”.
- Endpoint para conectar el visor a la cámara frontal del drone (en caso de ROS será el topic utilizado para realizar esta conexión). Por defecto es “/iris_fpv_cam/cam_frontal/image_raw/compressed”
- Dirección IP para conectar el visor a la cámara inferior del drone. Por defecto “localhost”.
- Puerto para conectar el visor a la cámara inferior del drone. Por defecto es “9090”.
- Endpoint para conectar el visor a la cámara inferior del drone(en caso de ROS será el topic utilizado para realizar esta conexión). Por defecto es “/iris_fpv_cam/cam_ventral/image_raw/compressed”
- Dirección IP para conectar el visor a la odometría del drone. Por defecto “localhost”.
- Puerto para conectar el visor a la odometría del drone. Por defecto es “9090”.
- Endpoint para conectar el visor a la odometría del drone (en caso de ROS será el topic utilizado para realizar esta conexión). Por defecto es “/mavros/local_position/pose”
- Dirección IP para conectar el visor a los extras (arranque, despegue, etc) del drone. Por defecto “localhost”.
- Puerto para conectar el visor a los extras (arranque, despegue, etc) del drone. Por defecto es “9090”.
- Endpoint para conectar el visor a los extras (arranque, despegue, etc) del drone (en caso de ROS será el topic utilizado para realizar esta conexión). Por defecto es “ardrone_extra”
- Dirección IP para conectar el visor con los motores del drone. Por defecto “localhost”.
- Puerto para conectar el visor con los motores del drone. Por defecto es “9090”.
- Endpoint para conectar el visor con los motores del TurtleBot (en caso de ROS será el topic utilizado para realizar esta conexión). Por defecto es “/mavros/setpoint_raw/local”

4.3.5. Experimentos

Este visor puede ser ejecutado mediante un navegador web o Electron. Para realizar los experimentos, se va a utilizar el simulador Gazebo[20] junto con el drone PX4[21].

En un primer terminal o consola se debe ejecutar el servidor intermedio de ROS que se puede ver en el esquema.

```
#>roslaunch rosbridge_server rosbridge_websocket.launch
```

Cuadro 4.37: Ejecución del servidor intermedio

En un segundo terminal se ejecuta gazebo con la simulación del drone PX4.

```
#>roslaunch gazebo_ros follow\_road.launch
```

Cuadro 4.38: Ejecución de gazebo con el drone PX4 label

En el tercer terminal se ejecuta DroneVizWeb.

- Como aplicación web utilizando Node.js[22]

```
#>node run.js
Se arranca el navegador y se introduce la URL http://localhost:7777/
```

Cuadro 4.39: Ejecución con Node.js

- Como aplicación de escritorio con Electron[23]

```
#>npm install
#>npm start
```

Cuadro 4.40: Ejecución con Electron

[20] http://wiki.ros.org/gazebo_ros
[21] https://px4.io/
[22] https://www.youtube.com/watch?v=MwOHXEBYYYw
[23] https://www.youtube.com/watch?v=oGQ1mQ3r4sQ

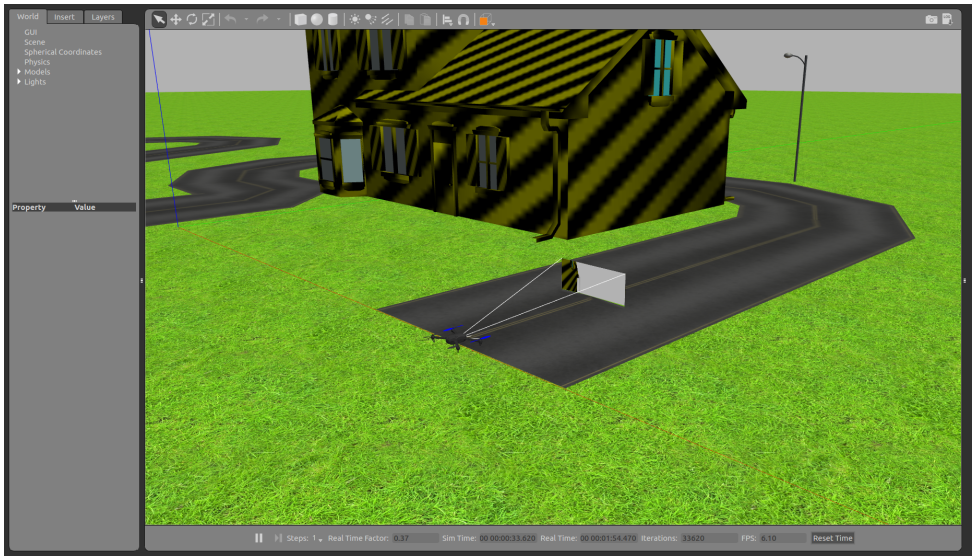


Figura 4.23: Simulación del drone con Gazebo

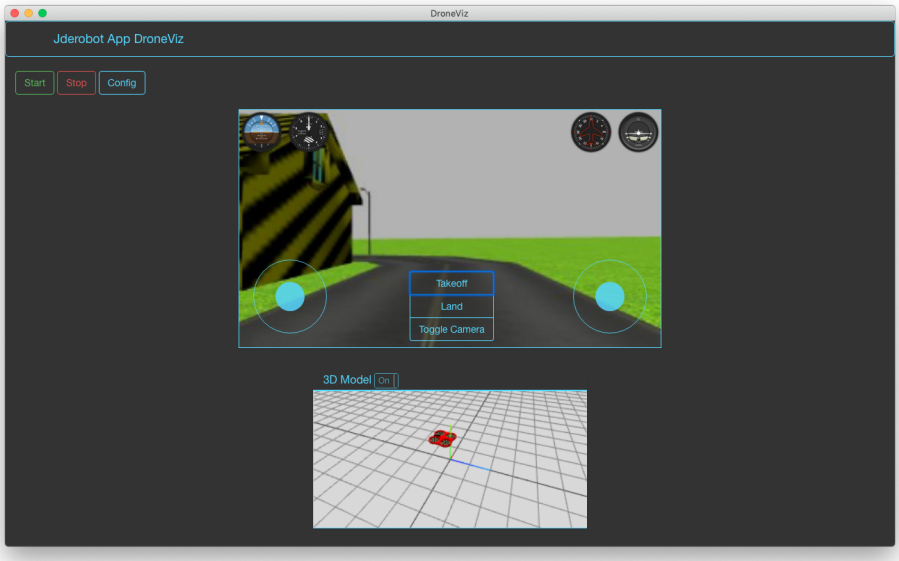


Figura 4.24: DroneVizWeb ejecutado en Electron

Capítulo 5

Visor web dinámico de objetos 3D

En este capítulo se trata la creación de un nuevo visor de primitivas 3D (objetos, puntos y segmentos) con tecnologías web. Este visor, al que se ha nombrado como 3DVizWeb[1], abre la posibilidad de obtener datos 3D por una o varias cámaras o sensores y mostrar una representación 3D de los mismos.

5.1. Diseño

Este visor está elaborado usando JavaScript y HTML5 como lenguajes de programación, y se usa el *middleware* ICE para la interconexión con los clientes que envían los datos a mostrar. En la figura 5.1 se muestra su diagrama de bloques:

[1] `https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez/tree/master/3DVizWeb`

Figura 5.1: Ejecución típica del visor 3D

En la figura 5.2 se muestra el diseño del visor de una manera más detallada:

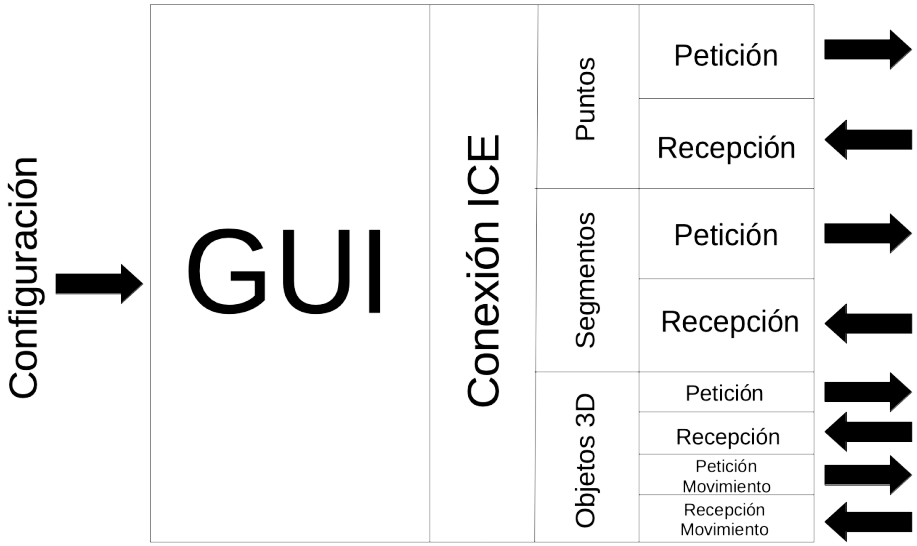


Figura 5.2: Diagrama de bloques del visor 3D

La herramienta está dividida en dos partes: la interfaz gráfica y las conexiones ICE.

La parte correspondiente a la interfaz gráfica es la que da origen al visor y marca cómo

se definen cada uno de los tipos de objetos que es capaz de mostrar. Hay tres tipos de primitivas 3D que el visor es capaz de visualizar: puntos, segmentos y objetos.

Las conexiones ICE con el servidor incluyen varias partes claramente diferenciadas. Estas partes corresponden a la petición y recepción de cada tipo de primitiva al servidor, haciendo especial hincapié en los objetos 3D que contarán con las peticiones y recepciones propias para mostrar nuevos objetos. El visor admite los formatos de archivo del tipo "dae" y del tipo"obj". Estos dos formatos son los más utilizados a la hora de crear objetos 3D y por tanto se considera que dando soporte a estos se permite usar cualquier objeto que se desee. Los objetos se podrán mover en 3D y el visor ofrece mecanismos eficientes para que las aplicaciones, mediante peticiones y recepciones, puedan mover los objetos existentes en la escena que se está visualizando.

5.2. Configuración

Dado que el visor en su interfaz gráfica únicamente muestra la escena donde se visualizan los objetos, es necesario utilizar una alternativa para configurar los datos de conexión (ip y puerto de escucha) y otros parámetros configurables (posición inicial de la cámara, tamaño de los puntos, tamaño de los segmentos y periodo de tiempo entre peticiones de cada tipo de objeto) antes de arrancar el visor. Para realizar está configuración se vuelve a utilizar un archivo en formato YAML, como se explica en la sección 4.1.6 de este trabajo.

Los parámetros que se pueden configurar para el visor son los siguientes:

- Dirección IP. Por defecto es "localhost"
- Puerto. Por defecto es "11000"
- Tiempo de refresco para la petición de puntos en milisegundos. Por defecto son 1000 ms
- Tiempo de refresco para la petición de segmentos en milisegundos. Por defecto son 1000 ms
- Tiempo de refresco para la petición de objetos 3D en milisegundos. Por defecto es 1 ms

- Tiempo de refresco para la petición del movimiento de objetos en milisegundos. Por defecto son 1000 ms
- Grosor de los segmentos en píxeles. Por defecto son 2 píxeles.
- Tamaño de los puntos en píxeles. Por defecto son 8 píxeles.
- Posición inicial de la cámara desde que se observa la escena. Está formada por las coordenadas “x”, “y” y “z” , siendo por defecto 50, 20 y 100 respectivamente.

5.3. Interfaz gráfica

La interfaz gráfica del visor está diseñada usando WebGL, y más concretamente usando la biblioteca `Three.js` descrita en capítulos anteriores. La interfaz gráfica de la aplicación, la ventana, la ocupa íntegramente el visor 3D, no tendrá ningún elemento adicional como pueden ser botones o diales. La escena 3D muestra inicialmente una rejilla que hará de plano horizontal, cuyo centro será la posición (0, 0, 0) del eje de coordenadas. Además ha sido necesario añadir iluminación y una cámara para visualizar la escena desde ella.

5.3.1. Escena base

El visor está contenido dentro de un elemento HTML `<canvas>`. En este elemento se crea una nueva escena usando la función de `Three.js` `THREE.Scene()` y se renderiza usando `THREE.WebGLRenderer()`. Posteriormente se crea una cámara con perspectiva utilizando `THREE.PerspectiveCamera()` ubicándola en una posición predeterminada (la indica en el archivo de configuración) y podrá ser controlada mediante teclado o ratón. Finalmente, para visualizar correctamente la escena se añaden varias luces puntuales a lo largo de la misma, así como una iluminación de ambiente utilizando `THREE.PointLight()` y `THREE.AmbientLight()` pasándole como parámetros el color de la luz y la intensidad de la misma.

Una vez tenemos una escena completamente visible, se procede a crear y añadir la rejilla. La función existente en la biblioteca `Three.js` `new THREE.GridHelper()` permite definirla muy fácilmente introduciendo por parámetro a la función la separación, cantidad

y color de las líneas de división que generan la rejilla. Para añadirla a la escena, basta con usar `.add(rejilla)`. Estos elementos forman la escena base del visor.

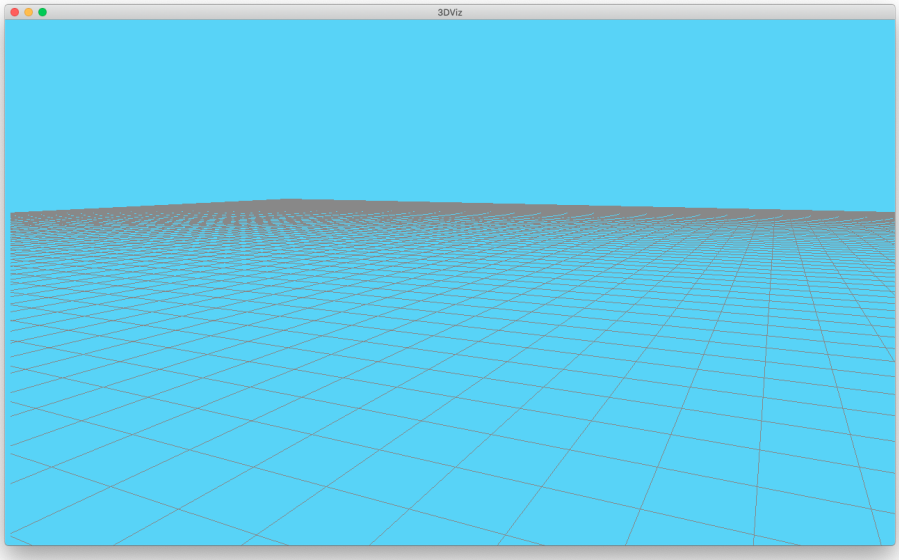


Figura 5.3: Interfaz gráfica con la escena base del visor

5.3.2. Visualización de puntos

Un punto es el elemento geométrico más simple que es posible representar. Pese a ello proporciona al visor la capacidad de realizar representaciones complejas mediante el uso de una gran cantidad de puntos. Para representar puntos, únicamente es necesario una posición en el eje de coordenadas, el tamaño del punto y el material o color que se quiera darle. La biblioteca `Three.js` proporciona una serie de elementos para facilitar la creación de puntos. El código se muestra en el cuadro 5.2.

```
function addPoint (point){
    var geometry = new THREE.Geometry();
    geometry.vertices.push( new THREE.Vector3(point.x,point.z,point.y));
    var material = new THREE.PointsMaterial( { size: 8, sizeAttenuation: false,
                                    alphaTest: 0.5, transparent: true } );
    material.color.setRGB( point.r, point.g, point.b);
    var particles = new THREE.Points( geometry, material );
    particles.name ="points";
```

```
    scene.add( particles );}
```

Cuadro 5.1: Creación y visualización de puntos

Primero es necesario indicar que se está creando una figura geométrica para posteriormente definirla. Al tratarse de un punto, solo va a tener un vértice por lo que al definir la geometría se indica que serán vertices pero únicamente se proporciona uno de ellos mediante un vector "x", "y" y "z", que serán las coordenadas centrales del punto a representar. Dado que el sistema de coordenadas de obtención en WebGL de los datos no se corresponde con el sistema de coordenadas gráficas de la escena, hay que realizar una conversión. Si recibimos un punto con coordenadas (x,y,z), en la escena corresponderán a (x,z,y).

Una vez definida la geometría lo siguiente es definir el tamaño del punto, su aspecto y su color. El tamaño indicado en el código es de 8 píxeles, sin embargo este tamaño es configurable a través del fichero de configuración. El color del punto se define mediante sus componentes RGB, que vienen indicadas por el servidor que transmite el punto. Las componentes vendrán separadas en "R", "G" y "B", y su valor oscilará entre "0" y "1" (se corresponde con el valor "255") para cada una de ellas.

Finalmente creamos el punto a partir de la geometría y el material proporcionado, le atribuimos el nombre "punto" para poder borrarlo discrecionalmente si así se desea, y se añade a la escena. Esta función será invocada cada vez que se reciba uno o más puntos procedentes del servidor. La figura 5.4 muestra varios puntos coloreados en el visor.

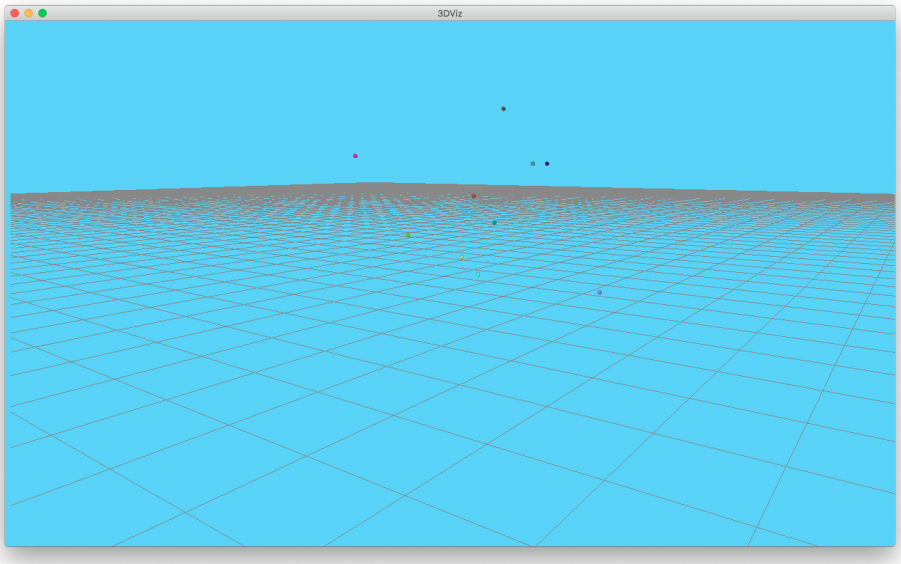


Figura 5.4: Varios puntos de distintos colores visualizados en el visor

5.3.3. Visualización de segmentos

El segmento es otro de los elementos fundamentales de la geometría y se puede definir cómo un fragmento de recta que está comprendido entre dos puntos. Teniendo en cuenta esto, para representarlo únicamente es necesario las coordenadas de dos puntos, el grosor del segmento y el color del mismo. En el cuadro 5.3 se muestra el código para la creación de un segmento.

```
function addLine(segment){
    var geometry = new THREE.Geometry();
    geometry.vertices.push(
        new THREE.Vector3(segment.fromPoint.x, segment.fromPoint.z,
                          segment.fromPoint.y),
        new THREE.Vector3(segment.toPoint.x, segment.toPoint.z,
                          segment.toPoint.y));
    var material = new THREE.LineBasicMaterial();
    material.color.setRGB(segment.r, segment.g, segment.b);
    material.linewidth = 2;
    line = new THREE.Line(geometry,material);
    line.name = "line";
```

```
    scene.add(line);}
```

Cuadro 5.2: Creación y visualización de segmentos

La estructura es muy similar que la función para crear un punto y al igual que allí, se necesita definir la geometría. En este caso vamos a tener dos vértices en lugar de uno (dos puntos). El primero es desde el lugar donde comience el segmento, y el segundo el lugar donde termina. En otras palabras, el segmento será la unión entre el primer vértice con el segundo. Las coordenadas de estos dos puntos serán proporcionadas por el servidor. Al igual que en el caso del punto, el sistema de coordenadas del servidor no coincide con el sistema del visor, por lo que se debe realizar la misma conversión, es decir la coordenada "z" que envíe el servidor, se corresponde a la coordenada "y" del visor, y viceversa.

Posteriormente se definen el material y aspecto del segmento proporcionando el grosor y el color. En el código anterior, el grosor es de 2 píxeles, sin embargo este parámetro es configurable mediante el fichero de configuración. El color, por el contrario, viene definido por la aplicación mediante sus componentes RGB, que al igual que para el caso del punto, vienen repartidas en "R", "G" y "B", valiendo entre "0" y "1" (corresponde al valor "255").

Finalmente se crea el segmento a partir de la geometría y el material proporcionado, dándole el nombre "line" para poder borrar selectivamente solo los segmentos. Una vez generado el elemento, se añade a la escena. Esta función será la invocada cada vez que se reciba uno o más segmentos. La figura 5.5 muestra varios segmentos en el visor.

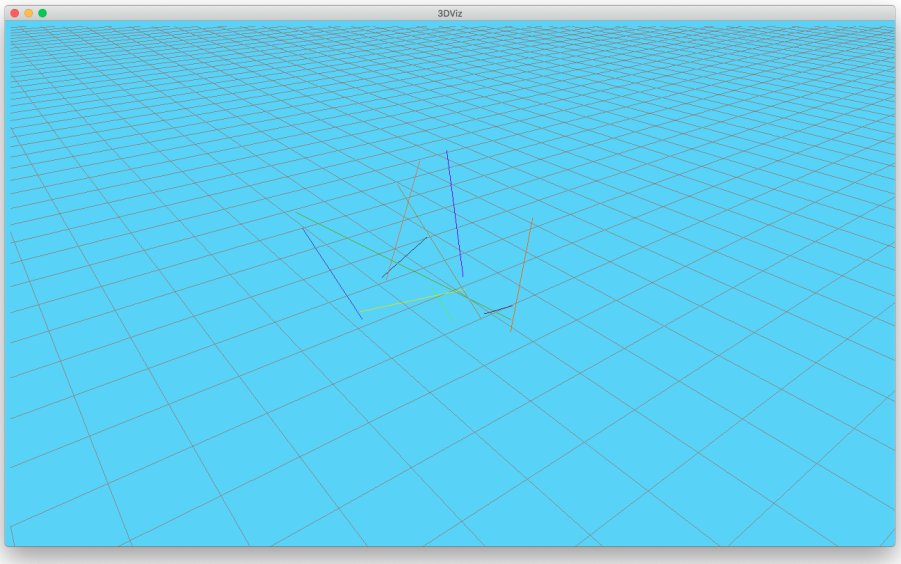


Figura 5.5: Varios segmentos de distintos colores visualizados en el visor

5.3.4. Visualización de objetos 3D

Un objeto 3D es una representación matemática de un objeto tridimensional cuya geometría se puede guardar en un archivo. El visor es capaz de representar dos formatos diferentes de estos archivos, “obj” y “dae”, y moverlos cuando la aplicación lo requiera.

5.3.4.1. Creación y visualización de los objetos 3D

Los objetos 3D pueden ser recibidos bien a través del archivo completo como texto plano, o bien a través de una URL. El visor será capaz de representar el objeto 3D y ubicarlo en la posición indicada. El código del cuadro 5.4 muestra cómo identificar si el objeto viene como texto plano o URL, para invocar a la función correspondiente dependiendo el formato del objeto.

```
function addObj(obj,pos){
    var type = obj.obj.split(":");
    if (type[0] == "https" || type[0] == "http") {
        var url = obj.obj
    } else{
        var file = new Blob([obj.obj], {type:'text/plain'});
```

```
        var url = window.URL.createObjectURL(file);
    }
    if (obj.format == "obj"){
        loadObj(url, obj,pos)
    } else if (obj.format == "dae") {
        loadDae(url,obj.pos);
    }
}
```

Cuadro 5.3: Creación y visualización de objetos 3D

Cuando se recibe el objeto procedente de la plicación, lo primero que se hace es trocear el mensaje para analizarlo y esclarecer si lo que se está enviando es el archivo completo o una URL al archivo. De tratarse de una URL, debe contener "https://" o "http://", se puede trocear mediante ":" y, con una sentencia condicional verificar si la primera parte de la separación es "https", "http" u otra cosa. En caso de no ser "https" o "http", lo que lo que se ha recibido es el objeto completo y se debe generar un archivo virtual utilizando el objeto *Blob* (objeto que representa un fichero) y posteriormente generar una URL virtual, ya que se ha recibido como texto plano y para visualizarlo es necesario que el objeto tenga una URL local al mismo.

Una vez se tiene la URL del objeto, es necesario saber el formato del archivo del objeto ("obj" o "dae"), ya que no se carga el objeto de la misma forma. Identificar el formato es sencillo, ya que vendrá indicado por la aplicación y, dependiendo del formato, se invoca a la función de procesamiento correspondiente pasando por parámetro la URL y la posición deseada del objeto en el visor (coordenadas y orientación del objeto) recibidas del servidor.

- **Representación de objetos del tipo obj**

  Un archivo con formato "obj" es conocido como Wavefront 3D Object File y el formato fue desarrollado por Wavefront Technologies. Es usado para un objeto tridimensional que contiene las coordenadas 3D (líneas poligonales y puntos), mapas de textura, y otra información de objetos. La biblioteca Three.js proporciona un API para cargar y mostrar un objeto de este tipo a través de una URL.

  El código del cuadro 5.5 carga y muestra un objeto "obj":

```
function loadObj(url,obj,pose3d){
```

```
    var loader = new THREE.OBJLoader();
    loader.load(
        url,
        function(object){
            object.name = obj.id;
            id_list.push(obj.id);
            object.position.set(pose3d.x, pose3d.z, pose3d.y);
            object.rotation.set(pose3d.rx*toDegrees, pose3d.rz * toDegrees,
                                pose3d.ry * toDegrees);
            scene.add(object);
        },
        function (xhr){
        },
        function (error){
            console.log(error);
        });
}
```

Cuadro 5.4: Carga y visualización de objetos “obj”

Lo primero que hace es inicializar el API de Three.js para cargar un objeto “obj” y llama a la función del API encargada de cargar el objeto. Esta función tiene como parámetros la URL y otras tres funciones más. La primera es la encargada de cargar el objeto y mostrar el mismo en el visor, la segunda se ejecuta mientras se está cargando el objeto (por ejemplo, para mostrar una barra de progreso) y la última se ejecutará si ocurre algún error durante la carga. La primera función es la importante, en ella identificamos al objeto dándole un `id` previamente establecido y lo añadimos a un `array` para poder borrarlo o moverlo posteriormente. También posiciona el objeto en 3D mediante las coordenadas enviadas por la aplicación. Igualmente orienta el objeto 3D mediante la rotación correspondiente a cada eje, que es enviada en radianes y se debe convertir a grados. Finalmente, se añade el objeto al visor.

- **Representación de objetos del tipo dae**

  El formato “dae” de ficheros, también conocido como “Collada”, es un formato de archivos 3D utilizado para el intercambio activo entre programas de gráficos y

basado en el esquema XML Collada. La biblioteca Three.js proporciona un API para la carga y visualización de este formato a partir de una URL. El código del cuadro 5.6 carga y muestra un objeto "dae":

```
function loadDae (url,obj,pose3d){
    var loader = new THREE.ColladaLoader();
    loader.load(url,
        function (object) {
            object = object.scene;
            object.name = obj.id;
            id_list.push(obj.id);
            object.position.set(pose3d.x, pose3d.z, pose3d.y);
            object.rotation.set(pose3d.rx, pose3d.rz, pose3d.ry);
            scene.add( object );
        },
        function (xhr){
        },
        function (error){
            console.log(error);
        });
}
```

Cuadro 5.5: Carga y visualización de objetos "dae"

Como se puede apreciar, el código es igual que el utilizado para el formato "obj", únicamente cambian la inicialización del API que en este caso es el API para cargar un archivo "dae" y la obtención del objeto, ya que se debe referenciar al mismo para poder cargarlo en el visor. El resto del código es el mismo para ambos casos.

Estas funciones de carga serán invocadas cada vez que se reciba un objeto nuevo que mostrar. La figura 5.6 muestra dos objetos (cada uno de un formato) en el visor.

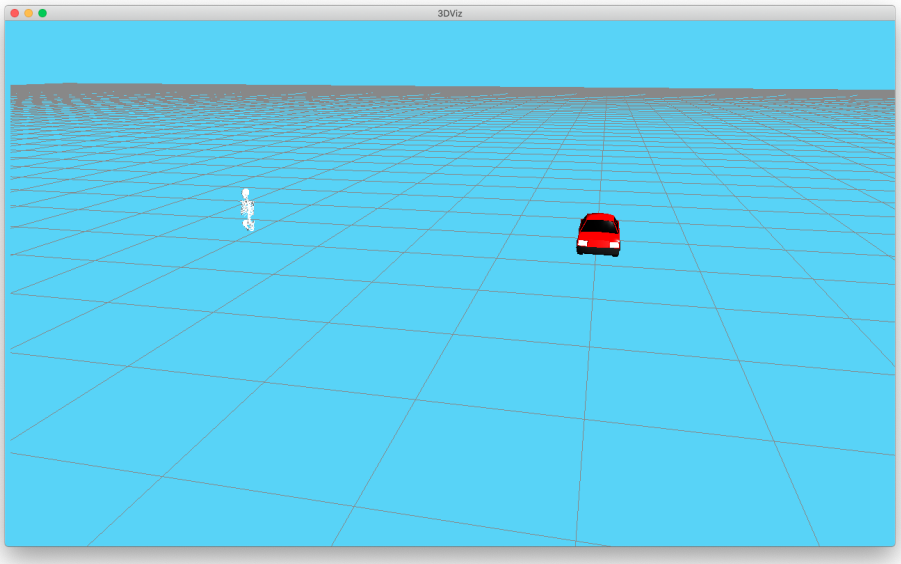


Figura 5.6: Dos objetos 3D mostrados en el visor

5.3.4.2. Movimiento de los objetos 3D

A diferencia de un punto o un segmento, en los que es sencillo eliminarlo y volver a crearlo en la nueva posición, un objeto 3D interesa reubicarlo a la nueva posición en lugar de eliminarlo y crearlo de nuevo. Recrearlo conllevaría tener que enviar una URL o un texto plano con el archivo del objeto y cargarlo de nuevo, teniendo un retardo y carga de trabajo importante para el visor.

Al haber identificado el objeto mediante un `id` único (en el caso del punto y la recta, el `id` es el mismo para cada elemento), 3DVizWeb permite realizar el movimiento.

```
function moveObj(objeto){
    selectedObject = scene.getObjectByName(objeto.id);
    selectedObject.position.set(objeto.x, objeto.z, objeto.y);
    selectedObject.rotation.set(objeto.rx, objeto.rz, objeto.ry);
}
```

Cuadro 5.6: Código para realizar el movimiento de los objetos 3D

Mover el objeto es mucho más rápido y sencillo que eliminarlo y crearlo de nuevo. Primero se selecciona el objeto mediante el identificador, posteriormente se reubica el

objeto seleccionado a las nuevas coordenadas y se aplica la nueva orientación. La figura 5.7 corresponde a los objetos de la figura 5.6 desplazados y rotados:

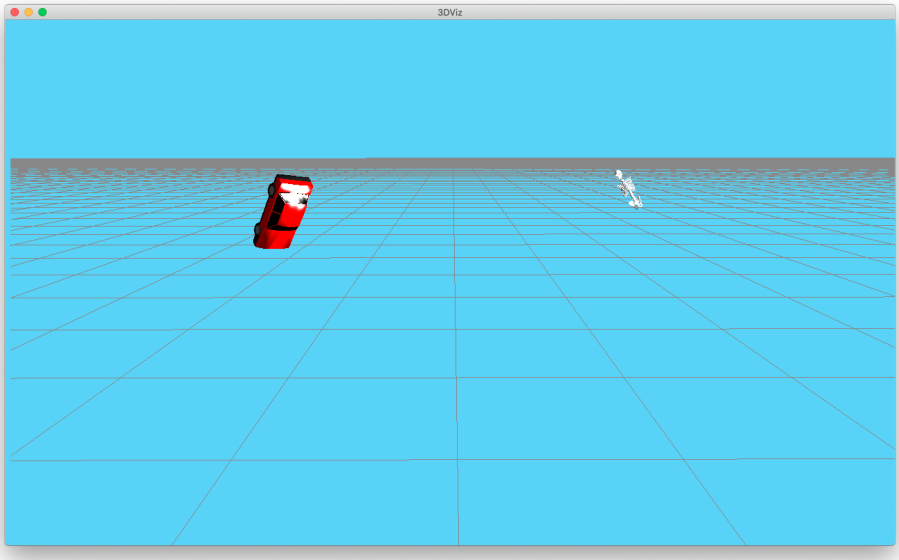


Figura 5.7: Objetos 3D reubicados y rotados

5.3.5. Borrado de elementos mostrados en el visor

El visor ofrece dos posibilidades de borrado, la primera es borrado total de lo que se mostraba previamente en el visor, la segunda posibilidad es el borrado parcial del visor, de modo que solo se borran los elementos indicados por la aplicación.

```
function deleteObj(id){
    if (id == ""){
        delete_list = id_list;
        id_list = ["points","line"];
        id_obj = [];
    } else if (id == "obj"){
        delete_list = id_obj;
        id_list = ["points","line"];
        id_obj = [];
    } else {
        delete_list = [id];
    }
```

```
    for (i = 0; i < id_list.length; i++){
        var selectedObject = scene.getObjectByName(delete_list[i]);
        while (selectedObject != null) {
            scene.remove(selectedObject);
            selectedObject = scene.getObjectByName(delete_list[i]);
        }
    }
}
```

Cuadro 5.7: Código para realizar el borrado de elementos mostrados en el visor

El código del cuadro 5.8 primero verifica cuál es el tipo de borrado, analizando la variable `id` que se pasa por parámetro. Si esta variable no tiene valor, el borrado es completo y la lista para borrar es la que contiene todos los identificadores de todos los elementos que se muestran en la escena, si toma el valor "obj" la lista a borrar es la correspondiente a la que contiene los identificadores de los objetos 3D. Por último, si toma el valor "line" o "points", la lista está formada únicamente por este identificador.

El borrado es selectivo por lo que para realizarlo previamente se debe haber identificado el elemento que se desea borrar y seleccionarlo en la escena mediante el identificador. Dado que la escena puede tener múltiples elementos con el mismo identificador (los segmentos y los puntos), es necesario realizar varias iteraciones buscando los elementos de la escena que el identificador coincida con el deseado hasta eliminar todos los elementos de la escena que lo tienen, ya que la llamada a `getObjectByName` solo devuelve un elemento, y no todos.

5.4. Estructura de los Mensajes

En esta sección se describe la estructura de los mensajes que intercambian la aplicación y 3DVizWeb para solicitar la visualización de cada tipo de primitiva 3D, ya sea punto, segmento u objeto. También cómo se definen utilizando el lenguaje de descripción Slice, explicado en el capítulo 3 de esta memoria.

5.4.1. Mensaje para visualizar los puntos

Para representar un punto únicamente se necesita una posición en el eje de coordenadas y el color que tendrá el punto. Teniendo en cuenta esto, la estructura del mensaje es muy simple:

- Coordenada "X"
- Coordenada "Y"
- Coordenada "Z"
- Componente de color "R" (valor decimal entre 0 y 1)
- Componente de color "G" (valor decimal entre 0 y 1)
- Componente de color "B" (valor decimal entre 0 y 1)

Esta estructura es definida mediante el lenguaje Slice de la siguiente forma en el archivo "primitives.ice", que contiene las definiciones de las estructuras intermedias creadas para definir estructuras más complejas y definitivas:

```
#ifndef PRIMITIVES_ICE
#define PRIMITIVES_ICE

module jderobot{
    struct RGBPoint{
            float x;
            float y;
            float z;
            float r;
            float g;
            float b;
    };
};
#endif
```

Cuadro 5.8: Definición de la estrucutra del punto con Slice

El visor tiene la iniciativa de las comunicaciones ICE y solicitará si hay nuevos puntos que mostrar cada cierto periodo de tiempo. Dado que puede interesar que este periodo sea largo para evitar constantes peticiones y mayor carga de trabajo, se ha decidido que no solo se pueda enviar un punto en cada petición, sino que se pueda enviar un buffer de puntos. La estructura del mensaje, por tanto, no es una única posición en el eje de coordenadas y el color, sino una colección variable de estos elementos (podrá ser uno o más).

Para dar la posibilidad al servidor de indicar si desea añadir o eliminar lo que está pintado en ese momento en el visor (refrescar la escena que muestra el visor), al mensaje se le añade un parámetro. Éste toma el valor "all", si se desea eliminar todo lo que se muestra en ese momento y únicamente que se visualice lo que se transmite en ese mensaje. El valor "part", si lo que se desea es eliminar únicamente los elementos de este tipo y añadir los que se transmite en este mensaje al resto de elementos mostrados en la escena. O el valor "nothing", si únicamente se desea añadir lo recibido sin eliminar nada de lo que está pintado en la escena. Por tanto el mensaje queda como sigue:

- Buffer de puntos con los siguientes parámetros:
  - Coordenada "X"
  - Coordenada "Y"
  - Coordenada "Z"
  - Componente de color "R" (valor decimal entre 0 y 1)
  - Componente de color "G" (valor decimal entre 0 y 1)
  - Componente de color "B" (valor decimal entre 0 y 1)
- Refresco del visor

Esta estructura final del mensaje con el buffer de puntos y el refresco está definida en el archivo Slice "visualization.ice", utilizando las primitivas creadas en el archivo "primitives.ice":

```
#ifndef VISUALIZATION_ICE
#define VISUALIZATION_ICE

#include <primitives.ice>
```

```
module jderobot{

    sequence<RGBPoint> Points;

    struct bufferPoints{
        Points buffer;
        string refresh;
    };
};
#endif
```

Cuadro 5.9: Definición del buffer de puntos con Slice

5.4.2. Mensaje para visualizar segmentos

Para visualizar un segmento únicamente es necesario enviar la posición en el eje de coordenadas de dos puntos y el color con el que se desea visualizar el segmento. Por tanto, el mensaje que transporta los segmentos no es más que una ampliación del mensaje de los puntos. En el cuadro 5.11 se muestra como queda la definición del mensaje en el archivo "primitives.ice"

```
structu Point{
    float x;
        float y;
        float z;
};

struct Segment{
    Point fromPoint;
    Point toPoint;
};



struct RGBSegment{
    Segment seg;
```

```
    float r;
    float g;
    float b;
};
```

Cuadro 5.10: Definición de la estructura del segmento con Slice

Ahora es necesario definir el formato definitivo del mensaje, para ello se extenderá el archivo "visualization.ice" como se muestra en el cuadro 5.12.

```
sequence<RGBSegment> Segments;

struct bufferSegments{
    Segments buffer;
    string refresh;
};
```

Cuadro 5.11: Definición del buffer de segmentos con Slice

5.4.3. Mensaje para visualizar un objeto 3D

Mientras que para los segmentos y para los puntos únicamente se hace la petición sin añadir ningún parámetro a la misma, en el caso de las peticiones de un objeto 3D desde el visor 3D al servidor, se debe incluir el identificador que se le va a dar al objeto (en caso de que el servidor tenga uno para enviar), de modo que tanto visor como servidor pueden hacer corresponder las peticiones de movimiento o borrado con un objeto concreto mediante este identificador.

Para mostrar un objeto se necesita conocer el archivo que contiene el objeto 3D, el formato del archivo, la posición en el eje de coordenadas, la orientación del objeto y la escala del objeto. El archivo puede ser enviado de dos formas, la primera mediante una URL a un servidor externo o página web, la segunda enviando el fichero como texto plano. Esta segunda vía tiene el inconveniente de que no pueden ser enviados archivos de más de 1 MB de tamaño, por lo que si se quiere enviar un archivo más grande, debe realizarse mediante URL.

Se transmite también el formato del archivo, que como se ha explicado puede tomar los valores “obj” o “dae”, para que el visor conozca el formato del objeto que se le está trasmitiendo.

La escala del objeto es un número decimal que indica el tamaño con el que se desea mostrar el objeto, permitiendo así mostrar varias veces el mismo objeto pero cambiando de tamaño (por ejemplo, podemos querer mostrar un brazo humano en diferentes tamaños para representar a un niño y a un adulto).

Por último, tanto para la posición como para la orientación se usará la clase “Pose3D”. Esta clase está formada por las coordenadas “X”, “Y” y “Z”, la coordenada homogénea “h” y el cuaternión “q0”, “q1”, “q2” y “q3” para, usando las formulas matemáticas correspondientes, proporcionar la orientación del objeto.

El formato del mensaje tiene la siguiente estructura:

- Archivo con el objeto 3D
- Formato del archivo
- Pose3D
  - Coordenada “X”
  - Coordenada “Y”
  - Coordenada “Z”
  - Coordenada homogénea“h”
  - Cuaternión “q0”
  - Cuaternión “q1”
  - Cuaternión “q2”
  - Cuaternión “q3”
- Escala
- Identificador
- Refresco del visor

Como se puede ver, se vuelve a enviar el identificador que había sido previamente asignado y transmitido por el visor, y se añade también el refresco al igual que en los puntos y los segmentos. Sin embargo, debido a las limitaciones de tamaño de los archivos que se pueden enviar mediante ICE, y la necesidad de establecer un identificador común, solo es posible enviar un objeto 3D por petición realizada y no un buffer de objetos, como sí se hace con los puntos y los segmentos.

La clase "Pose3D" es definida en un nuevo archivo Slice llamado "pose3d.ice":

```
#ifndef POSE3D_ICE
#define POSE3D_ICE

module jderobot{

  class Pose3DData
  {
        float x;
        float y;
        float z;
        float h;
        float q0;
        float q1;
        float q2;
        float q3;
  };

};
#endif
```

Cuadro 5.12: Definición de la clase Pose3D con Slice

Este mensaje queda definido en el archivo Slice "visualization.ice", al tratarse de un formato final y no de una primitiva (se trata de una estructura de mensaje final y no una estructura intermedia para crear nuevas). Además, es necesario incluir al principio la referencia al archivo 'pose3d.ice":

```
#include <pose3d.ice>
    ....
```

```
struct object3d {
        string obj;
        string id;
        string format;
        float scale;
        Pose3DData pos;
        string refresh;
};
```

Cuadro 5.13: Definición de la estructura del objeto 3D con Slice

5.4.4. Mensaje para mover los objetos 3D

Los objetos 3D que se visualizan probablemente se desee moverlos de posición u orientación, por lo que ha sido necesario incorporar una secuencia de petición y envío de movimiento. Solo se comenzará a realizar la petición de movimiento una vez se muestre un objeto ya en el visor. La estructura de este mensaje es una versión reducida del mensaje para crear un objeto. En este caso únicamente se envía la nueva posición mediante la clase "Pose3D" y el identificador del objeto que se desea mover. Por tanto, la estructura queda de la siguiente manera:

- Pose3D
  - Coordenada "X"
  - Coordenada "Y"
  - Coordenada "Z"
  - Coordenada homogénea"h"
  - Cuaternión "q0"
  - Cuaternión "q1"
  - Cuaternión "q2"
  - Cuaternión "q3"
- Identificador

Este mensaje queda definido en el archivo "visualization.ice", al tratarse de un movimiento y no de un elemento nuevo:

```
struct PoseObj3D{
    Pose3DData pos;
    string id;
 };
```

Cuadro 5.14: Definición de la estructura del movimiento de los objetos 3D con Slice

Se puede enviar una secuencia de movimientos para uno o más objetos, por lo que, al no tener el impedimento del tamaño de los archivos o del identificador, se decide permitir el envío de buffers con estas secuencias de movimiento. Si entre petición y petición se ha movido por ejemplo un objeto real varias veces, que se puedan realizar esos movimientos al mismo tiempo en el visor, evitando largos retardos que provocarían enviar los movimientos de uno en uno. En esta ocasión el campo refresco no se envía. El *timing* para actualizar el movimiento está establecido en el fichero de configuración del visor. La estructura final del mensaje es la siguiente:

- Buffer de movimientos
  - Pose3D
    - Coordenada "X"
    - Coordenada "Y"
    - Coordenada "Z"
    - Coordenada homogénea "h"
    - Cuaternión "q0"
    - Cuaternión "q1"
    - Cuaternión "q2"
    - Cuaternión "q3"
  - Identificador

El mensaje final queda definido de la siguiente forma en el archivo "visualization.ice":

```
sequence<PoseObj3D> bufferPoseObj3D;
```

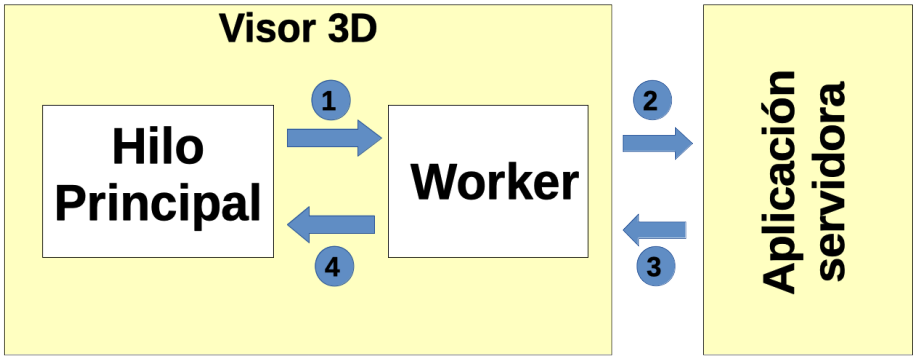
Cuadro 5.15: Definición del buffer de movimientos con Slice

5.5. Conexiones

En esta sección se describe cómo se realiza la conexión y la posterior recepción de cada uno de los objetos y objetos 3D anteriormente indicados. La conexión se realiza mediante el middleware ICE y su lenguaje de especificación Slice. Primero se detalla cómo se realiza la conexión con la aplicación que quiere usar el visor y la posterior petición de cada tipo de objeto. Finalmente se explica cómo se realiza la recepción de cada objeto y el tratamiento que recibe cada uno para su posterior visualización en el visor.

5.5.1. Conexión y peticiones a la aplicación

Para la conexión e intercambios con la aplicación, el visor hace uso de los Web Workers de HTML5[2] que permiten ejecutar código en segundo plano sin bloquear al proceso principal. Su uso es imprescindible para que el hilo principal no se bloquee y pueda mostrar los elementos que se reciban mientras espera la recepción del resto de peticiones realizadas. En la figura 5.8 se muestra el esquema del funcionamiento del Worker:



1.- El hilo principal indica al *Worker* que envíe los mensajes de solicitud a la aplicación servidora.
2.- El *Worker* envía los mensajes de solicitud.
3.- La aplicación servidora responde a esos mensajes con lo que desea mostrar.
4.- El *Worker* reenviá lo recibido al hilo principal para que lo procese.

Figura 5.8: Esquema de funcionamiento del Web Worker en el Visor 3D

[2] `https://www.w3schools.com/html/html5_webworkers.asp`

Una vez que el visor se ha lanzado y se ha terminado de cargar, se crea el Web Worker y se envía el mensaje para que se establezca la conexión. Esto se realiza utilizando el código del cuadro 5.16

```
w = new Worker("js/3DViz_worker.js");
w.postMessage({func:"Start",server:config.Server, port:config.Port});
```

Cuadro 5.16: Creación del Worker

Cuando se ha creado el Worker se inicializa la conexión ICE que devuelve un objeto `Ice.Communicator`, que es el objeto principal para establecer una comunicación ICE. Posteriormente se crea un objeto con la interfaz indicada anteriormente para realizar la conexión. También se inicializan las variables para posteriormente lanzar el *Promise* (permite manejar la naturaleza asíncrona de ICE) y la variable donde se guardará el *proxy* con la conexión realizada con el servidor. Esta secuencia de código se puede visualizar en el cuadro 5.17.

```
var ic = Ice.initialize();
var communicator;
var Promise;
var Prx = jderobot.VisualizationPrx;
var srv;
```

Cuadro 5.17: Código para la inicialización de la conexión ICE

Una vez incializada la conexión ICE, creado el objeto con la interfaz y recibido en el *Worker* el mensaje para que se establezca la conexión, se comienza el proceso para realizarla. Lo primero que se debe realizar es crear el *endpoint* mediante la IP y el puerto indicado en el fichero de configuración, después se crea un *proxy* realizando una petición al servidor mediante la llamada a `stringToProxy`, pasándole como parámetro una cadena de texto que contiene la identidad del objeto (será a partir de la cual el servidor sea capaz de identificar a qué *proxy* se está intentando conectar el cliente) y el *endpoint*. El *proxy* que devuelve es del tipo `Ice.ObjectPrx`, pero realmente lo que se necesita es un *proxy* a la interfaz creada en "visualization.ice", que se hará mediante una petición *Promise*, el objeto con la interfaz y el *proxy* que se acaba de recibir. Esta petición pregunta al servidor si el *proxy* que ha devuelto es un *proxy* para el objeto de la interfaz de "visualization.ice". Si lo es devuelve un *proxy* del tipo `jderobot.VisualizationPrx` que se guarda en la

variable creada para almacenar la conexión. Si no, devuelve un error. Finalmente, el *Worker* manda un mensaje al hilo principal indicando que la conexión se ha realizado correctamente. Todo esto se realiza mediante el método `connect` que se muestra en el cuadro 5.18

```
function connect(server,port){
  endpoint = "ws -h " + server + " -p " + port;
  var proxy = ic.stringToProxy("3DViz:" + endpoint);
  Promise = Prx.checkedCast(proxy).then(
      function(printer)
      {
          srv = printer;
          self.postMessage({func:"Connect"});
      });
}
```

Cuadro 5.18: Código para realizar la conexión ICE

Cuando el hilo principal recibe el mensaje indicando que la conexión ha sido exitosa establece las llamadas periódicas (utilizando el tiempo indicado en el fichero de configuración) a las funciones que se encargan de realizar las peticiones al servidor mediante el método de HTML `setInterval()`[3] como se muestra en código del cuadro 5.19. Si no lo ha sido elimina el *Worker* e imprime el mensaje por consola.

```
w.onmessage = function(event) {
        if (event.data.func == "Connect"){
       pointInterval = setInterval(function(){
                  setPoint();
                      }, config.updatePoints);
       lineInterval = setInterval(function(){
                  setLine();
                  },config.updateSegments);
       objInterval = setInterval(function(){
                  setObj();
                  },config.updateModels);}
       } else {
```

[3] https://www.w3schools.com/jsref/met_win_setinterval.asp

```
        console.log(event.data);
        w.terminate();
      }
```

Cuadro 5.19: Código para crear las llamadas periódicas a las funciones encargadas de realizar las peticiones al servidor

Cuando se pase el tiempo, se activan las funciones que envían el mensaje al *Worker* para que realice las peticiones al servidor. Estas peticiones serán para todos los objetos igual salvo en el caso de la petición de los objetos 3D, que como se ha explicado anteriormente, enviará el identificador que se dará al objeto en caso de que haya uno. Este identificador será una cadena de "obj" y un contador de objetos que hay en la escena, es decir, si no hay objetos en la escena, el siguiente tendrá de identificador "obj1", el siguiente "obj2" y así sucesivamente. Finalmente, esta función termina realizando una llamada a la función que se encarga de gestionar las respuestas del servidor. Esta secuencia de código se puede ver en el cuadro 5.20

```
function setPoint(){
     w.postMessage({func:"setPoint"});
     getData();
}
function setLine(){
    w.postMessage({func:"setLine"});
    getData();
}
function setObj(){
    id = "obj" + cont;
    w.postMessage({func:"setObj", id: id});
    getData();
}
```

Cuadro 5.20: Definición de las funciones de petición

En el *Worker*, cuando se reciban los mensajes que solicitan pintar cada tipo de elemento al servidor, se realiza la petición usando el proxy con la conexión y las funciones definidas en la interfaz Slice explicada anteriormente. Si a la petición se recibe respuesta, se reenvía al hilo principal para su manejo.

```
function setPoint(point){
    srv.getPoints().then(function(data){
        self.postMessage({func:"drawPoint",points: data});
    });
}

function setLine(){
    srv.getSegment().then(function(data){
            self.postMessage({func:"drawLine", segments: data});
    });
}

function setObj(id){
    srv.getObj3D(id).then(function(data){
            self.postMessage({func:"drawObj", obj: data});
    });
}
```

Cuadro 5.21: Código que realiza las peticiones al servidor

Como se puede ver en el cuadro 5.21, solo se están realizando las peticiones para los puntos, los segmentos y los objetos, pero no para los movimientos de los objetos, ya que solo se realiza una vez que se ha recibido un objeto para mostrar. Una vez que se tiene un objeto, se activa la llamada periódica a la función para que envíe el mensaje al *Worker* para que se realice la petición al servidor de la misma forma que las demás.

Las funciones para realizar las peticiones al servidor se han tenido que definir previamente mediante el lenguaje de descriptivo de ICE, Slice. Para realizar la definición se ha extendido el archivo "visualization.ice" explicado en la sección 5.4. A este archivo se le añade la descripción del cuadro 5.22.

```
interface Visualization
    {
          bufferSegments getSegment ();
          bufferPoints getPoints();
          object3d getObj3D(string id);
          bufferPoseObj3D getPoseObj3DData();
```

```
    };
};
```

Cuadro 5.22: Código añadido a las definiciones Slice creadas

5.5.2. Recepción y tratamiento de los mensajes recibidos

Ya se ha explicado cómo se conecta, cómo se envían los mensajes de petición al servidor y cómo se recibe la respuesta en el Worker que la transmite al hilo principal. Ahora se explicará cómo trata esos mensajes el hilo principal para mostrarlo en el visor.

En el hilo principal hay un manejador de mensajes que es la función `getData()`, se invoca cada vez que se realiza una petición. En esta función se analiza el mensaje que se recibe procedente del *Worker*. Se revisa cuál es el tipo y se realizan las tareas previas necesarias para posteriormente visualizar el elemento usando los métodos explicados en la sección 5.3. El condicional que se muestra en el cuadro 5.23 es el encargado de realizar este análisis.

```
function getData (){
    w.onmessage = function(event) {
        if (event.data.func == "drawPoint"){
            ...
        } else if (event.data.func == "drawLine"){
            ...
        } else if (event.data.func == "drawObj") {
            ...
        } else if (event.data.func == "pose3d") {
            ...
        }
    }
}
```

Cuadro 5.23: Manejador que analiza los mensajes recibidos

5.5.2.1. Tratamiento de los puntos

Si el mensaje enviado por el *Worker* es del tipo `drawPoint`, el manejador indicará que se deben ejecutar las sentencias correspondientes a la visualización de los puntos. Lo primero que se realiza es verificar si el servidor ha solicitado que haya refresco del visor total, parcial o que no haya refresco, si el servidor ha indicado que se debe refrescar el visor (y si el mensaje trae puntos que mostrar, ya que si no se interpretará el mensaje como erróneo), se llama al método encargado de eliminar todos o algunos de los elementos que se muestran en ese momento en el visor y que se explica en la sección 5.3.5.

Tanto si se ha refrescado como si no, se recorre el buffer de puntos mediante un bucle `for`, invocando al método encargado de mostrar un punto en el visor, en cada iteración hasta que ya no queden más puntos para mostrar. Al método se le pasa por parámetros el punto completo (coordenadas y componentes RGB), ya que se encarga de diferenciar cada uno y realizar las tareas necesarias para mostrarlos. En el cuadro 5.24 se muestra la secuencia de código que realiza lo descrito anteriormente.

```
if (event.data.func == "drawPoint"){
    if (event.data.points.buffer.length !=0){
        if (event.data.points.refresh == "all"){
            deleteObj("");
        } else if ((event.data.points.refresh == "part")) {
            deleteObj("points");
        }
    }
    points = event.data.points.buffer;
    for (var i = 0; i < points.length; i+=1) {
              addPoint(points[i]);
    }
}
```

Cuadro 5.24: Código para tratar los mensajes con los puntos

Dado que el tratamiento de los segmentos es muy similar al tratamiento de los puntos, no se profundiza en ello.

5.5.2.2. Tratamiento de los objetos 3D

El tratamiento de los objetos 3D comienza revisando si requiere refresco, o no, del visor. Una vez que se ha borrado, o no, el contenido del visor, se actualiza el contador que constituye el identificador que se ha indicado en las secciones anteriores. Posteriormente es necesario realizar la conversión de la ubicación y orientación enviada como "Pose3D" a un formato que el método encargado de mostrar los objetos sea capaz de entender.

Para realizar esta conversión, lo primero es crear una clase de JavaScript cuya estructura contendrá los parámetros que se necesitan para mostrar el objeto, es decir la posición en el eje de coordenadas (coordenadas "x", "y" y "z"), las orientaciones en cada eje de coordenadas (orientación "rx", "ry" y "rz") y el identificador del objeto 3D. La creación de esta clase se muestra en el cuadro 5.25.

```
class obj3DPose {
    constructor(id, x, y, z, rx, ry, rz){
        this.id = id;
        this.x = x;
        this.y = y;
        this.z = z;
        this.rx = rx;
        this.ry = ry;
        this.rz = rz;
    }
}
```

Cuadro 5.25: Definición de la clase que proporciona la posición de un objeto 3D

Para realizar la conversión se han creado tres funciones diferentes: `getYaw`, `getRoll` y `getPithc`. Estas funciones devuelven la orientación en cada uno de los ejes de coordenadas, usando las formulas matemáticas que transforman los cuaterniones en los ángulos usados para describir la orientación de un objeto tridimensional y que son un tipo de ángulos de Euler[4]. Este sistema es muy utilizado en navegación de los aviones y drones, y están formados por la dirección ("yaw"), elevación ("pitch") y ángulo de alabeo ("roll") correspondiendo respectivamente a la orientación en el eje "z", el eje "y" y el eje "z".

Finalmente se crea una función que devuelve un objeto de la nueva clase creada y que

[4] `https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles`

tiene ya la conversión (cuadro 5.26). A esta función se le pasa por parámetro el objeto completo recibido y, mediante la llamada a las funciones para realizar la conversión, crea el objeto de la nueva clase que contendrá el identificador, las coordenadas "x", "y" y "z", sin ningún tipo de conversión, y las orientaciones en cada eje convertidas y que están en radianes.

```
function getPose3D(data){
    var rotateZ=getYaw(data.pos.q0, data.pos.q1, data.pos.q2, data.pos.q3);
    var rotateY=getPitch(data.pos.q0, data.pos.q1, data.pos.q2, data.pos.q3);
    var rotateX=getRoll(data.pos.q0, data.pos.q1, data.pos.q2, data.pos.q3);
    var objpose3d = new obj3DPose(data.id, data.pos.x, data.pos.y,
                        data.pos.z, rotateX, rotateY, rotateZ);
    return objpose3d;
}
```

Cuadro 5.26: Función que devuelve la posición tras la conversión

Una vez que ya hemos realizado la conversión se llama al método encargado de cargar los objetos en el visor y que se explicó en la sección 5.3.4. A este método, se le pasa por parámetro el objeto completo recibido del servidor y el objeto de la nueva clase que incorpora la conversión de la orientación, para que pueda mostrar el objeto en el visor. Tras ello, se arranca la petición periódica de movimientos para el objeto (si no estaba arrancada previamente) al tener ya mínimo un objeto para mover. Esta secuencia de código se puede ver en el cuadro 5.27.

```
else if (event.data.func == "drawObj") {
    if (event.data.obj.obj !=""){
        if (event.data.obj.refresh == "all"){
            deleteObj("");
        } else if ((event.data.obj.refresh == "part")) {
            deleteObj("obj");
        }
    }
    cont += 1
    var pos = getPose3D(event.data.obj);
    addObj(event.data.obj,pos);
    if (posInterval == null){
```

```
        posInterval = setInterval(function(){
            setPose3D();
        }, config.updatePose3D);
    }
}
```

Cuadro 5.27: Código para tratar los mensajes con el objeto 3D

5.5.2.3. Tratamiento del movimiento de los objetos

En el caso de los mensajes con el movimiento únicamente se recorrerá buffer que contiene todos los movimientos realizando la conversión de Pose3D al formato admitido e invocando al método que materializa el movimiento de los objetos.

En la clase que se ha creado para la conversión se incluye el identificador del objeto que se va a mover, ya que, en este caso solo pasamos por parámetro la nueva clase por lo que es necesario incluir el identificador.

```
else if (event.data.func == "pose3d") {
    for (var i = 0; i < event.data.bpose3d.length; i += 1){
        data = event.data.bpose3d[i];
        var objpose3d = getPose3D(data);
        moveObj(objpose3d);}
    }
}
```

Cuadro 5.28: Código para tratar los mensajes con los movimientos de los objetos 3D

5.6. Experimentos

El visor está preparado para ser usado principalmente con Electron, ya que la lectura del fichero de configuración solo está habilitada si se usa con Electron. Se puede utilizar en el navegador web, pero la configuración será la que se indica al visor por defecto, no pudiendo modificarse, lo que limita en gran manera las funciones del mismo. La ejecución con Electron se realiza de la misma manera que se explicó en la sección 3.8.

Para ejecutar 3DVizWeb y realizar pruebas, se ha creado una aplicación de test en python que envía un coche rotado como objeto 3D, segmentos que forman una carretera y puntos que crean una barrera limitando los bordes de la carretera .

1. En un terminal ejecutar el servidor de prueba:

```
#>python server_test.py
```

Cuadro 5.29: Ejecutar servidor de prueba

2. En otro terminal, ejecutar 3DVizWeb

   - Como aplicación web utilizando Node.js[5]

```
#>node run.js
Se arranca el navegador y se introduce la URL http://localhost:7777/
```

Cuadro 5.30: Ejecución con Node.js

   - Como aplicación de escritorio con Electron[6]

```
#>npm install
#>npm start
```

Cuadro 5.31: Ejecución con Electron

A continuación se muestra la prueba realizada con el servidor, tomando varias secuencias del mismo experimento:

[5] https://www.youtube.com/watch?v=1fMw6iklIUs
[6] https://www.youtube.com/watch?v=-lKj8eAktMI

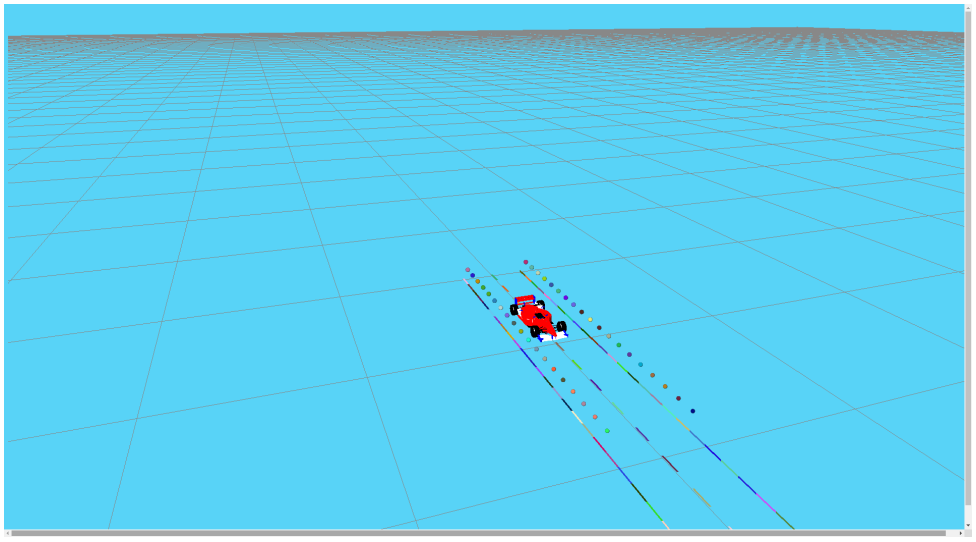


Figura 5.9: Test realizado en Electron

Capítulo 6

Servidor de imágenes con tecnologías web

En este capítulo se expone la creación de un nuevo driver para la plataforma JdeRobot. Este driver es un servidor de imágenes, al que se ha llamado CamServerWeb[1], obtenidas por una webcam (ya sea externa o interna del ordenador) de modo que cualquier aplicación externa pueda obtener las imágenes obtenidas.

6.1. Diseño

Este driver está diseñado con JavaScript y HTML como lenguajes de programación, WebRTC como biblioteca de captura de imágenes y ROS como middleware para la interconexión con los diferentes clientes. El driver puede ser ejecutado como una aplicación de escritorio usando el *framework* Electron o como una página web usando un navegador. Gracias a esto se permite la ejecución multiplataforma sin necesidad de realizar modificaciones, lo que provee a este driver de una gran ventaja respecto a los tradicionales que se tenían, que están desarrollados para una única plataforma. En la figura 6.1 se muestra el esquema de funcionamiento del driver.

[1] `https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez/tree/master/camserverWeb`

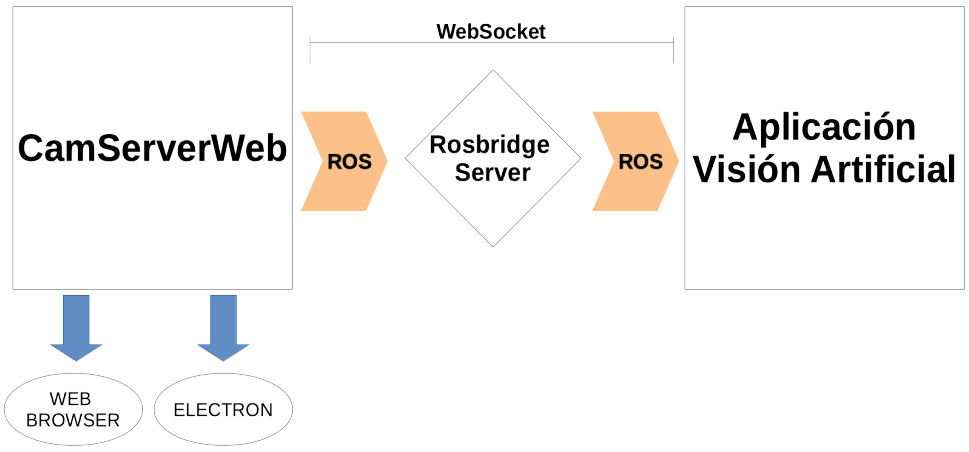


Figura 6.1: Diseño del driver CamServerWeb

Se puede apreciar en la figura 6.1 que entre el driver y la aplicación de visión artificial hay un servidor intermedio. Este servidor proporciona una capa de transporte WebSocket para realizar la conexión. En este caso permite la comunicación entre el servidor de imágenes y la aplicación de visión artificial que desea obtener las imágenes.

El driver cuenta con dos partes bien diferenciadas. La primera parte corresponde a la adquisición de las imágenes usando WebRTC para realizar la conexión con la fuente de video, y la segunda parte es la encargada de realizar la conexión mediante ROS y el posterior envío de los mensajes mediante los *Publisher* de ROS para que sean recibidos por la aplicación de visión artificial. Gracias al uso de ROS se posibilita que la aplicación de visión artificial esté desarrollada en cualquier lenguaje de programación y ejecutada en cualquier dispositivo ubicado en cualquier lugar del mundo. Finalmente, el driver cuenta con una interfaz gráfica de apoyo para realizar la configuración en tiempo de ejecución. Esta estructura se puede observar en la figura 6.2.

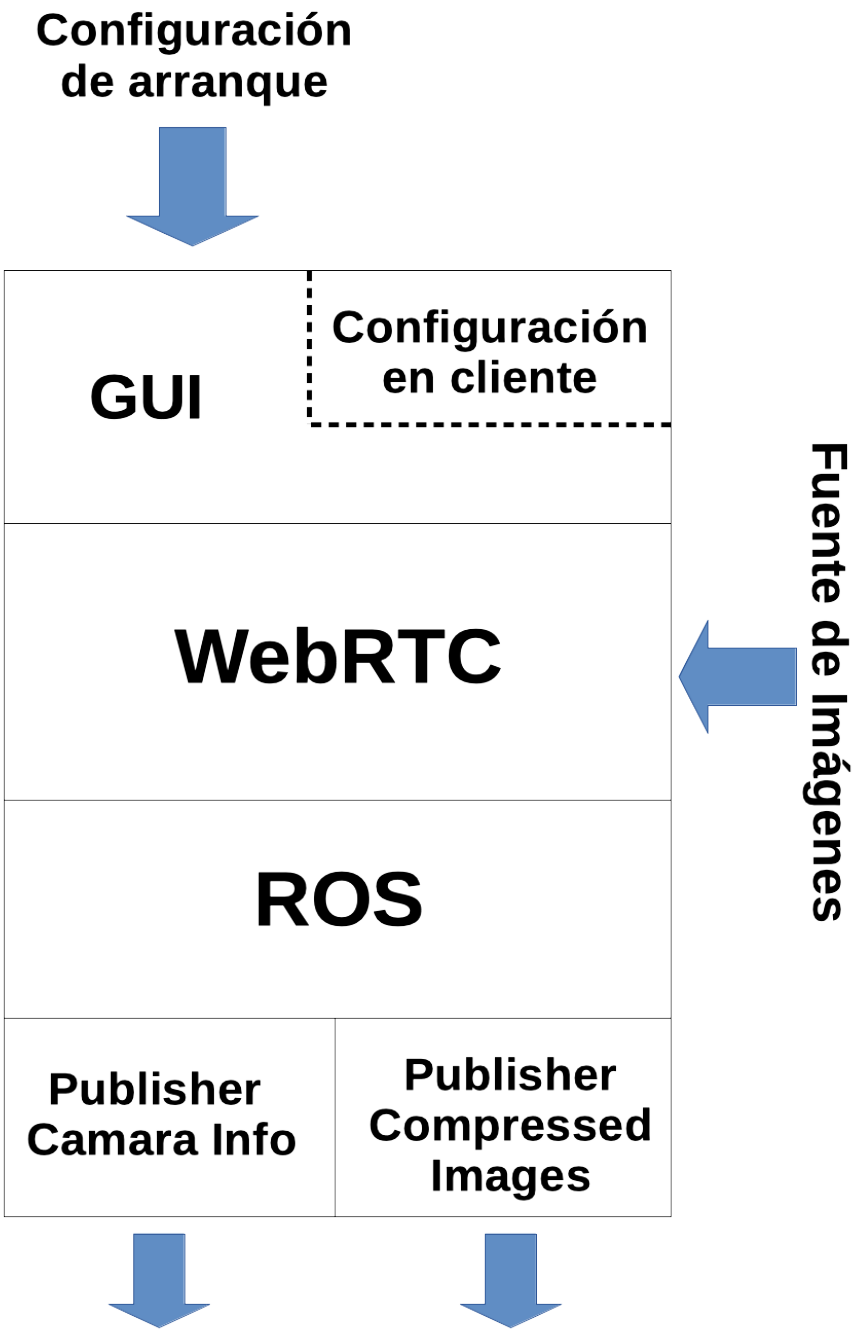


Figura 6.2: Estructura interna de CamServerWeb

6.2. Adquisición y preparación de imágenes

Para la obtención de las imágenes se hace uso del proyecto de código abierto WebRTC, que permite la captura y transmisión en tiempo real de audio, video y datos. Mediante

WebRTC se obtienen las imágenes de una cámara utilizando muy pocas líneas de código, lo que facilita enormemente el trabajo. El único problema que se ha tenido que solucionar, es el formato en el que se obtienen esas imágenes que es incompatible con ROS, por lo que se han tenido que llevar a cabo modificaciones en el formato de la imagen.

Lo primero que el driver debe realizar es recopilar todos los dispositivos conectados al ordenador y separar los dispositivos de video, que son los que realmente interesan. Para lograrlo, se utiliza el api `Navigator` de WebRTC, proporcionándo el objeto `navigator.mediaDevices`, al cual si le añadimos `.enumerateDevices().then(function (devices){})`, obtenemos todos los dispositivos multimedia conectados al ordenador donde se está ejecutando. Una vez que se tienen todos los dispositivos, simplemente interesan capturadores de video, es decir, serán aquellos cuyo tipo es de entrada de video (en el código sería un condicional `if (devices.kind == "videoinput"){}`). La lista de dispositivos se muestra en el menú de configuración, mediante un campo desplegable, como se muestra en la figura 6.3.

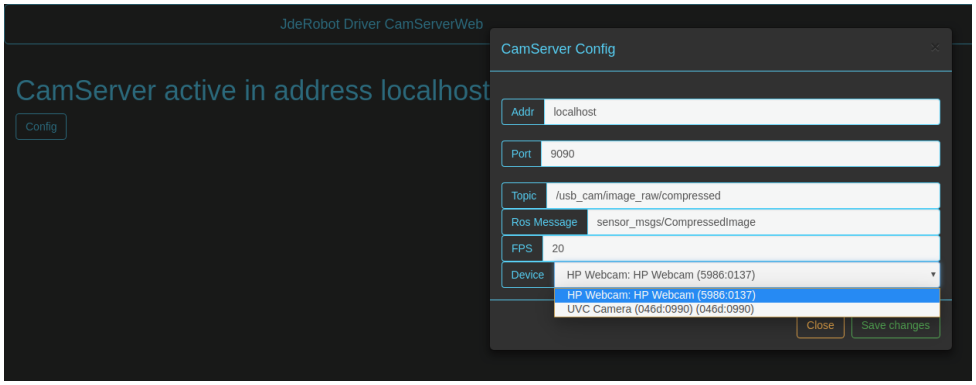


Figura 6.3: Selector del dispositivo de entrada de video

Una vez que se selecciona el dispositivo que vamos a utilizar para adquirir las imágenes, hay que conectarse a él. Para esta conexión se vuelve a utilizar la API de WebRTC, `Navigator` y el objeto `Navigator.mediaDevices`, sin embargo en esta ocasión utilizamos el método `navigator.mediaDevices.getUserMedia(constraints).then(function(stream){})`, donde `constraints` define los dispositivos multimedia (en este caso el dispositivo de entrada de video escogido) y `stream` es el flujo de datos obtenido de ellos.

Este flujo de datos está en el formato `MediaStream`, el cual no es apto para ser enviado o visualizado, por tanto es necesario realizar una conversión. Para realizar la conversión,

se utiliza el elemento de HTML `Canvas` y el método JavaScript asociado a este elemento `toDataURL()`. Este método devuelve un `data URI` (URLs prefijados que permiten a los creadores de contenido incorporar pequeños archivos en línea en los documentos) que contiene una representación de la imagen en el formato especificado por el parámetro `type`, tomando en nuestro caso el valor `image/jpeg`, para obtener las imágenes en el formato comprimido jpeg. Todo está contenido en un canvas virtual, ya que no se mostrará en ningún lugar y únicamente se utiliza como pasarela entre el API de WebRTC y el envío de las imágenes.

6.3. Conexiones

En esta sección se explica cómo se realiza la conexión tipo ROS con la aplicación de visión artificial y el posterior envío de imágenes mediante un *Publisher* de ROS.

6.3.1. Establecimiento de la conexión

Para realizar la conexión es necesario el uso de la biblioteca `roslibjs` y que se ha explicado en capítulos anteriores. Esta biblioteca proporciona todo el código necesario para realizar la conexión, indicando si se ha realizado correctamente o a si ha ocurrido algún error. Para realizar la conexión, la biblioteca `roslibjs` proporciona el objeto `ROSLIB.Ros`, y el método proporcionado por este objeto, `ros.on`. El código para establecer la conexión se muestra en el cuadro 6.1.

```
ros = new ROSLIB.Ros();
ros = new ROSLIB.Ros({
          url : "ws://IP:Puerto"
 });
```

Cuadro 6.1: Establecer conexión con ROS

En este código se indica que la conexión se hace utilizando un canal de comunicación WebSocket, y la IP y puerto por la que se transmite. De esta forma ya se habrá establecido la conexión ROS, pero aún hay que definir el tipo de mensaje a enviar y a través de qué etiqueta de ROS (*topic* de ROS) pueden conectarse las diferentes aplicaciones de visión artificial que deseen obtener las imágenes que se transmiten.

6.3.2. Estructura de los mensajes

El tipo de mensaje que vamos a utilizar para el envío es el tipo predefinido en el API de ROS, `sensor_msgs/CompressedImage`. El motivo de esta elección es que las imágenes se obtienen en formato comprimido jpeg, tal y como se ha explicado en la sección anterior. Sin embargo, este tipo de mensaje no envía información acerca del tamaño de la imagen (altura y anchura), por lo que es necesario enviar otro mensaje adicional para completar esta información. Este mensaje de apoyo es de tipo `sensor_msgs/CameraInfo` y transmitirá toda la información sobre la cámara (altura, anchura, etc).

Para definir estos dos mensajes se utiliza el objeto proporcionado por la biblioteca `roslibjs`, `ROSLIB.Topic`. A este objeto se le deben introducir como parámetros el objetos `ROSLIB.Ros` generado para realizar la conexión, el nombre del *topic* y el tipo de mensaje, por lo que se genera el *Publisher* para servir las imágenes y la información de la cámara mediante el código del cuadro 6.2.

```
var imagenTopic = new ROSLIB.Topic({
    ros:ros,
    name: config.Topic,
    messageType : "sensor_msgs/CompressedImage Message"})

var cameraInfo = new ROSLIB.Topic({
        ros: self.ros,
        name : "/usb_cam/camera_info",
        messageType: "sensor_msgs/CameraInfo"
      })
```

Cuadro 6.2: Estructura de los mensajes

6.3.3. Creación de los mensajes y publicación

Definidos los tipos de mensajes y establecida la conexión, el siguiente paso es transmitir los mensajes. Para ello se deben crear los mensajes con la información que se desea transmitir con el *Publisher*, utilizando de nuevo un objeto definido en la biblioteca `roslibjs`, `new ROSLIB.Message`. Para crear este objeto se debe pasar por parámetros el contenido que se quiere que tenga el mensaje, siempre cumpliendo las especificaciones

definidas para cada tipo de mensaje que están definidas en la documentación de ROS[2]. En este caso, para definir los dos mensajes que se transmiten se hara mediante el código del cuadro 6.3.

```
var videomensaje = new ROSLIB.Message({
   format : "jpeg",
   data : data.replace("", "")
   })
```

Cuadro 6.3: Definición del mensaje para las imágenes

Del código del cuadro 6.3 cabe destacar que `data` corresponde a los datos obtenidos mediante el método `toDataURL` indicado en la sección 6.2, remplazando la cabecera donde se indica el formato, ya que se indica en el propio mensaje mediante `format`.

Para el mensaje donde se transmite la información de la cámara se usa el código del cuadro 6.4.

```
var camarainfo = new ROSLIB.Message({
   height: imagen.height,
   width: imagen.width})
```

Cuadro 6.4: Definición del mensaje para la información de la cámara

Una vez creados los mensajes, solo falta publicarlos, lo que se consigue de una manera sencilla mediante el código que se muestra en el cuadro 6.5.

```
imageTopic.publish(imageMessage);
cameraInfo.publish(infoMessage);
```

Se puede apreciar fácilmente que lo que se está realizando es publicar los dos mensajes creados mediante la estructura definida anteriormente. Finalmente, es necesario enviar de una manera periódica estos mensajes, ya que con lo indicado anteriormente, únicamente se está enviando un mensaje de cada categoría. Para crear un envío periódico, se hace uso del método de JavaScript `SetInterval()`. Este método llama a una función o evalúa una expresión cuando pase un intervalo de tiempo (en milisegundos), que se indica en la llamada al método junto a la función que se quiere ejecutar cuando se cumpla el citado intervalo.

[2] `http://wiki.ros.org/common_msgs`

6.4. Configuración

Para configurar el driver se ofrecen dos posibilidades:

- Mediante el uso de un fichero con formato "YAML", al igual que en resto de herramientas de este trabajo.
- Mediante el menú de configuración incorporado en la interfaz gráfica.

El primer método realiza la configuración inicial del driver, ya que se configura al arrancar el driver. El segundo método permite realizar la configuración del driver durante la ejecución del mismo, ofreciendo la posibilidad de reconfigurar en tiempo de ejecución sin que sea necesario cerrar y volver a lanzar el driver.

Los parámetros configurables son los siguientes:

- Dirección IP. Por defecto es localhost.
- Puerto. Por defecto será 9090.
- *Topic* al que se registran los clientes. Por defecto es `/usb_cam/image_raw/compressed`
- Formato del mensaje. Por defecto es `sensor_msgs/CompressedImage`
- *Framerate*. Por defecto son 20 fotogramas por segundo.
- Fuente de video. Por defecto es la primera fuente detectada por WebRTC. Este parámetro únicamente puede ser configurado mediante la configuración en tiempo de ejecución.

6.4.1. Interfaz gráfica

La interfaz está realizada mediante HTML y Bootstrap, siguiendo el modelo del resto de aplicaciones web de la plataforma JdeRobot.

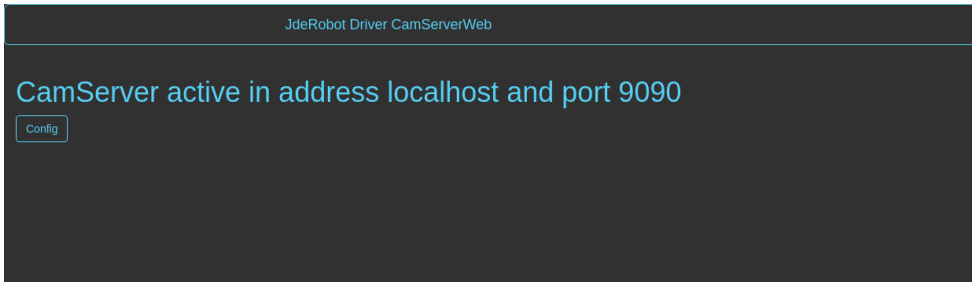


Figura 6.4: Interfaz gráfica del driver

La configuración se realiza gracias a un menú desplegable mediante la pulsación de un botón y una vez que se pulse el botón guardar, se almacena la nueva configuración para realizar las conexiones. Este menú se puede apreciar en la figura 6.5.

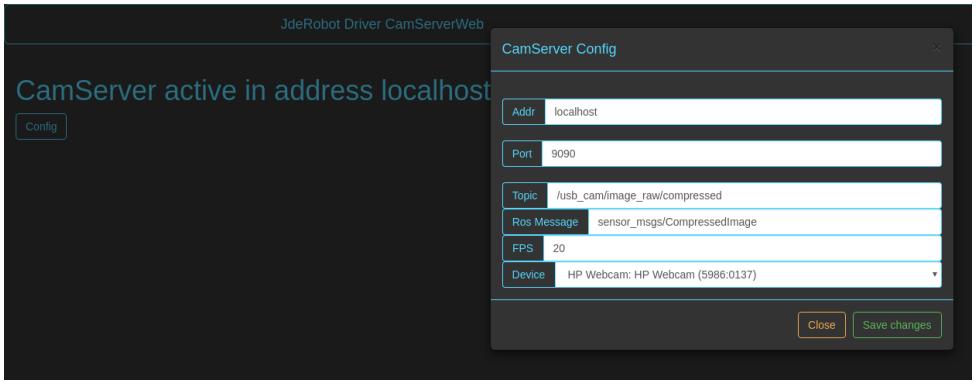


Figura 6.5: Menú de configuración del servidor de imágenes

6.5. Experimentos

Como se ha visto anteriormente, el driver puede ejecutarse a través de dos vías: Electron y Node.js. La preparación para que funcione correctamente es la misma para ambas vías, siendo la única diferencia la forma de ejecutar el driver.

En un primer terminal o consola se debe ejecutar el servidor intermedio de ROS.

```
#>roslaunch rosbridge_server rosbridge_websocket.launch
```

Cuadro 6.5: Ejecución del servidor intermedio

En un segundo terminal se ejecutará el driver.

- Como aplicación web utilizando Node.js[3]

```
#>node run.js
Se arranca el navegador y se introduce la URL http://localhost:7777/
```

Cuadro 6.6: Ejecución con Node.js

- Como aplicación de escritorio con Electron[4]

```
#>npm install
#>npm start
```

Cuadro 6.7: Ejecución con Electron

En ambos casos debemos configurar el driver para que se conecte con el servidor intermedio, en la imagen 4.5 muestra una posible configuración del driver.

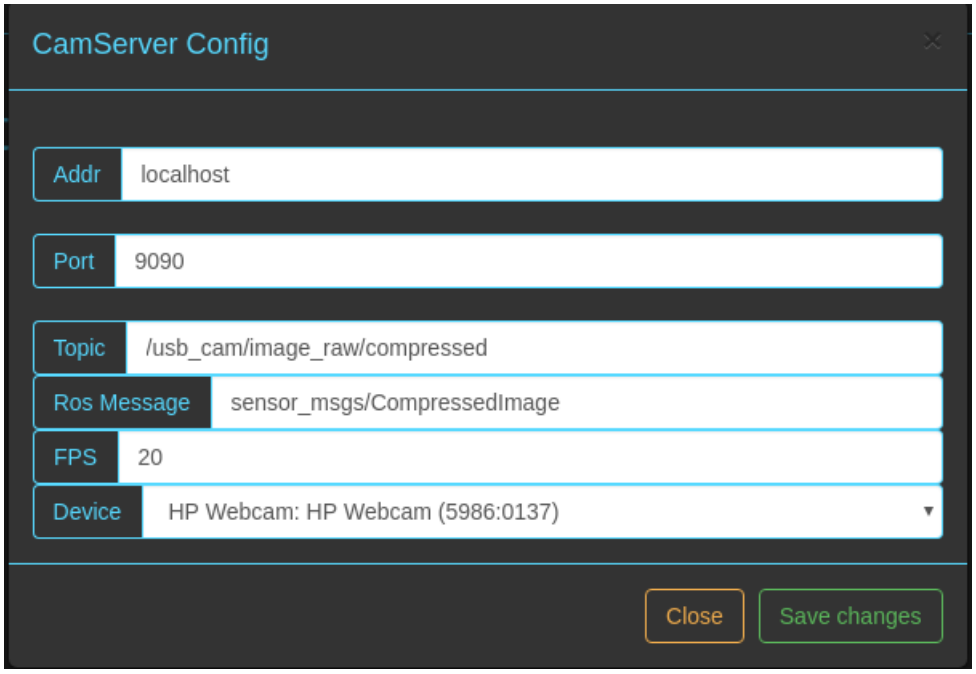


Figura 6.6: Ejemplo de configuración del driver

Finalmente, para verificar que está funcionando correctamente, en un tercer terminal lanzaremos la herramienta `rqt_image_view`, facilitada por ROS para visualizar imágenes enviadas a través de un mensaje de ROS, que hará la función de cliente.

[3] https://www.youtube.com/watch?v=taQzWUbYp-s
[4] https://www.youtube.com/watch?v=3PTJyvtEaDE

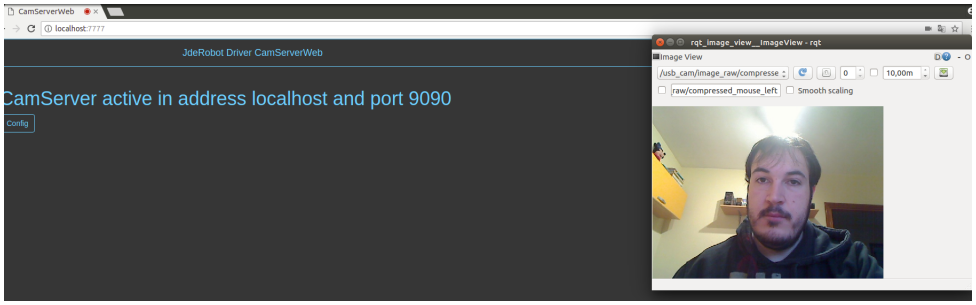


Figura 6.7: CamServerWeb ejecutado en un navegador web

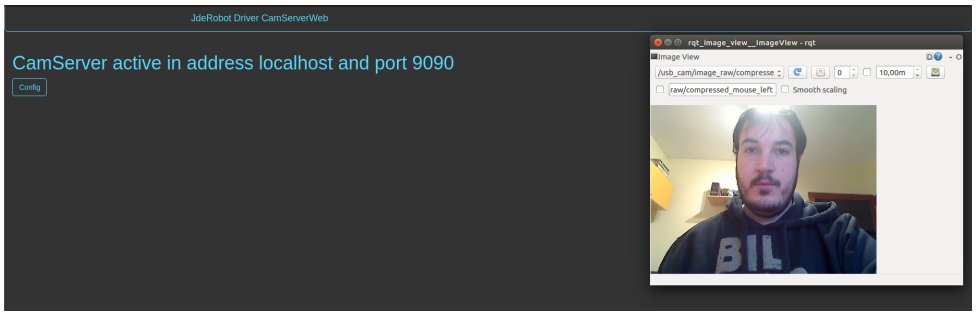


Figura 6.8: CamServerWeb ejecutado en Electron

Capítulo 7

Conclusiones

Una vez se ha detallado en los últimos capítulos todo el software desarrollado, ha llegado el momento de analizar las contribuciones realizadas y verificar si se han cumplido los objetivos marcados.

7.1. Contribuciones

El principal objetivo marcado en este trabajo era enriquecer las herramientas existentes en la plataforma JdeRobot con nuevas herramientas con tecnologías web. Se puede concluir que se ha alcanzado el objetivo satisfactoriamente realizando la conversión de los visores para su utilización con Electron y ROS, se ha aportado un nuevo visor de elementos 3D y se ha elaborado un nuevo driver para complementar los drivers para fuentes de video ya existentes en la plataforma JdeRobot.

El primer subobjetivo era modificar los tres visores web existentes previamente en la plataforma JdeRobot para que pudieran ser usados tanto en el navegador como con Electron y, a su vez, que pudieran conectarse mediante los middleware ICE y ROS. Como se ha explicado en el capitulo 4 de este trabajo, esta meta se ha alcanzado de manera satisfactoria, teniendo ahora nuevas versiones de los visores llamadas `CamVizWeb`, `TurtlebotVizWeb` y `DroneVizWeb`. Estos tres visores son capaces de comunicarse mediante los middleware de comunicación ICE y ROS, dotándoles de una gran versatilidad al poder conectar un amplio abanico de drivers robóticos.

La segunda meta era crear un nuevo visor 3D con WebGL que sustituyera al

visor existente desarrollado en C++, utilizado en la práctica de Robotics Academy de reconstrucción 3D. Este subobjetivo se ha alcanzado con la herramienta web `3DVizWeb` y, como se ha visto en el capitulo 5, no solo hasta el punto que se había marcado inicialmente (visualización de puntos), sino que se decidió ampliar el visor para que pudiera mostrar también segmentos y objetos 3D. No ha sido posible imitar completamente el funcionamiento del visor anterior, en el nuevo cambia la manera de comunicarse con las aplicaciones. El visor 3D es el encargado de llevar la iniciativa y debe ser él que solicite los elementos al servidor, y no que los reciba sin más como funcionaba anteriormente. Esta herramienta es muy útil para que otros desarrolladores e investigadores puedan representar escenas adquiridas mediante sensores en un mundo tridimensional.

La tercera meta fijada era crear un componente que pudiera obtener y enviar las imágenes obtenidas de una fuente de video WebRTC, complementando a los componentes `cameraserver` y `cameraserver_py` de JdeRobot que realizan la misma función pero desarrollados con C++ y Python respectivamente. Este subobjetivo se ha alcanzado con el componente `CamServerWeb` que, como se ha mostrado en el capitulo 6, obtiene imágenes mediante WebRTC de los dispositivos de video conectados y los transmite mediante el middleware de comunicación ROS. Con este componente se da la posibilidad de que cualquier aplicación de visión artificial pueda recibir las imágenes grabadas por las cámaras conectadas a un robot.

Además se ha conseguido que todas las herramientas puedan ser ejecutadas con Electron, lo que permite utilizarlas como una aplicación de escritorio, facilitando su uso por otros usuarios. Gracias a ello, todas las herramientas pueden ejecutarse a través de dos vías (Electron y navegador web) y son multiplataforma.

Analizando todo lo realizado y tras probar su eficiencia mediante las experimentaciones que han resultado exitosas en todos los casos, se puede concluir que se ha enriquecido con tecnologías web la plataforma JdeRobot, que hasta ahora su desarrollo principal era con C++ y Python.

Finalmente, a título personal, he ampliado mis capacidades con tecnologías web y aprendido el uso del entorno Electron que es muy útil para elaborar aplicaciones de escritorio pero programadas como si fueran web. He conocido el mundo de la robótica y como mediante middlewares como ICE o ROS es posible conectar sensores o actuadores a una aplicación para compartir información entre ambos.

7.2. Trabajos Futuros

Todas las herramientas creadas son completamente funcionales y cualquier usuario puede utilizarlas sin problemas, sin embargo existen varios aspectos donde se pueden extender para hacerlas aún más útiles.

- La primera posibilidad es permitir que tanto la herramienta `CamVizWeb` como el componente `CamServerWeb` sean compatibles con las imágenes en crudo de ROS y no solo imágenes comprimidas como ahora. Esta mejora permitiría conectar este visor al resto de drivers de fuentes de video.
- Otra posibilidad a modificar es conseguir que `3DVizWeb` reciba los elementos sin que haya realizado la petición previamente, de modo que la iniciativa la lleve enteramente la aplicación y no el visor. Este hecho permitirá que pueda conectarse al visor más de una aplicación simultáneamente, además reducirá el retardo del visor al reducir a la mitad el intercambio de mensajes.
- Empaquetar las herramientas web mediante paquetes npm, de modo que permita a otros usuarios descargar e instalar fácil y rápidamente las herramientas para ser usadas.

Bibliografía

[1] Darpa robotic challenge. https://www.darpa.mil/program/darpa-robotics-challenge.

[2] Zeroc documentation. http://wiki.ros.org/.

[3] Ros wiki. https://doc.zeroc.com/.

[4] Simulación en Gazebo. https://robologs.net/2016/06/25/gazebo-simulator-simular-un-robot-nunca-fue-tan-facil/.

[5] Definición de Robot. http://conceptodefinicion.de/robot/.

[6] Robot Web Tools Libraries and Widgets. http://robotwebtools.org/tools.html.

[7] Documentación Node.js. https://nodejs.org/es/docs/.

[8] Documentación Electron. https://electronjs.org/docs.

[9] WebGL Wiki. https://www.khronos.org/webgl/wiki/Main_Page.

[10] Three.js Manual. https://threejs.org/docs/.

[11] Getting Started WebRTC. https://webrtc.org/start/.

[12] W3Schools Online Web Tutorials. https://www.w3schools.com/.

[13] Aitor Martínez. Tecnologías web en la plataforma JdeRobot, 2015.

[14] Pablo Moreno. Nuevas prácticas docentes en el entorno Robotics-Academy, 2019.

[15] Galo Fariño R. Modelo Espiral de un proyecto de desarrollo de software, Administración y Evaluación de Proyectos. http://www.ojovisual.net/galofarino/modeloespiral.pdf.

[16] Modelo de desarrollo en Espiral. https://pdfs.semanticscholar.org/presentation/2403/7678f5cfd0d23d1a7d59b9b9ff2babc31d99.pdf.

[17] Simulador Gazebo. https://dev.px4.io/en/simulation/gazebo.html.

[18] Descripción de OpenCV. https://opencv.org/about.html.

[19] PX4 Developer Guide. https://dev.px4.io/en/.

[20] Teleoperation with MAVROS. http://docs.erlerobotics.com/simulation/vehicles/erle_copter/tutorial_4.