



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE  
TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS  
AUDIOVISUALES Y MULTIMEDIA

**TRABAJO FIN DE GRADO**

Por Definir

Autor: Roberto Pérez González

Tutor: José María Cañas Plaza

Curso académico 2018/2019

# 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

<b>1. Introducción</b>	<b>2</b>
1.1. Tecnologías de desarrollo web . . . . .	2
1.1.1. Introducción . . . . .	2
1.1.2. Arquitectura de una aplicación web . . . . .	3
1.1.3. Tecnologías del lado del cliente . . . . .	4
1.1.4. Tecnologías del lado del servidor . . . . .	4
1.2. Robotica . . . . .	6
1.2.1. Introducción . . . . .	6
1.2.2. Softwares para el desarrollo de componentes robóticos . . . . .	7
1.2.2.1. Middleware . . . . .	7
1.2.2.2. Bibliotecas . . . . .	8
1.2.2.3. Simuladores . . . . .	9
1.3. Tecnologías web en robótica . . . . .	10
1.3.1. Introducción . . . . .	10
1.3.2. Trabajos Previos . . . . .	11
1.3.2.1. CameraViewjs . . . . .	11
1.3.2.2. KobukiViewerjs . . . . .	12
1.3.2.3. UavViewerjs . . . . .	12
<b>2. Objetivos</b>	<b>14</b>
2.1. Objetivos . . . . .	14
2.2. Requisitos . . . . .	15
2.3. Metodología . . . . .	15

2.4. Plan de Trabajo . . . . .	17
<b>3. Infraestructura</b>	<b>19</b>
3.1. Node.js . . . . .	19
3.2. El entorno ROS . . . . .	20
3.2.1. Robot Web Tools . . . . .	22
3.3. El entorno ICE . . . . .	23
3.4. La plataforma JdeRobot . . . . .	24
3.5. WebGL y Three.js . . . . .	24
3.6. WebRTC . . . . .	25
3.7. El framework Electron . . . . .	26
<b>4. Servidor web de imágenes mediante ROS</b>	<b>28</b>
4.1. Introducción e infraestructura . . . . .	28
4.2. Interfaz gráfica . . . . .	29
4.3. Adquisición de las imágenes . . . . .	31
4.4. Conexión y transmisión de las imágenes mediante ROS . . . . .	33
4.4.1. Establecer conexión con ROS . . . . .	33
4.4.2. Definición de los mensajes ROS . . . . .	33
4.4.3. Creación de los mensajes y publicación . . . . .	34
4.5. Adaptación para su funcionamiento con Electron . . . . .	35
4.5.1. package.json . . . . .	35
4.5.2. main.js . . . . .	36
4.6. Ejecución del servidor de imagenes . . . . .	37
<b>5. Visor web de componentes 3D</b>	<b>40</b>
5.1. Introducción y Estructura . . . . .	40
5.1.1. Estructura del mensaje para visualizar los puntos . . . . .	42

5.1.2.	Estructura del mensaje para visualizar segmentos . . . . .	43
5.1.3.	Estructura del mensaje para visualizar un modelo 3D . . . . .	44
5.1.4.	Estructura del mensaje para mover los modelos 3D . . . . .	46
5.1.5.	Fichero de configuración . . . . .	47
5.2.	Interfaz gráfica . . . . .	48
5.2.1.	Escena base . . . . .	49
5.2.2.	Visualizar puntos . . . . .	50
5.2.3.	Visualizar segmentos . . . . .	51
5.2.4.	Visualizar modelos 3D . . . . .	53
5.2.5.	Creación y visualización de los modelos 3D . . . . .	53
5.2.6.	Mover los modelos 3D . . . . .	57
5.2.7.	Borrar elementos mostrados en el visor . . . . .	58
5.3.	Conexión con el servidor y recepción de los objetos 3D . . . . .	59
5.3.1.	Interfaces Slice . . . . .	59
5.3.1.1.	primitives.ice . . . . .	60
5.3.1.2.	pose3d.ice . . . . .	61
5.3.1.3.	visualization.ice . . . . .	62
5.3.2.	Conexión y peticiones al servidor . . . . .	64
<b>6.</b>	<b>Visores web modificados para ser usados con Electron</b>	<b>69</b>
<b>7.</b>	<b>Conclusiones</b>	<b>70</b>

# Capítulo 1

## Introducción

En este primer capítulo voy a tratar de explicar el contexto de las bases tecnológicas en las que se apoya este proyecto, que son principalmente las tecnologías web y la robótica. Para empezar hablare del contexto de las tecnologías web y como de importantes son en la sociedad actual, para continuar explicando su arquitectura y las tecnologías más importantes. Continuare hablando sobre el estado actual de la robótica y la gran expansión de la misma en nuestros días. Para finalizar este capitulo, introduciré en Jderobot y en los proyectos previos que combinan tecnologías web con componentes robóticos.

### 1.1. Tecnologías de desarrollo web

#### 1.1.1. Introducción

Desde que en el año 1992, Tim Berners-Lee ideara y desarrollara las primeras herramientas para facilitar compartir información entre los científicos del CERN desde cualquier parte del mundo, dando lugar a la posteriormente llamada World Wide Web o como se conoce coloquialmente la web, ha sufrido una gran evolución, consiguiendo que la sociedad actual no se pueda entender sin la existencia de la misma. Sin embargo, pese a la gran evolución, el propósito de la web no ha cambiado, es decir que acceder a la información sea lo más fácil posible, si lo ha hecho la manera en que la utilizamos. La aparición de aplicaciones web como las redes sociales, los servicio de streaming o los comercios electrónicos, han supuesto un impulso importante en el uso de la web y, por

consiguiente, la necesidad de idear nuevas herramientas que faciliten el desarrollo de webs.

### 1.1.2. Arquitectura de una aplicación web

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

- 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 realiza 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 creado por Berners-Lee que permite el intercambio de información entre el cliente y el servidor.



Figura 1.1: Arquitectura de una aplicación web

### 1.1.3. Tecnologías del lado del cliente

Estas tecnologías 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 estas tecnologías. Las más utilizadas son las siguientes:

- Hyper-Text Markup Language (HTML): Es el lenguaje de descripción de aplicaciones web que nos permite especificar las características visuales.
- Hojas de estilo en cascada (CSS): Es el lenguaje utilizado para describir la presentación semántica (el aspecto y el formato) de un documento en lenguaje de marcas.
- JavaScript: Es un lenguaje de script orientado a objetos y guiado por eventos que nos permiten realizar acciones en el cliente e interactuar con el servidor u otras aplicaciones web.

### 1.1.4. Tecnologías del lado del servidor

Estas tecnologías 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, se encarga de realizar todas las tareas necesarias para crear la aplicación web que se visualizara en el cliente. Las tecnologías más utilizadas son las siguientes:

- Common Gateway Interface (CGI): Fue de las primeras tecnologías del lado del servidor en aparecer, se creó inicialmente para gestionar formularios. No se trata de un lenguaje de programación sino de un mecanismo de comunicación entre el cliente y un programa externo, que proporcionara el contenido de la aplicación web. Actualmente se utiliza una variante llamada Fast-CGI que proporciona una mayor rapidez.
- PHP: Creado en 1994, es la tecnología más utilizada. Se trata de un lenguaje de programación de uso general de script que originalmente fue diseñado para proporcionar contenido web dinámico. Su código está empujado en el código HTML y es interpretado por un servidor web para generar la aplicación web, evitando la necesidad de acceder a un archivo externo.



- **Servlets:** Son programas escritos en Java que se ejecutan sobre un servidor de aplicaciones mediante la maquina virtual de Java (JVM). Estos programas (servlets) son ejecutados por el servidor para manejar cada una de las peticiones del cliente. Se trata de una tecnología con un concepto similar a CGI, pero beneficiándose de las ventajas de el entorno Java.
- **JavaServer Pages (JSP):** Se trata de una tecnología Java que permite generar contenido web dinámico en forma de documentos HTML. El código Java, a diferencia que en los Servlets, va incrustado en el HTML y se compila dinámicamente como un servlet.
- **Active Server Page (ASP):** Creado por Microsoft en 1996. Se trata de código que se ejecuta en el servidor y genera un archivo HTML que devuelve al cliente. Al ser una tecnología creada por Microsoft, permite la compatibilidad con componentes ActiveX (acceso a base de datos, scripts, etc) lo que proporciona una gran potencia y flexibilidad. Esta tecnología solo puede ser utilizada en servidores con sistemas operativos de Microsoft.
- **Python, Django:** Se trata de un framework 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 se trata de un modelo 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 paso por Django que analizara la URL solicitada y pasara la petición a la función correspondiente llamada vista. En esta función se ejecutara lo necesario para proporcionar al cliente lo solicitado con su URL.
- **Ruby on Rails:** Al igual que Django, Ruby on Rails se trata de un framework 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:** Se trata de un entorno de ejecución multiplataforma de código abierto para el lado del servidor basandose en el lenguaje JavaScript. Este entorno 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 con robustas.

## 1.2. Robotica

### 1.2.1. Introducción

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. El termino robótica proviene del escritor y profesor de bioquímica Isaac Asimov, quien, en su Saga de la Fundación, definió las leyes de la robótica:

- Primera ley: Un robot no puede hacer daño a un ser humano ni, por inacción, permitir que un ser humano sufra daño.
- Segunda ley: Un robot debe obedecer las órdenes dadas por los seres humanos, excepto cuando estas entren en conflicto con la primera ley.
- Tercera ley: Un robot debe proteger su propia integridad, siempre y cuando esto no impida el cumplimiento de la primera y segunda ley.

Desde que Asimov acuñara estas tres leyes, la sociedad a temido lo que los robot puedan hacer al ser humano, desde esclavizarnos hasta quitarnos el trabajo, siendo este miedo el más reciente y el que ha supuesto un mayor freno al desarrollo. 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 remplazar al ser humano como nos muestran tantas obras literarias y cinematográficas, sino que vemos robótica a lo largo de nuestro día a días, desde un brazo mecánico en una cadena de montaje, hasta un aspirador robótico. Gracias a la robótica, tareas 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 la robotización de las mismas.

Todo robot debe tener una parte hardware (el componente robótico ya sea real o simulado) y una parte software (la lógica de ese componente robótico).

### 1.2.2. Softwares para el desarrollo de componentes robóticos

El software es el encargado de proporcionar al robot la inteligencia y autonomía necesaria para que realice las funciones o acciones que deseamos. Para facilitar esta tarea, sean creado middleware y bibliotecas específicas.

#### 1.2.2.1. Middleware

Un robot es un sistema complejo ya que es un conjunto de diferentes componentes hardware y software (sensores, actuadores, etc). Para cada robot, sería necesario un software personalizado que sea capaz de: leer y extraer la información necesaria de cada uno de los sensores, calcular y enviar las secuencias de acciones para que realice una determinada tarea o acción, controlar los actuadores, etc. Desarrollar este software se vuelve una tarea compleja e inútil dado que por lo general, si cambiáramos de robot este software no funcionaría de manera correcta. En este escenario es donde un middleware nos proporcionara las herramientas que necesitamos para facilitar en gran medida la labor, ya que nos permitirá estructurar el software 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 gracias al middleware. Actualmente hay una gran cantidad de middleware que permiten esta abstracción, siendo algunos de los más destacados los que señalo a continuación:

- Robot Operating System (ROS)<sup>1</sup>: Pese a ser un middleware, se podría decir que ROS pretende ser algo más. Nos 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). El principal objetivo es permitir la reutilización del código en la investigación y desarrollo de robótica, lanzando 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.
- Internet Communication Engine (ICE)<sup>2</sup>: Creado por la empresa ZeroC, se trata de

---

<sup>1</sup><http://www.ros.org/>

<sup>2</sup><https://zeroc.com/>

un middleware orientado a objetos que permite la creación de aplicaciones multilenguaje y multiplataforma, permitiendo la comunicación con diferentes arquitecturas de red (UDP, TCP, WebSockets, etc). ICE no es un middleware robótico propiamente dicho, pero su uso facilita el desarrollo de la conectividad con sistemas robóticos, al permitirnos crear nuestras propias interfaces para el intercambio de datos.

- Open Robot Control Software project (Orocos) <sup>3</sup>: Se trata de un proyecto de software libre cuyo objetivo es la creación de un paquete de software para el control de robots.
- Orca <sup>4</sup>: Se trata de un middleware de código libre que se originó a partir del proyecto Orocos. Orca está diseñado para desarrollar sistemas robóticos basados en componentes. Su principal objetivo es permitir, facilitar y simplificar la reutilización del código entre desarrolladores.
- Middleware for Robots (Miro): Se trata de un middleware orientado a objetos para el control de robots móviles, y basado en CORBA (Common Object Request Broker Architecture). El hecho de que sea orientado a objetos permite la interoperabilidad entre procesos y plataformas cruzadas para el control de robots distribuidos.
- JdeRobot <sup>5</sup>: Se trata de una plataforma de código abierto 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++.

### 1.2.2.2. Bibliotecas

En programación, una biblioteca es una colección de recursos utilizados para el desarrollo de software. Hay multitud de bibliotecas para el desarrollo de sistemas robóticos: bibliotecas para procesar imágenes, interactuar con el robot, procesar elementos 3D, etc. Las bibliotecas más utilizadas son:

- OpenCV <sup>6</sup>: Es la biblioteca de visión artificial por excelencia. Creada originalmente por Intel, es de código abierto y desarrollada en C++. Actualmente dispone

---

<sup>3</sup><http://www.oroocos.org/>

<sup>4</sup><https://www.orcaconfig.com/>

<sup>5</sup><https://jderobot.org/>

<sup>6</sup><https://opencv.org/>

de interfaces en C++, C, Python, Java y están empezando a desarrollar para JavaScript.

- Point Cloud Library (PCL) <sup>7</sup>: Se trata de una biblioteca de código abierto que proporciona algoritmos para el procesamiento de nubes de puntos y geometría 3D.
- Bibliotecas de ROS: ROS proporciona bibliotecas para cada lenguaje de programación al que da soporte, ofreciendo una serie de funciones y algoritmos que nos permite crear aplicaciones que interactúan rápidamente con ROS. Las bibliotecas más utilizadas son rospy <sup>8</sup>, que está desarrollada para Python y roscpp <sup>9</sup>, que está desarrollada para ser usada con C++. Cabe destacada, también, la biblioteca roslibjs, que ofrece las funcionalidades para JavaScript.

### 1.2.2.3. Simuladores

Un simulador robótico nos permite imitar el funcionamiento de un sistema robótico para poder 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 a nivel monetario y de tiempo, que conllevarían retrasos importantes a la hora de realizar el desarrollo. Los simuladores más utilizados son:

- Gazebo <sup>10</sup>: Se trata de uno de los simuladores más utilizados en la actualidad gracias a su fácil manejo y su intuitiva interface. Gazebo es un simulador de código abierto que ofrece múltiples motores de físicas, motores de renderizado avanzado, soporte para plugins 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 utilizar nuestra aplicación con el sistema físico sin miedo a que sufra daños.
- ROS Development Studio (RDS <sup>11</sup>): Elaborado por la empresa española TheConstruct, se trata de un simulador web que permite simular sistemas robóticos, a la

---

<sup>7</sup><http://pointclouds.org/>

<sup>8</sup><http://wiki.ros.org/rospy>

<sup>9</sup><http://wiki.ros.org/roscpp>

<sup>10</sup><http://gazebo-sim.org/>

<sup>11</sup><http://www.theconstructsim.com/>

vez que ofrece un editor y una consola para poder crear nuestro código, pero únicamente ofrece soporte para Python gracias a Jupyter. La gran ventaja que ofrece este simulador es que no es necesario realizar ninguna instalación, simplemente registrarte en su página web y acceder con un navegador. Ofrecen desde una versión básica gratuita hasta una versión para expertos con una tarifa mensual.

Existen otros simuladores como son Stage para la simulación en 2D o Webots, pero no tan utilizados como Gazebo o sin las ventajas de RDS.

### 1.3. Tecnologías web en robótica

#### 1.3.1. Introducción

La utilización de tecnologías web en robótica es aún, pese a su aumento en los últimos tiempos, 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.

Como he mencionado anteriormente, aún no hay muchos desarrollos basados en tecnologías web, siendo el más importante de ellos las bibliotecas y herramientas de código abierto para su utilización con el middleware ROS, elaboradas por la comunidad de Robot Web Tools <sup>12</sup>. Los desarrollos más importantes que han realizado son:

- Rosbridge Suite: Proporciona una interfaz usando JSON para ROS que permite a cualquier cliente enviar JSON para conectarnos 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 interacción 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.

---

<sup>12</sup><http://robotwebtools.org/>

Adicionalmente ofrecen una serie de herramientas como son 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.

### 1.3.2. Trabajos Previos

Este proyecto está basado en el TFG elaborado por Aitor Martinez <sup>13</sup>. En este trabajó, utilizando tecnologías web, creo tres aplicaciones que eran capaces de conectarse con varios sistemas robóticos.

#### 1.3.2.1. CameraViewjs

Se trata de 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. Esta aplicación 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.

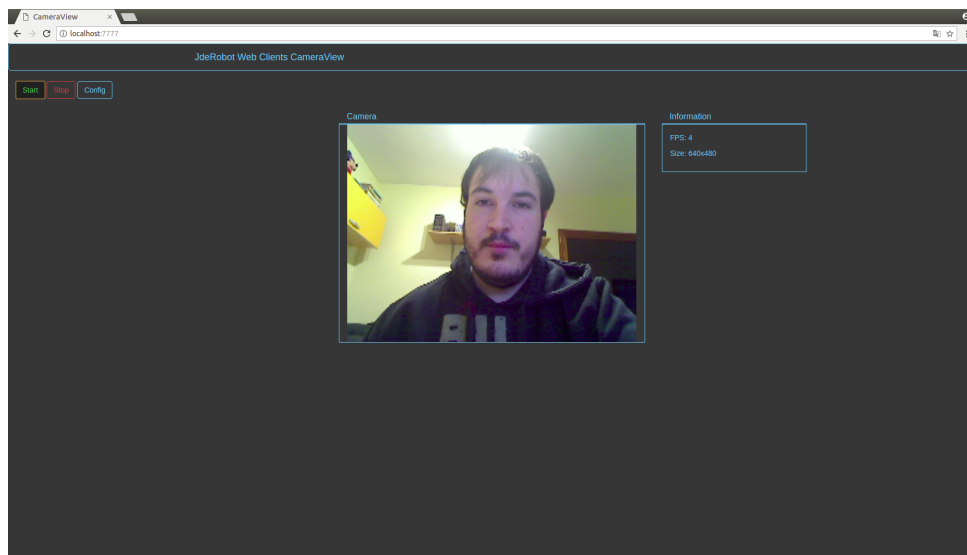


Figura 1.2: CameraViewjs

---

<sup>13</sup><https://jderobot.org/Aitormf-tfg>

### 1.3.2.2. KobukiViewerjs

Se trata de un visualizador y teleoperador de robots 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 la imagen del escaneo laser obtenida, 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 linear recta como la orientación del mismo.

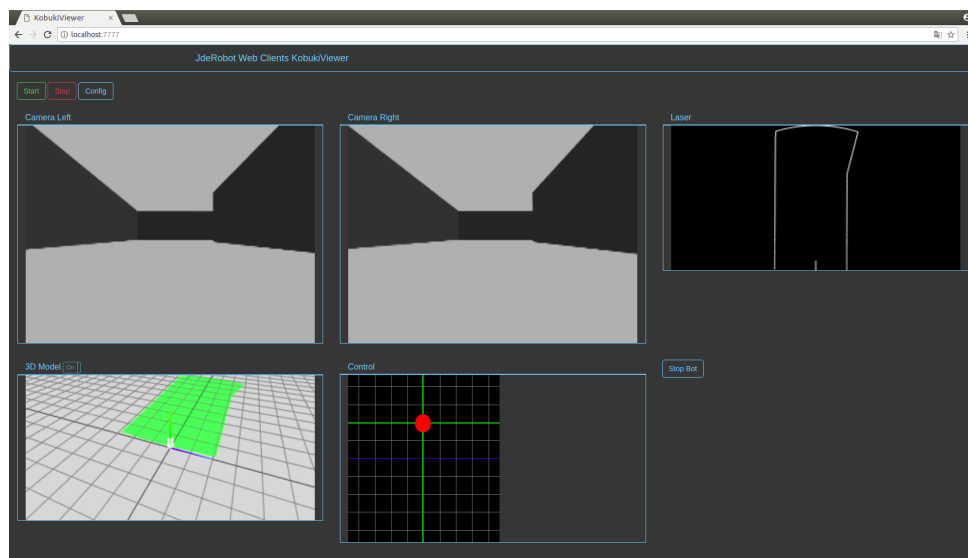


Figura 1.3: KobukiViewerjs

### 1.3.2.3. UavViewerjs

Se trata de 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 de abajo 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 cuando se desea aterrizar y despegar.



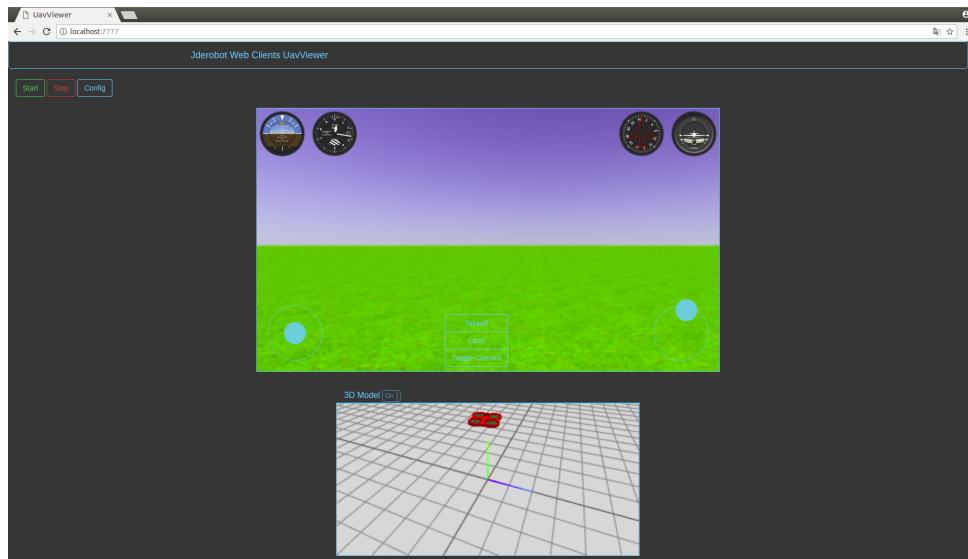


Figura 1.4: UavViewerjs

# Capítulo 2

## Objetivos

En este capítulo se presentaran los objetivos planteados para este proyecto, los requisitos marcados y la metodología utilizada 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:

- Modificar los tres visores elaboradas mediante tecnologías web existentes en la plataforma JdeRobot, CameraViewjs, KobukiViewerjs y UavViewerjs, para su utilización con el middleware ROS, además de ICE como hasta ahora.
- Elaborar un nuevo driver robótico utilizando tecnologías web, cuyo cometido sera el de servidor de imágenes a los distintos visores existentes en JdeRobot.
- Creación mediante tecnologías web, de un nuevo visor, que permitirá la visualización de elementos 3D (puntos, líneas y modelos 3D) que utilizara como middleware de comunicación, ICE.

Además, se ha utilizado el framework Electron en todos ellos para posibilitar que, además de usando un navegador, se puedan ejecutar como si de una aplicación de escritorio se tratase.

### 2.2. Requisitos

Los requisitos que se han tenido en cuenta a la hora de desarrollar todo el software de este trabajo han sido los siguientes:

- El sistema operativo utilizado ha sido Ubuntu 16.04 LTS y macOS HighSierra, además, gracias a utilizar tecnologías web, todos los componentes pueden ser ejecutados en cualquier sistema operativo.
- Se ha utilizado la versión de NodeJS 8.9.1 y de NPM 6.4.0.
- Los middleware utilizados han sido JdeRobot en su versión 5.6.4, ROS en su versión Kinetic y ICE en su versión 3.6.4.
- Para permitir ejecutar la ejecución de todo los elementos como una aplicación de escritorio, se ha utilizado Electron en la versión 1.8.
- Se ha utilizado HTML5, CSS3 y JavaScript para la elaboración de los software.
- El funcionamiento con el middleware ROS sera mediante la biblioteca Rosbridge Server de RobotWebTools.

### 2.3. 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 esta 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 poder llevar un mejor control 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<sup>1</sup> de JdeRobot, donde quede reflejado todos los progresos así como videos demostrativos de los avances. También se dispone de un repositorio en github<sup>2</sup> donde se ha ido subiendo todo el código para su verificación y prueba por personas externas.

### 2.4. Plan de Trabajo

De cara a cumplir los objetivos marcados y enfocar mejor los posibles problemas y sus posibles soluciones, lo primero que se ha realizado es una familiarización con el entorno de trabajo.

1. Comprender el funcionamiento de Electron y realización de pequeñas pruebas para aprender a usarlo.
2. Familiarización con el entorno JdeRobot y Gazebo.
3. Comprender el funcionamiento de los visores existentes desarrollados en tecnologías web (CameraViewjs, KobukiViewerjs y UavViewerjs) y ejecutarlos para analizar su funcionamiento.
4. Realizar las modificaciones para que funcionen con el framework Electron.
5. Comprender el funcionamiento de los middleware ICE y ROS, realizando pequeñas pruebas.
6. Modificación de los visores para que funcionen tanto con ICE (como hasta ahora), como con ROS.
7. Creación de nuevo driver para servir imágenes con el middleware ROS.
8. Comprender el funcionamiento del visor 3D existente en la plataforma JdeRobot desarrollado en C++, así como su conectividad con la práctica de Reconstrucción 3D de JdeRobot Academy.
9. Elaboración del visor 3D con tecnologías web y el middleware ICE para que muestre puntos y se conecte con los elementos de la práctica de Reconstrucción 3D.

---

<sup>1</sup><https://jderobot.org/Rperez-tfg>

<sup>2</sup><https://github.com/RoboticsURJC-students/2017-tfg-roberto-perez>

10. Ampliación del visor 3D para que además de los puntos, muestre también segmentos y modelos 3D enviados por un cliente escrito en cualquier lenguaje de programación.

# Capítulo 3

## Infraestructura

En este capítulo se describirán las tecnologías sobre las que se cimienta este trabajo. Para empezar, se detallará el funcionamiento del framework Electron y Node.js, ya que todo el desarrollo de este trabajo será compatible con él. Se profundizará en los dos middleware utilizados, ICE y ROS, y en el entorno JdeRobot. Para finalizar, se describirá el API para el renderizado de gráficos WebGL, concretamente la biblioteca Three.js, y WebRTC para la conexión con los elementos multimedia.

### 3.1. Node.js

Node.js <sup>1</sup> 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, nos permite ejecutar código JavaScript fuera de un navegador y aprovechar las ventajas que nos 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. Sin embargo, al ser de código abierto, existe una gran comunidad de desarrolladores que crean nuevos módulos para que cualquiera pueda utilizarlos.

---

<sup>1</sup><https://nodejs.org/es/>

Estos módulos son fácilmente compartidos gracias al manejador de paquetes npm <sup>2</sup>, permitiéndonos 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 usara nuestro programa y nos facilitara su instalación conjunta usando el comando npm install, así como se referenciara al fichero .js indicado anteriormente.

### 3.2. El entorno ROS

Se trata de un Framework 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.). ROS ofrece una serie de herramientas, bibliotecas y convenciones para simplificar la tarea de crear complejos y robustos 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 otra persona pueda utilizarlo para usarlo en su proyecto.

El elemento fundamental del funcionamiento de ROS es el nodo. Un nodo es un proceso que realiza cálculos y se comunican entre sí mediante un sistema de publicación - suscripción para conexiones asíncronas y anónimas, o servicios para las conexiones síncronas. Los nodos operan en una escala de bajo nivel, es decir cada nodo se ocupa de una parte del sistema de control de robot, por ejemplo un nodo se ocupa de los motores, otro nodo se ocupa 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, 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 suscriba 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).

---

<sup>2</sup><https://www.npmjs.com/>



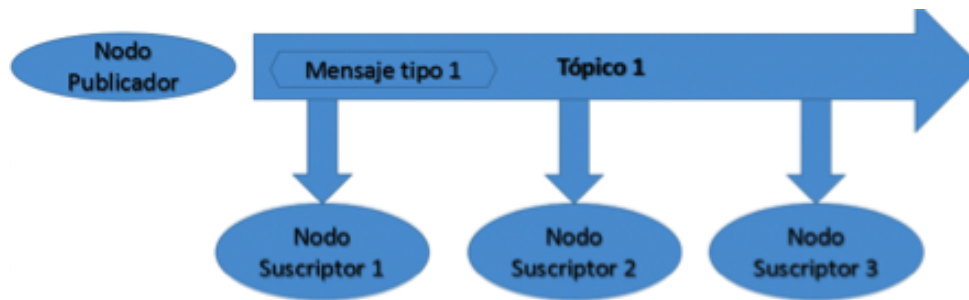


Figura 3.1: Estructura del sistema de comunicación Publicador - Subscriber

El sistema de comunicación síncrona, es el típico sistema cliente-servidor 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 el tipo de servicio que define la cantidad y tipo de datos que necesita el servicio tanto para recibir como petición, como para enviar como respuesta.



Figura 3.2: Estructura del sistema de comunicación por Servicios ROS

Un elemento que es muy útil para este proyecto es que ROS nos proporciona total compatibilidad con Gazebo mediante un conjunto de paquetes llamados *gazebo\_ros\_pkgs*<sup>3</sup>. 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.

<sup>3</sup>[http://wiki.ros.org/gazebo\\_ros\\_pkgs](http://wiki.ros.org/gazebo_ros_pkgs)

### 3.2.1. Robot Web Tools

Comunidad nacida a partir de la de ROS, nos permite conectar aplicaciones web a elementos robóticos gracias a un protocolo llamado Rosbridge. Este protocolo es 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`, sin embargo en este trabajo solo se utilizara `roslibjs`.

La biblioteca `roslibjs` es la encargada de ofrecernos las funcionalidades necesarias para conectarnos, enviar o recibir mensajes ya sea mediante publicación y subscripción, o servicios. La conexión se realiza a un servidor intermedio (servidor Rosbridge), el cual se encarga de gestionar las conexiones, los topic publicados o los servicios activos en cada momento, de modo que cuando te subscribas o realices la petición de un servicio, este servidor se encargue de encaminar la petición mediante WebSockets.

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

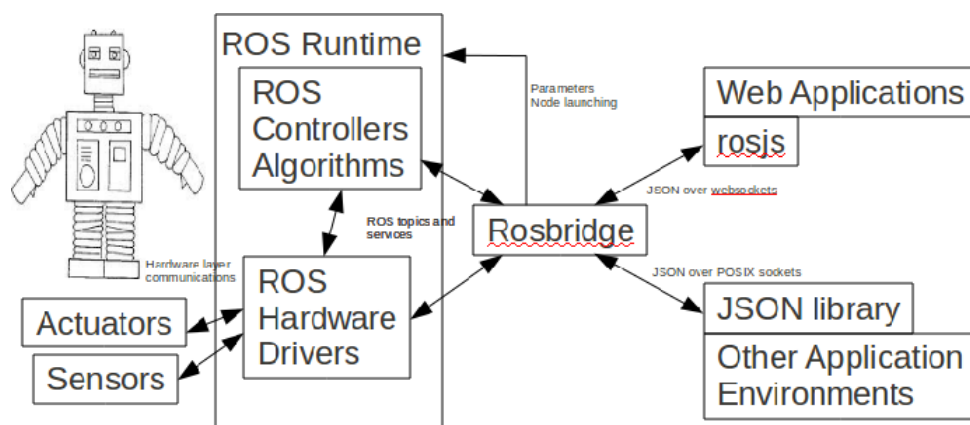


Figura 3.3: Estructura de una aplicación con Rosbridge

### 3.3. El entorno ICE

Se trata de un Framework orientado a objetos que ayuda a crear aplicaciones distribuidas fácilmente. ICE se ocupa de todas las interacciones con las interfaces de programación de bajo nivel de red (inhibe al desarrollador de la tarea de apertura de puertos, conexiones de red o serialización de datos). El objetivo principal de ICE es facilitar su uso y el desarrollo de aplicaciones, de modo que en muy poco tiempo se pueda aprender a utilizarlo.

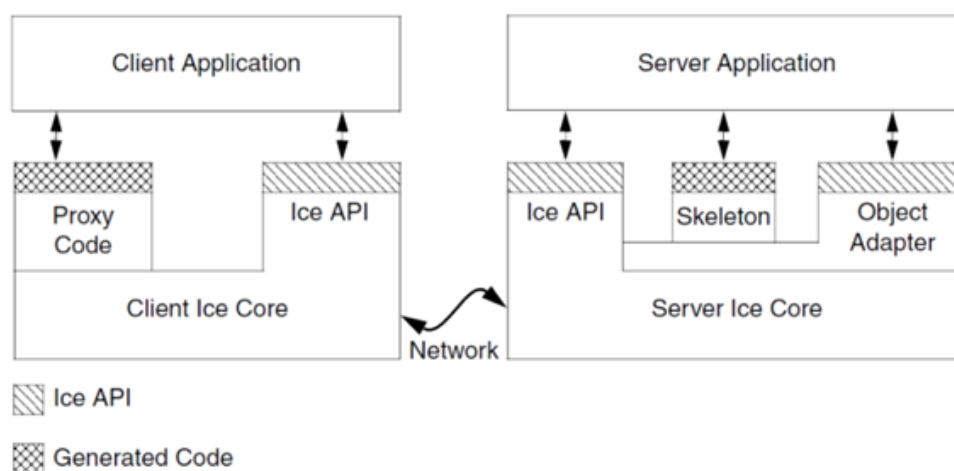


Figura 3.4: Estructura de Cliente - Servidor con Ice

ICE tiene un lenguaje de especificación propio llamado Slice (Specification Language for ICE) que nos permite la abstracción fundamental para separar interfaces de objetos de sus implementaciones. Este lenguaje especifica las interfaces, operaciones y tipos de parámetros utilizados por nuestra aplicación. Cada una de las aplicaciones que queramos que interaccionen entre sí, deben compartir la misma descripción Slice. Esta descripción es independiente del lenguaje en el que está desarrollada nuestro cliente o servidor de modo que sea posible su utilización con clientes y servidores escritos en diferentes lenguajes de programación.

Para este trabajo se va a utilizar su versión para JavaScript, sin embargo 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 disponibles para JavaScript. Sobre todo este hecho es manifiesto en que 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.

### 3.4. La plataforma JdeRobot

JdeRobot <sup>4</sup> es el framework de software libre para el desarrollo de robótica y visión artificial creado por el grupo de robótica de la Universidad Rey Juan Carlos y licenciado bajo GPL v3 <sup>5</sup>. Su desarrollo principalmente está realizado mediante C, C++ y Python, incorporando desarrollo en JavaScript como el tratado 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 ordenes 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 ordenes), ya sean reales o simulados. Estas funciones locales forman la API de la capa de abstracción. La plataforma también ofrece una serie de herramientas para facilitar la teleoperación o el tratamiento de las mediciones de los sensores, y bibliotecas.

### 3.5. WebGL y Three.js

WebGL <sup>6</sup> es un API multiplataforma utilizada 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, o 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 <sup>7</sup> nace como remedio a la complejidad de usar WebGL. Se trata de una

---

<sup>4</sup>[https://jderobot.org/Main\\_Page](https://jderobot.org/Main_Page)

<sup>5</sup><http://www.gnu.org/licenses/gpl-3.0-standalone.html>

<sup>6</sup><https://get.webgl.org/>

<sup>7</sup><https://threejs.org/>

biblioteca desarrollada en JavaScript que permite crear y mostrar gráficos 3D en un navegador web usando una API de alto nivel, proporcionados 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))`, nos permite crear una esfera 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 textura obtenida desde una imagen)

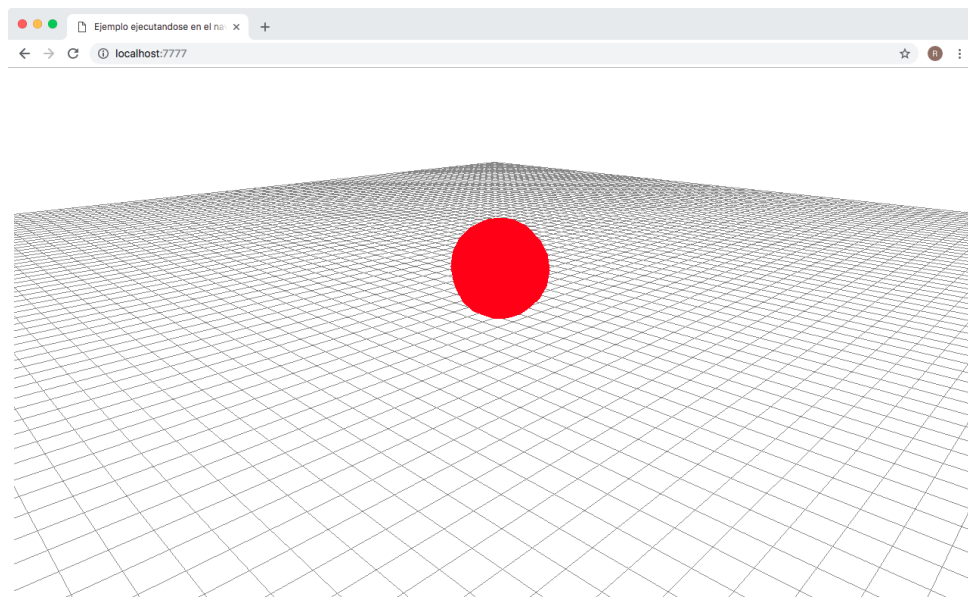


Figura 3.5: Esfera creada con la biblioteca Three.js y mostrada en un navegador

### 3.6. WebRTC

WebRTC <sup>8</sup> una tecnología que permite a una aplicación web capturar y transmitir audio y video, así como intercambiar datos con otros navegadores sin necesidad de intermediarios. Este intercambio se realiza de igual a igual (peer-to-peer) sin necesidad de instalaciones o softwares adicionales. WebRTC proporciona varias 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.

---

<sup>8</sup><https://webrtc.org/>

En este proyecto, usaremos WebRTC para la adquisición de del video obtenido a través de una cámara web. Para lograr este objetivo, se utilizara el API de WebRTC Media Stream, que nos proporcionara 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.

### 3.7. El framework Electron

Electron <sup>9</sup> es un framework de código abierto desarrollado por GitHub. Comenzó su desarrollo en 2013, en el mismo grupo de trabajo del editor Atom <sup>10</sup>. Concebido con la idea de permitir la creación de aplicaciones de escritorio multiplataforma con tecnologías web. Electron es la combinación de NodeJS y Chromium <sup>11</sup> en una misma ejecución.

La arquitectura de una aplicación que utiliza Electron esta 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 mas de una pagina web al mismo tiempo. Esta labor se realiza mediante la instancia al objeto BrowserWindow de Electron, ejecutandose una página web cada vez que se instancia. Cuando se destruye una de estas instancia, se esta 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 instanacia al objeto BrowserWindow y la ejecucion de la página web correspondiente. Una aplicación con Electron puede tener multitud de procesos renderizadores, siendo cada uno independiente del resto. Cada proceso solos se preocupa de la página web que se esta 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. Este nos aporta un gran número de ventajas

---

<sup>9</sup><https://electronjs.org/docs>

<sup>10</sup><https://atom.io/>

<sup>11</sup><https://www.chromium.org/Home>

como una mayor seguridad al cargar contenido remoto, tener siempre actualizadas nuestras aplicaciones o tener un gran número de bibliotecas disponibles.

Todas las aplicaciones desarrolladas en este trabajo podrán ser ejecutadas utilizando Electron, lo que nos permite utilizarlas en cualquier plataforma o, incluso, empaquetarlas usando npm o mediante un archivo Asar <sup>12</sup>.

Una aplicación que utiliza Electron estará formada por un archivo HTML, un fichero main.js que definirá la ventana donde se mostrara el fichero HTML y se creara la misma, y, al igual que se indica en la sección de Node.js, el archivo Package.json que definirá nuestra aplicación en Electron (nombre, versión, descripción, etc.), se indicaran las dependencias a módulos externos y el fichero main.js para que creé la aplicación. Una vez que contamos con al menos estos tres elementos, podemos instalar las dependencias mediante npm install (igual que para Node.js) y ejecutar la aplicación mediante npm start o npm test (dependerá de como hayamos definido Package.json).

En la figura 3.6 se muestra el mismo HTML ejecutado con Node.js y con Electron. Como se puede apreciar, el resultado es el mismo.

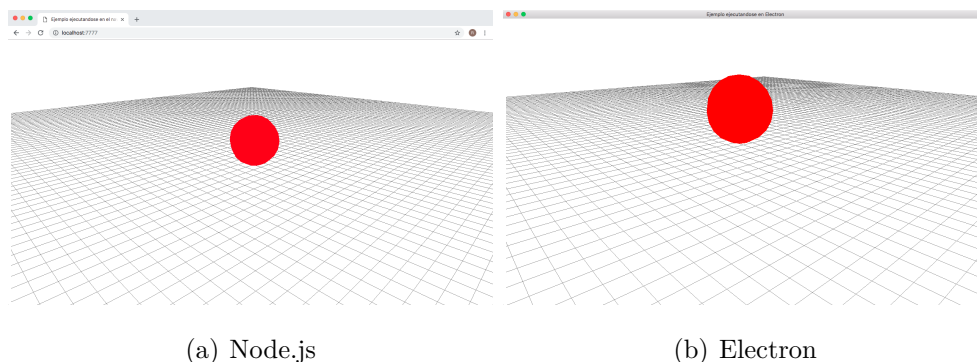


Figura 3.6: Ejemplo de la esfera anterior ejecutado con Node.js y con Electron

---

<sup>12</sup><https://github.com/electron/asar>

## Capítulo 4

# Servidor web de imágenes mediante ROS

En este capítulo expondrá la creación de un nuevo driver para la plataforma JdeRobot. Este driver será un servidor de imágenes al que se ha llamado camServerWeb, obtenidas mediante una webcam (ya sea externa o interna del ordenador) de modo que cualquier aplicación externa pueda obtener las imágenes obtenidas.

### 4.1. Introducción e infraestructura

Este driver está diseñado con JavaScript y HTML como lenguajes de programación, y ROS como middleware para la interconexión con los diferentes clientes. La imagen 4.1 muestra un esquema de funcionamiento.



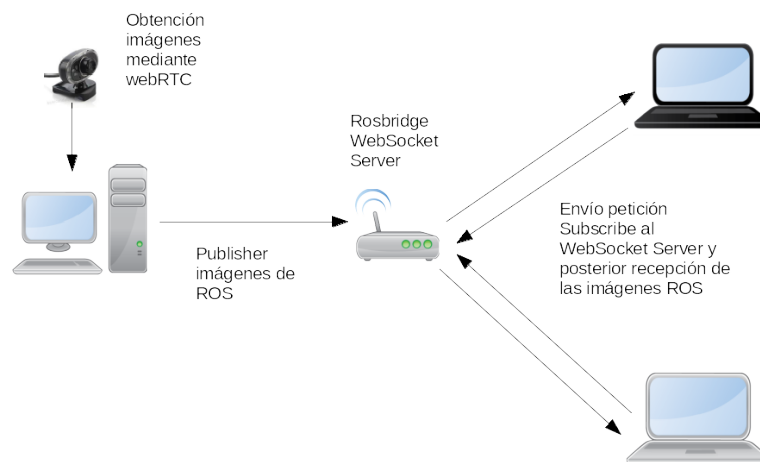


Figura 4.1: Esquema del funcionamiento del servidor de imágenes

La imagen se obtiene de la webcam de la que se desee (por defecto se selecciona la webcam interna del ordenador) mediante webRTC. Esta imagen se transmite mediante el middleware ROS utilizando el modelo Publicador/Subscriptor. El driver será el Publicador, y lo hará estableciendo un topic o etiqueta y usando las imágenes comprimidas de ROS (CompressedImage). Dado que, como se ha explicado en los capítulos anteriores, ROS no ofrece soporte por defecto para JavaScript, se usa roslibjs. Una vez se ha comenzado a publicar las imágenes, cualquier aplicación que se suscriba al topic, podrá recibirlas, dando igual el lenguaje de programación o sistema operativo que utilice. El driver se divide en tres partes, una primera parte que es la interfaz gráfica, una segunda que es la conexión y obtención de las imágenes, y la tercera y última que es la correspondiente a la transmisión mediante ROS.

## 4.2. Interfaz gráfica

Dado que es un driver y su cometido no es mostrar nada, la interfaz gráfica que se ha creado es muy simple. En ella únicamente se mostrara si está conectado y a la IP y Puerto al que lo está. La interfaz está realizada mediante HTML y Bootstrap, siguiendo

el modelo del resto de aplicaciones web de la plataforma JdeRobot.

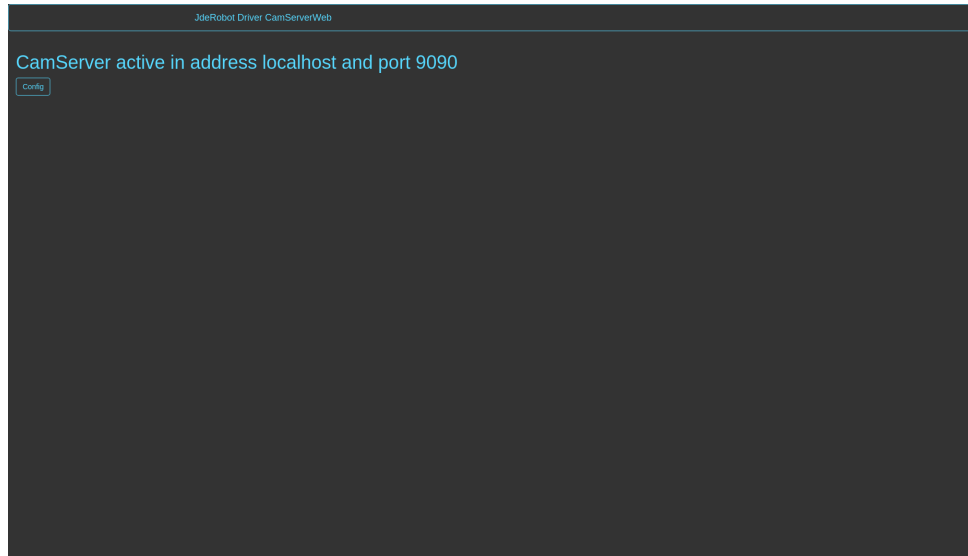


Figura 4.2: Interfaz gráfica del driver

La única funcionalidad destacable de la interfaz y el motivo para que haya sido necesario aportar una interfaz, es el menú de configuración. En este menú el usuario establecerá la IP y el puerto por el que se va a transmitir los mensajes, el topic al que se deberán subscribir los clientes y el tipo de mensajes, y la cámara de la que queremos obtener las imágenes que, como se indica anteriormente, por defecto será la interna del ordenador.

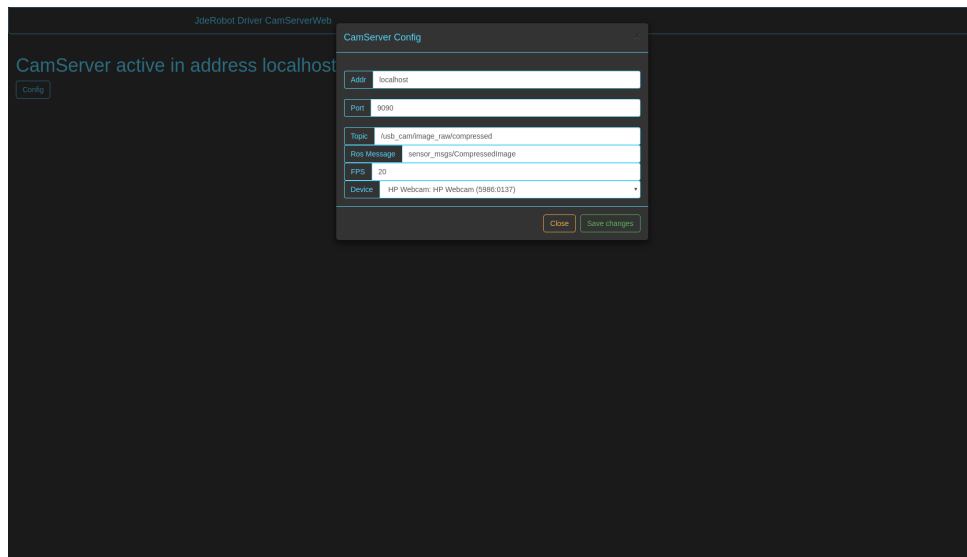


Figura 4.3: Menu de configuración del servidor de imágenes

### 4.3. Adquisición de las imágenes

Para la obtención de las imágenes, como se ha indicado anteriormente, se hace uso del proyecto de código abierto webRTC, que nos permite la transmisión en tiempo real de audio, video y datos. Mediante webRTC, obtenemos las imágenes de una cámara utilizando muy pocas líneas de código, lo que nos facilita enormemente el trabajo. El único problema que se ha tenido que tener en cuenta y 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 se debe realizar es recopilar todos los dispositivos conectados al ordenador y separar los dispositivos de video, que son los que realmente nos interesan. Para lograrlo, se utiliza el api Navigator de webRTC, proporcionándonos el objeto `navigator.mediaDevices`, al cual si le añadimos `.enumerateDevices().then(function (devices){}`, obtenemos todos los dispositivos multimedia conectados al ordenador donde se esta ejecutando. Una vez que ya tenemos todos los dispositivos, simplemente nos interesan como se ha indicado anteriormente, aquellos dispositivos capturadores de video, es decir serán aquellos dispositivos cuyo tipo es de entrada de video (en nuestro código sería un condicional `if (devices.kind == "videoinput")`). La lista de dispositivos se mostrara en el menu de configuración, mostrado en la sección anterior, mediante un campo desplegable

como se muestra en la siguiente imagen:

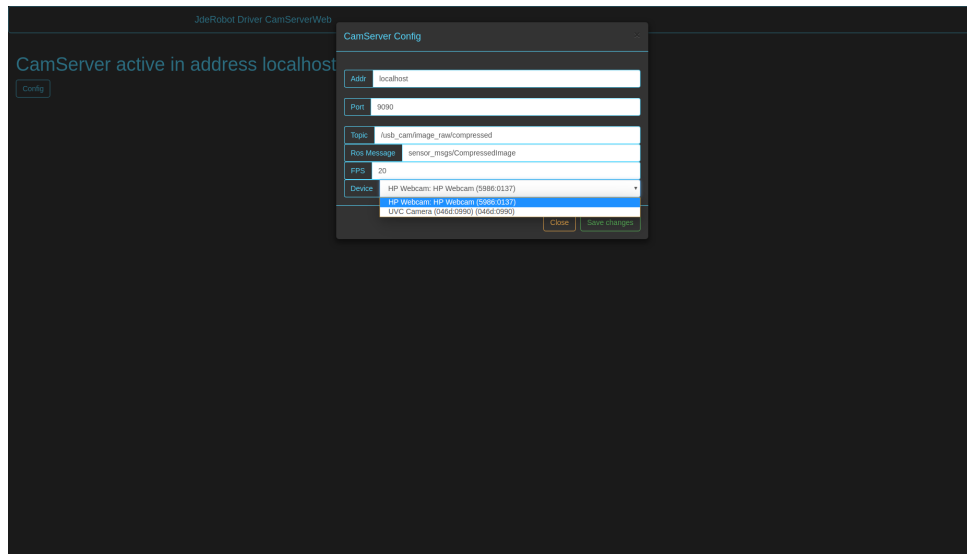


Figura 4.4: Selector del dispositivo de entrada de video

Una vez que seleccionamos el dispositivo que vamos a utilizar para adquirir las imágenes, hay que conectarse a él. Para esta conexión, volvemos 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 esta en el formato `MediaStream`, el cual no es apto para ser enviado o visualizado, por tanto es necesario realizar una reconversión. Para realizar la reconversión, vamos a utilizar el elemento de HTML `canvas` y el método JavaScript asociado a este elemento `toDataURL()`. Este método nos devuelve un data URI (URLs prefijados que permiten a los creadores de contenido incorporar pequeños archivos en linea 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 esto estará contenido en un `canvas` virtual, ya que no se mostrara en ningún lugar y únicamente se utilizara como pasarela entre el API de webRTC y el envío de las imágenes.

## 4.4. Conexión y transmisión de las imágenes mediante ROS

En esta sección se explicara cómo realizar la conexión a ROS y el posterior envío de imágenes mediante un Publisher de ROS. Como se indica en los capítulos anteriores, dado la limitación y la política de privacidad las tecnologías web, es necesario la existencia de un servidor intermedio que estará escuchando en una dirección IP y en un Puerto determinado, y hará las funciones de control de acceso y encaminamiento.

### 4.4.1. Establecer conexión con ROS

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

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

En este código, indicaremos que la conexión se hará utilizando un canal de comunicación WebSocket, y la IP y puerto por la que se transmitirá. De esta forma ya habremos establecido la conexión ROS, pero aún deberemos definir el tipo de mensaje a enviar y a través de que etiqueta de ROS (topic de ROS) nos pueden localizar los clientes que deseen obtener lo que nuestro servidor de imágenes esta transmitiendo.

### 4.4.2. Definición de los mensajes ROS

El tipo de mensaje que vamos a utilizar para el envío sera el tipo predefinido en el API de ROS, `"sensor_msgs/CompressedImage"`. El motivo de la elección de este tipo es debido a 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 será de tipo "sensor\_msgs/CameraInfo" y transmitirá toda la información sobre la cámara (frames por segundo, 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 objeto ROSLIB.Ros generado para realizar la conexión, el nombre del topic y el tipo de mensaje, por lo que generaremos nuestro Publisher para servir las imágenes (el Publisher para la información de la cámara será similar) mediante el siguiente código:

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

#### 4.4.3. Creación de los mensajes y publicación

Definidos los mensajes y establecida la conexión, el siguiente paso es transmitir los mensajes. Para ello se deben crear los mensajes con la información que deseamos transmitir con el Publisher, utilizando de nuevo un objeto definido en la biblioteca roslibjs, new ROSLIB.Message. Para crear este objeto, debemos pasar por parámetros el contenido que queramos que tenga nuestro mensaje, siempre cumpliendo las especificaciones definidas para cada tipo de mensaje escogidos anteriormente y que están definidas en la documentación de ROS <sup>1</sup>. En nuestro caso, para definir los dos mensajes que transmitiremos lo haremos mediante el siguiente código para el caso de la transmisión de las imágenes:

```
var videomensaje = new ROSLIB.Message({
  format : "jpeg",
  data : data.replace("data:image/jpeg;base64,", "")
})
```

De el anterior código cabe destacar que data corresponde a los datos obtenidos mediante

---

<sup>1</sup>[http://wiki.ros.org/common\\_msgs](http://wiki.ros.org/common_msgs)

el método `toDataURL` indicado anteriormente, reemplazando la cabecera donde se indica el formato, ya que el formato ya se indica en el propio mensaje mediante `format`. Para el mensaje donde se transmite la información de la cámara usaremos el siguiente código:

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

Creados los mensajes, solo nos falta publicarlos, lo que se consigue de una manera sencilla mediante

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

Se puede apreciar fácilmente que lo que estamos haciendo 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 estamos enviando un mensaje de cada. 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.

## 4.5. Adaptación para su funcionamiento con Electron

Hacer funcionar el servidor de imágenes con Electron es muy simple, únicamente deberemos generar un archivo JavaScript, que será el encargado de generar la ventana y mostrar el fichero HTML que contendrá la interfaz gráfica, y crear el archivo `package.json`.

### 4.5.1. package.json

Como se indicó en capítulos anteriores, Electron funciona bajo Node.js y su gestor de paquetes, npm, este hecho, nos obliga a que tengamos que generar este archivo `package.json`. El cometido del mismo será definir las dependencias de paquetes de terceros que tendrá nuestro driver, definir las características de nuestro driver (nombre, versión,

autor, etc.) y, lo que es más importante, de que manera se lanzara nuestra driver y el archivo que se debe ejecutar al lanzarla. En el caso del servidor de imágenes, la dependencia será con Electron (tiene más dependencias pero no son vitales para su ejecución), el archivo principal, al que se ha nombrado como "main.js", será el encargado de generar la ventana, tal y como hemos indicado anteriormente, y se lanzara a través del framework Electron. Por lo tanto, el package.json quedará como sigue:

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

### 4.5.2. main.js

El archivo "main.js" indicado anteriormente, contendrá muy pocas líneas de código al tratarse de un elemento totalmente externo a nuestra driver y no tener ningún papel en el funcionamiento de el driver, únicamente es necesario para ejecutar el driver utilizando Electron. A continuación se muestra el código 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 realiza es definir los módulos que se requieren para que funcione correctamente y que se utilizaran en el resto del código. Posteriormente, se define el tamaño de la ventana, en este caso la ventana sera de 1800 pixeles de ancho y 1000 de alto. Mediante `win.loadURL` se simula el funcionamiento de un navegador y como gestionan las diferentes URLs, este método se le pasara como parámetros el archivo HTML principal de nuestra driver, en otras palabras, el archivo HTML que activa el funcionamiento de el driver al contener la lógica de la misma, 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, protocol tomaría el valor "http:"). Finalmente, la última de código llamara a la función que crea la ventana y muestra el HTML en el momento que Electron haya terminado de inicializarse y esta preparado para la creación de la ventana.

## 4.6. Ejecución del servidor de imagenes

Como hemos visto anteriormente, el driver puede ejecutarse a través de dos vías: Electron y Node.js. Sin embargo, 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 explicado en capítulos anteriores.

```
roslaunch rosbridge_server rosbridge_websocket.launch
```

El segundo terminal será donde lanzamos el driver, por lo que tenemos dos formas de hacerlo:

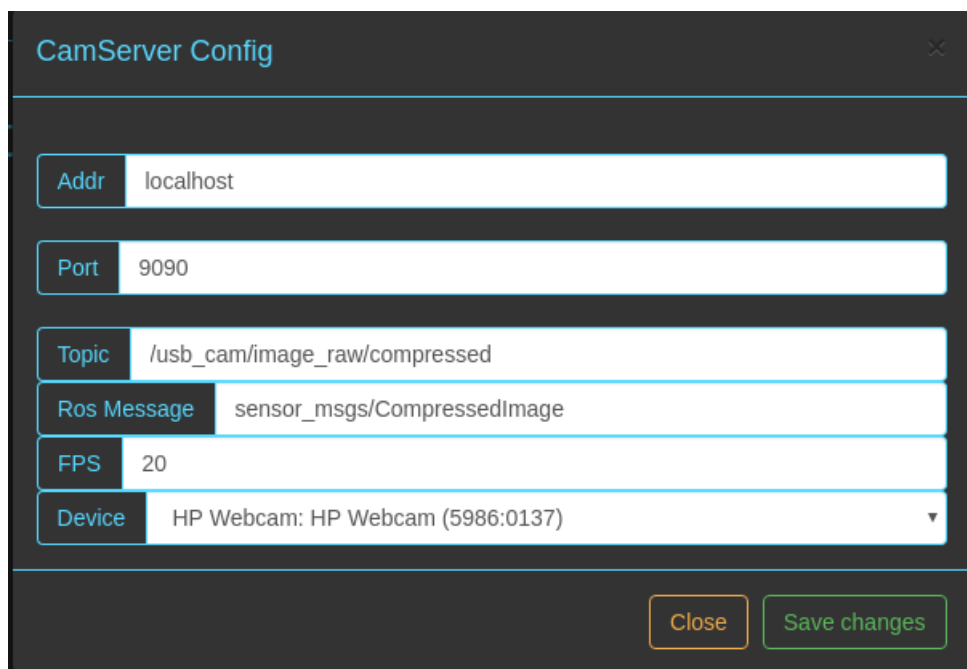
- Con Node.js

```
node run.js  
Arrancamos nuestro navegador favorito y ponemos la url http://localhost:7777/
```

- Con Electron

```
npm install  
npm start
```

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.



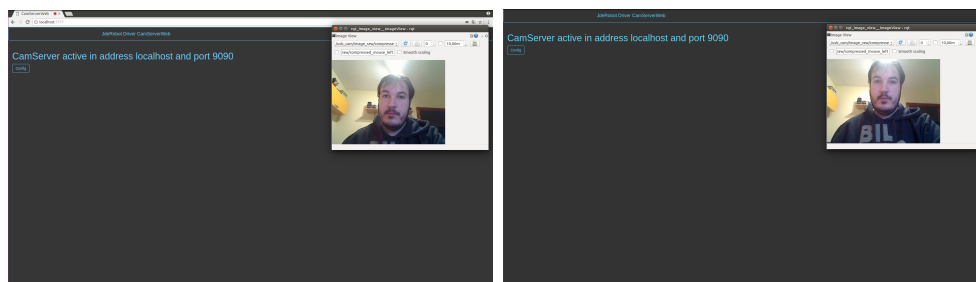
The image shows a 'CamServer Config' window with a dark background and light blue text. It contains several input fields with labels on the left and values on the right:

- Addr:** localhost
- Port:** 9090
- Topic:** /usb\_cam/image\_raw/compressed
- Ros Message:** sensor\_msgs/CompressedImage
- FPS:** 20
- Device:** HP Webcam: HP Webcam (5986:0137) (with a dropdown arrow)

At the bottom right, there are two buttons: 'Close' (orange border) and 'Save changes' (green border).

Figura 4.5: Ejemplo de configuración del driver

Finalmente, para verificar que esta 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.



(a) Node.js

(b) Electron

Figura 4.6: camServerWeb ejecutado con Node.js y con Electron

# Capítulo 5

## Visor web de componentes 3D

En este capítulo se va tratar la creación de un nuevo visor de elementos 3D (modelos, puntos y rectas) con tecnologías web. Este visor, al que se ha nombrado como 3DVizWeb, nos abrirá la posibilidad de obtener datos por una cámara o sensor y mostrar una representación 3D de los mismos.

### 5.1. Introducción y Estructura

Este visor está elaborado usando JavaScript y HTML como lenguajes de programación, y se usará el middleware ICE para la interconexión con los clientes que nos envíaran los datos a mostrar. La imagen 5.1 muestra un esquema de su funcionamiento:

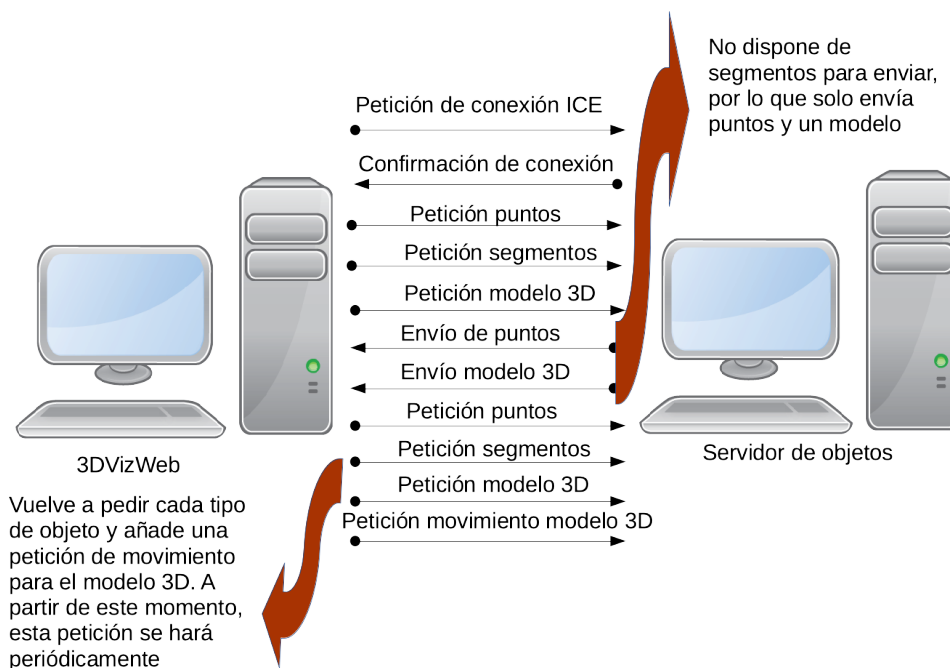


Figura 5.1: Esquema de funcionamiento del visor 3D

Cómo se puede apreciar en el esquema, se trata de un modelo Cliente-Servidor. La función del Cliente la hará el visor 3D y el Servidor enviara, o servirá, los objetos 3D mediante diferentes formatos, dependiendo de si lo que se está enviando son puntos, rectas o modelos 3D. Al realizar la función de cliente el visor llevara la iniciativa en la conexión, es decir deberá ser el encargado de solicitar al servidor si tiene algo que mostrar cada cierto tiempo. Este periodo de tiempo será configurable mediante un fichero de configuración que se explicara en esta sección. Una vez recibido el objeto, el visor mostrara lo que recibe, manteniendo, eliminándolo o moviendo lo que ya mostraba (si se trata de un modelo 3D), dependiendo de lo que indique el Servidor. Debido a las limitaciones que tiene ICE para JavaScript, se ha decidido seguir este modelo, pese a los inconvenientes que presenta: únicamente puede conectarse un servidor al mismo tiempo, la iniciativa la lleva el visor 3D por lo que estará solicitando actualizaciones al Servidor continuamente, tiene un mayor retardo, etc. En próximas versiones se tratara de corregir estos inconvenientes.

### 5.1.1. Estructura del mensaje para visualizar los puntos

Para representar un punto, como veremos en la siguiente sección, únicamente necesitamos una posición en el eje de coordenadas y el color que tendrá el punto. Teniendo en cuenta esto, la estructura del mensaje sería 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)

Sin embargo, como se ha explicado anteriormente, el visor tiene la iniciativa 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 será una posición en el eje de coordenadas y el color, sino una colección variable de estos elementos (podrá ser uno o más). Finalmente, para dar la posibilidad al servidor de indicar si desea añadir o eliminar lo que se muestra en ese momento en el visor (refrescar la escena que muestra el visor), al mensaje se le añade un booleano que valga True, si se desea eliminar lo que se muestra en ese momento y únicamente que se visualice lo que se transmite en ese mensaje, o False, si lo que se desea es añadir lo transmitido en este mensaje a lo que ya se muestra. Por tanto nuestro mensaje quedará 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

### 5.1.2. Estructura del mensaje para visualizar segmentos

Como se vera en la siguiente sección, 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, la estructura del mensaje para representar un segmento será la siguiente:

- Punto de inicio
  - Coordenada “X”
  - Coordenada “Y”
  - Coordenada “Z”
- Punto de fin
  - 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)

Sin embargo, al igual que para el caso de los puntos, se desea poder enviar varios segmentos. El mensaje, por tanto, contendrá un buffer de segmentos cuya estructura será una colección de dos puntos, con su posición en el eje de coordenadas, y el color de cada segmento. Finalmente, también se incluye el booleano para indicar si se debe refrescar el visor o por el contrario simplemente se deben añadir a lo que ya se muestra. La estructura final del mensaje de envío de segmentos es la siguiente:

- Buffer de segmentos
  - Punto de inicio
    - Coordenada “X”
    - Coordenada “Y”
    - Coordenada “Z”
  - Punto de fin
    - 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

### 5.1.3. Estructura del mensaje para visualizar un modelo 3D

En este caso, primero hay que indicar que el mensaje de petición es algo diferente a los casos anteriores. Mientras que para los segmentos y para los puntos únicamente se hace la petición sin añadir ningún parámetro a la misma (la forma de realizar la petición se detallara en secciones posteriores), en el caso de las peticiones de un modelo 3D desde el visor 3D al servidor, se le debe incluir el identificador que se le va a dar al modelo (en caso de que el servidor tenga uno para enviar), de modo que tanto visor como servidor puedan relacionar las peticiones de movimiento o borrado de un modelo mediante este identificador. Una vez explicado esto, ya podemos ver cómo es la estructura del mensaje para visualizar un modelo 3D. Para mostrar un modelo, se necesita conocer el archivo que contiene el modelo, el formato del archivo, la posición en el eje de coordenadas, la orientación del modelo y la escala del modelo.

El archivo puede ser enviado de dos formas, la primera forma es mediante una url a un servidor externo o página web, la segunda forma es enviar el fichero como texto plano, sin embargo 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.

El formato del archivo, como se explicara en próximas secciones, puede ser del tipo “dae” o “obj”. Estos dos formatos son los mas utilizados a la hora de crear modelos 3D y por tanto se considera que dando soporte a los dos formatos permite usar cualquier modelo que se desee.

La escala del modelo será un número decimal que indicara el tamaño con el que se desea mostrar el modelo, permitiendo así mostrar varias veces el mismo modelo 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 usara la clase “Pose3D”. Esta clase esta formada por las coordenadas “X”, “Y” y “Z”, una componente “h” que no se utilizara, y los cuaterniones “q0”, “q1”, “q2” y “q3” para después usando las formulas matemáticas correspondientes y que se explicaran más adelante, nos proporciona la orientación del objeto.

Por tanto, el formato del mensaje tendrá la siguiente estructura:

- Archivo con el modelo 3D
- Formato del archivo
- Pose3D
  - Coordenada “X”
  - Coordenada “Y”
  - Coordenada “Z”
  - “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 transmitido por el visor, y se añade también el refresco al igual que para los puntos y los segmentos. Sin embargo, debido a las limitaciones de tamaño de los archivos que se pueden enviar mediante ICE, y al necesidad de establecer un identificador común, solo es posible enviar un modelo 3D por petición realizada y no un buffer de modelos, como si podemos hacer con los puntos y los segmentos.

### 5.1.4. Estructura del mensaje para mover los modelos 3D

Como se explicara en secciones sucesivas, los modelos 3D que se visualizan probablemente se deseen moverlos de posición u orientación y borrarlos y crearlos de nuevo implica un mayor retardo y carga de trabajo, por lo que ha sido necesario incorporar una secuencia de petición y envío de movimiento, pero solo se comenzará a realiza la petición de movimiento una vez se muestre un modelo ya en el visor. La estructura de este mensaje será una versión reducida del mensaje para crear un modelo. En este caso únicamente se deberá enviar la nueva posición mediante la clase “Pose3D” y el identificador del modelo que se desea mover. Por tanto, la estructura quedara de la siguiente manera:

- Pose3D
  - Coordenada “X”
  - Coordenada “Y”
  - Coordenada “Z”
  - “h”
  - Cuaternión “q0”
  - Cuaternión “q1”
  - Cuaternión “q2”
  - Cuaternión “q3”
- Identificador

En este caso, se vuelve a poder enviar una secuencia de movimientos para uno o más modelos, 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 de manera que si entre petición y petición, se ha movido un modelo varias veces o se han movido varios modelos que se muestran en el visor, se pueda realizar esos movimientos al mismo tiempo en el visor, evitando largos retardos que provocarían enviar los movimientos de uno en uno. Sin embargo, como es evidente, en esta ocasión el refresco no se envíe al no tener sentido eliminar todo lo que se muestra en la escena, sí estamos enviando la actualización de algo que se muestra en la escena. Teniendo en cuenta esto, la estructura final del mensaje es la siguiente:

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

### 5.1.5. Fichero de configuración

En este apartado se va a explicar el método utilizado para configurar el visor 3D. Dado que el visor en su interfaz gráfica únicamente muestra la escena donde se mostrara los objetos como se explicara en la siguiente sección, 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. Se ha decidido usar un archivo externo con formato “YAML” que será leído y cargada la información

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, somos capaces de cargar y guardar en una variable la información que contiene un archivo de este tipo. Para cargarlo en el visor, vamos a usar el siguiente código:

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

Con este código, habríamos cargado el contenido del archivo “config.yml” (“.yml” es la extensión de “YAML”) en la variable “config”.

El archivo “config.yml” tendrá el siguiente contenido:

```
Server: "localhost" #IP local
Port: "11000" #Puerto de escucha del servidor (puede ser cualquiera)
updatePoints: 1000 #miliseconds
updateSegments: 1000 #miliseconds
updateModels: 1 #miliseconds
updateMove: 1000 #miliseconds
linewidth: 2 #width of the line
pointsize: 8 #size of the point
camera: #camera position
  x: 50
  y: 20
  z: 100
```

Como se puede apreciar, el fichero contiene todos los parámetros configurables del visor.

## 5.2. 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 será íntegramente el visor (no tendrá ningún elemento adicional), el cual mostrara inicialmente una rejilla que hará de plano horizontal, cuyo centro será la posición (0, 0, 0) del eje de coordenadas. Además es necesario añadir iluminación y una cámara para poder visualizar la escena.

### 5.2.1. Escena base

El visor esta 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 perspectiva utilizando `THREE.PerspectiveCamera()` ubicándola en una posición predeterminada (la indica en el archivo de configuración anteriormente indicado) 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()`, nos permite definirla muy fácilmente introduciendo por parámetro a la función la separación, cantidad y color de las líneas de división que generan nuestra rejilla. Para añadirla a la escena, bastara con usar `.add(rejilla)`. Estos elementos formaran la escena base del visor.

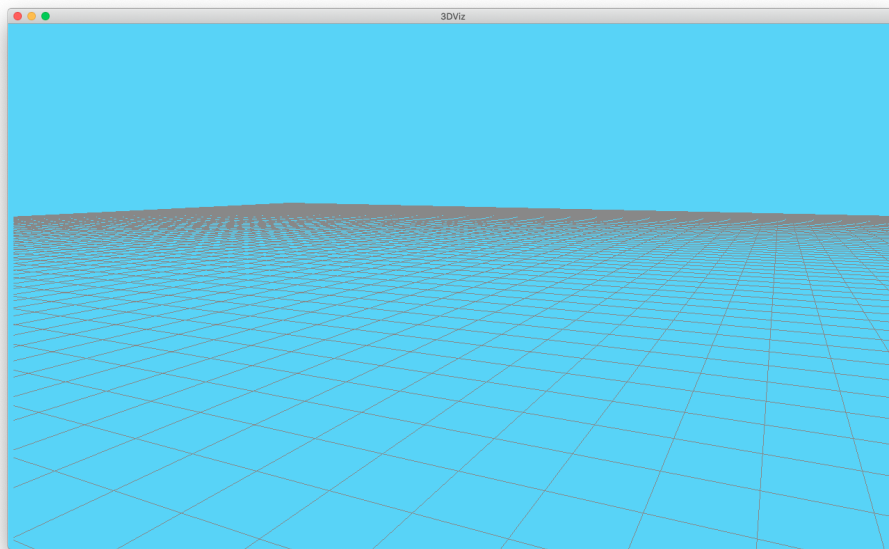


Figura 5.2: Interfaz gráfica base del visor

### 5.2.2. Visualizar puntos

Un punto es el elemento geométrico más simple que es posible representar. Sin embargo, pese a ello, proporciona a nuestro visor la capacidad de realizar representaciones más complejas mediante el uso de una gran cantidad de puntos. Para representar puntos, únicamente es necesarios una posición en el eje de coordenadas, el tamaño del punto y el material o color que queramos darle. La biblioteca Three.js nos proporciona una serie de elementos para facilitar la creación de puntos. A continuación se muestra el código para crear y mostrar un punto:

```
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 );  
}
```

Primero es necesario indicar que estamos 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 indicaremos que serán vertices pero únicamente se proporciona uno de ellos mediante un vector x, y z, el cual serán las coordenadas centrales del punto que queremos representar. Cabe destacar que dado que el sistema de coordenadas de obtención de los datos no se corresponde con el sistema de coordenadas de nuestra escena, hay que realizar una conversión de modo que 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, posteriormente, su color. El tamaño indicado en el código es de 8 pixels, sin embargo este tamaño es configurable a través de un fichero de configuración indicado en la sección anterior. El color del punto se definirá mediante sus componentes RGB que como se ha indicado anteriormente, vienen indicadas por el servidor que nos transmite el punto. Las componentes vendrán separadas en R, G y B, y su valor oscilara 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 creado, le atribuimos el nombre “punto” para poder borrarlo, si así lo deseamos, y lo añadimos a la escena. Esta función será invocada cada vez que se reciba uno o más puntos procedentes del servidor. La siguiente imagen muestra varios puntos en el visor:

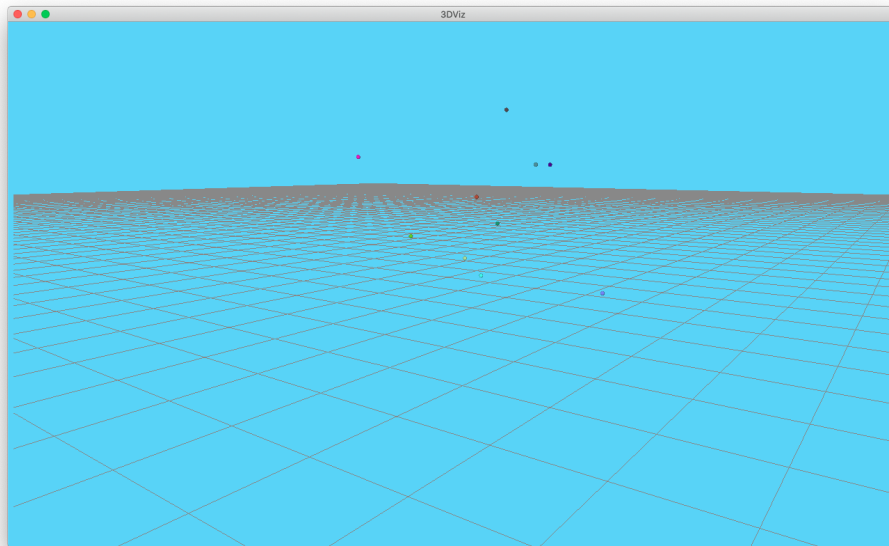


Figura 5.3: Varios puntos visualizados en el visor

### 5.2.3. Visualizar segmentos

El segmento es otro de los elementos fundamentales de la geometría y se puede definir cómo un fragmento de recta que esta comprendido entre dos puntos. Teniendo en cuenta esto, para poder representarlo únicamente es necesario las coordenadas de dos puntos, el grosor del segmento y el color del mismo. A continuación se muestra el código para la creación de una línea:

```
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);  
}
```

Es fácil de apreciar que la estructura es muy similar que la función para crear un punto. Al igual que para un punto, necesitamos definir la geometría pero, como se ha indicado anteriormente, en este caso vamos a tener dos vertices en lugar de uno (dos puntos), el primer vértice será desde el lugar donde comience la recta, y el segundo vértice, el lugar donde termine, en otras palabras, la recta será la unión entre el primer vértice con el segundo vértice. Las coordenadas de estos dos puntos será proporcionadas por el servidor y, 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, definimos el material y aspecto del segmento proporcionando el grosor y el color. En el código anterior, el grosor es de 2 pixeles, sin embargo este parámetro será configurable mediante el fichero de configuración explicado anteriormente. El color, por el contrario, vendrá definido por parte del servidor mediante sus componentes RGB, que al igual que para el caso del punto, vendrán sementadas 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 creado, dandole el nombre “line” para poder borrar únicamente los segmentos, cómo se ya se ha explicado anteriormente. Una vez generado el elemento, se añade a la escena. Esta función sera la invocada cada vez que se reciba uno o más segmentos desde el servidor. La siguiente imagen muestra varios segmentos en el visor:



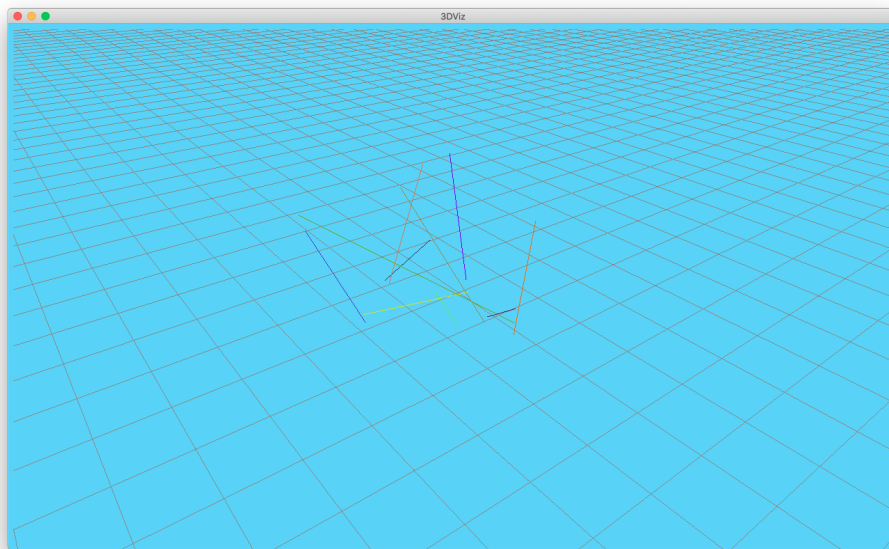


Figura 5.4: Varios segmentos visualizados en el visor

#### 5.2.4. Visualizar modelos 3D

Un modelo 3D es una representación matemática de un objeto tridimensional y que se puede guardar en un archivo de definición de geometría. El visor será capaz de representar dos formatos diferentes de estos archivos, “obj” y “dae” y moverlos cuando el servidor lo quiera.

#### 5.2.5. Creación y visualización de los modelos 3D

Los modelos 3D podrán ser recibidos o bien a través del archivo completo como texto plano, o bien a través de una url. El visor será capaz de representar el modelo 3D y ubicarlo en la posición indicada. El siguiente código muestra como identificar si el modelo viene como texto plano o url, para posteriormente invocar a la función correspondiente dependiendo el formato del modelo:

```
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);
    }
}
```

Cuando recibimos el modelo procedente del servidor, lo primero que hacemos es truncar el objeto para poder analizarlo y poder esclarecer si lo que nos está enviando el servidor es el archivo completo o una url al archivo completo. Dado que se trata de una url, debe contener “https://” o “http://”, podemos truncar mediante “.” y, con una sentencia condicional verificar si la primera parte de la separación es “https”, “http” u otra cosa. En caso de ser “https” o “http”, lo que hemos recibido es la ruta al archivo, sino lo que se recibe es el modelo 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 lo hemos recibido como texto plano y para poder visualizarlo necesitamos que el modelo sea un archivo y tener una ruta al mismo. Como se puede ver, por ambos caminos tenemos una url, de modo que ya podemos representar el modelo del mismo modo.

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

#### ■ Representar modelos del tipo obj

Un archivo “obj” es conocido como Wavefront 3D Object File y desarrollado por Wavefront Technologies. Es un formato de archivo 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 nos proporciona un API para poder cargar y mostrar un modelo de este tipo a través de una url. El

siguiente código carga y muestra un modelo “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);
        });
}
```

El código anterior lo primero que hace es inicializar el API de Three.js para cargar un modelo “obj” y se llama a la función del API encargada de cargar el modelo. Esta función tiene como parámetros la url y otras tres funciones más. La primera función es la encargada de cargar el modelo y mostrar el mismo en el visor, la segunda función se ejecuta mientras se está cargando el modelo (por ejemplo, para mostrar una barra de progreso) y la última función se ejecutara 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, y que se explicara en siguientes secciones, para poder borrarlo o moverlo posteriormente, se posiciona el modelo mediante las coordenadas enviadas por el servidor, teniendo en cuenta el problema explicado en las secciones anteriores acerca de la disparidad entre entre los sistemas de coordenadas, y se orienta mediante la rotación correspondiente a cada eje, que nos es enviada en radianes y se debe convertir a grados. Finalmente, se añade el modelo al visor.

### ■ Representar modelos del tipo dae

Un archivo “dae”, también conocido como archivo “Collada”, es un formato de intercambio 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 siguiente código carga y muestra un modelo “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*toDegrees, pose3d.rz * toDegrees,
                               pose3d.ry * toDegrees);
            scene.add( object );
        },
        function (xhr){
        },
        function (error){
            console.log(error);
        });
}
```

Cómo se puede apreciar, el código es igual que el utilizado para el formato “obj”, únicamente cambia la inicialización del API que en este caso es el API para cargar un archivo “dae” o “Collada” y la obtención del modelo, 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 serán invocadas cada vez que se reciba un modelo nuevo que mostrar. Indicar que si bien en esta memoria se muestra dos funciones diferentes para cargar un modelo “obj” o “dae” de modo que quede lo más claro posible, en el código final se agrupa en una única función, utilizando en ella la sentencia condicional para identificar el tipo de

formato del modelo, evitándonos el uso de código repetido. La siguiente imagen muestra dos modelos (cada uno de un formato) en el visor

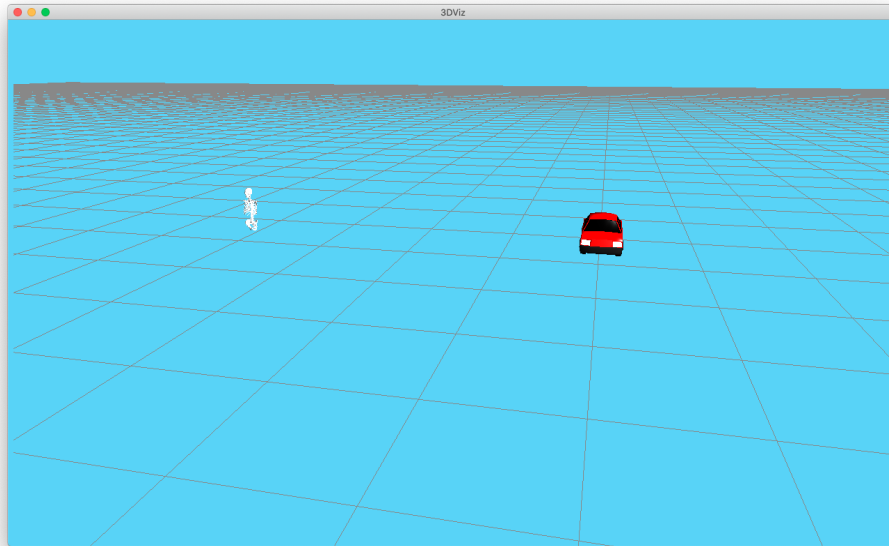


Figura 5.5: Modelos 3D mostrados en el visor

### 5.2.6. Mover los modelos 3D

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

Al haber identificado el modelo mediante un id único ( en el caso del punto y la recta, el id es el mismo para cada elemento), nos permite realizar el movimiento. Esto se realiza mediante la siguiente secuencia de código:

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

Como se puede apreciar, mover el modelo es mucho más rápido y sencillo que eliminarlo y crearlo de nuevo. Primero seleccionamos el modelo mediante el identificador, posteriormente reubicamos el modelo seleccionado a las nuevas coordenadas y le aplicamos la nueva orientación. La imagen que se muestra a continuación corresponde a los modelos anteriores desplazados y rotados:

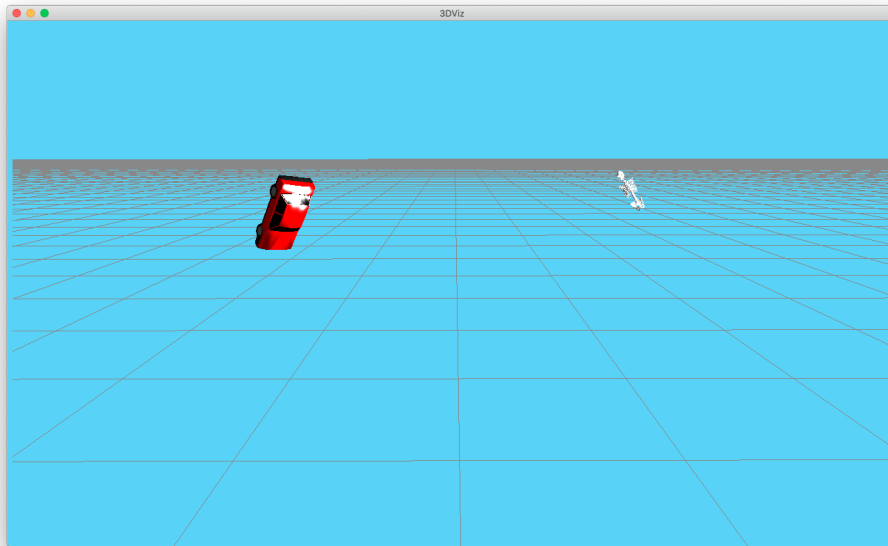


Figura 5.6: Modelos 3D reubicados y rotados

### 5.2.7. Borrar elementos mostrados en el visor

Como ya se ha indicado anteriormente, el visor ofrece la posibilidad de eliminar todos los elementos que se están mostrando en ese momento en él, si así lo ha indicado el servidor cuando envía un nuevo elemento para mostrar. Eliminar todos los elementos de la escena es simple, el siguiente código elimina todos los elementos que hubiese en la escena:

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

```
}  
}
```

En este código lo que se hace es recorrer el array con los identificadores de todos los objetos que hay en el visor y que se han ido añadiendo en el momento de incluir al visor los modelos (dado que los puntos y los segmentos tienen el mismo id, son añadidos a la lista de identificadores cuando se reciben los primeros elementos de cada tipo). Se selecciona el objeto en el visor que coincida con el identificador de la lista. Posteriormente se realiza otro bucle para eliminar todos los elementos del visor con ese identificador (esto se realiza para los segmentos y los puntos, que como ya se ha indicado, todos tienen el mismo).

### 5.3. Conexión con el servidor y recepción de los objetos 3D

En esta sección se va a tratar cómo se realiza la conexión y la posterior recepción de cada uno de los objetos y modelos 3D anteriormente indicados. La conexión se realizará mediante el middleware ICE, explicado en capítulos anteriores, y su lenguaje de especificación Slice nombrando a los ficheros creados con este lenguaje interfaces slice. Lo primero que se detallará en este capítulo serán estas interfaces, ya que son la base de la conexión y que tanto el servidor como el visor deben compartir. Si alguno de las dos partes no usan la misma interfaz, la conexión no podrá llevarse a cabo correctamente. Después, se detallará cómo se realiza la conexión con el servidor y la posterior petición de cada tipo de objeto. Finalmente, se explicará cómo se realiza la recepción de cada objeto y el tratamiento que recibe cada uno para su posterior visualización en el visor mediante las funciones indicadas en la anterior sección.

#### 5.3.1. Interfaces Slice

Las interfaces Slice podrían 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 compilan usando el compilador correspondiente al lenguaje de programación correspondiente. 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.

En la plataforma JdeRobot hay una gran cantidad de interfaces slice creadas para la interconexión de varios drivers y aplicaciones ya existentes, por lo que se ha realizado en este trabajo es ampliar varias de esas interfaces y utilizar otras ya creadas, en concreto se han utilizado las interfaces “primitives.ice”, “pose3d.ice” y “visualization.ice”. Estas interfaces contienen la definición de la estructura de los mensajes indicado anteriormente en este capítulo y las funciones que se utilizaran para el intercambio de los mismos.

#### 5.3.1.1. primitives.ice

Esta interface contiene las estructura necesarias para el intercambio de puntos y segmentos. A continuación se muestra como se define estas estructuras:

```
#ifndef PRIMITIVES_ICE
#define PRIMITIVES_ICE

module jderobot{
    struct RGBPoint{
        float x;
        float y;
        float z;
        float r;
        float g;
        float b;
    };
    struct Point{
        float x;
        float y;
        float z;
    };
    struct Segment{
        Point fromPoint;
        Point toPoint;
    };
};
```



```
};  
#endif
```

Esta interface contiene tres estructuras diferentes. La primera estructura corresponde al formato del punto que se utilizara para mostrarlos en el visor, ya que como se puede ver contiene las coordenadas para colocar el punto y las componentes RGB para indicar el color. La segunda estructura corresponde también a un punto pero únicamente contiene las coordenadas, esta estructura es utilizada para definir la tercera estructura de la interface que es el segmento, ya que, como se ha indicado, esta formada por dos puntos.

#### 5.3.1.2. pose3d.ice

Esta interfaz es la que se utiliza para definir la ubicación y el movimiento de los modelos 3D, utilizando Pose3D. A continuación se muestra el contenido de esta interface:

```
#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
```

Como se puede ver, en este caso en lugar de usar “struct” como en el caso anterior, se usa

el tipo “Class” de Slice ya que esta interface se usa en otros componentes de la plataforma JdeRobot donde es necesario que sea así (ofrece una serie de posibilidades que no ofrece el tipo “struct”). En términos de este trabajo, se manejarán los dos tipos por igual.

#### 5.3.1.3. visualization.ice

Esta interface será la clave del visor, ya que contendrá las estructuras finales que se usarán en el visor y las funciones que facilitaran el intercambio de mensajes entre el visor y el servidor. A continuación se muestra el contenido de este archivo:

```
#ifndef VISUALIZATION_ICE
#define VISUALIZATION_ICE

#include <primitives.ice>
#include <pose3d.ice>

module jderobot{

    struct Color{
        float r;
        float g;
        float b;
    };

    struct RGBSegment{
        Segment seg;
        Color c;
    };

    sequence<RGBSegment> Segments;
    sequence<RGBPoint> Points;

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

```
struct bufferPoints{
    Points buffer;
    bool refresh;
};

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

struct PoseObj3D{
    string id;
    Pose3DData pos;
};

sequence<PoseObj3D> bufferPoseObj3D;

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

Lo primero que se define en este archivo es una estructura con las componentes de color RGB para extender la estructura de los segmentos dando lugar a una nueva. A continuación se definen los arrays para crear los buffers para enviar enviar varios puntos

y segmentos en lugar de uno de cada tipo, para posteriormente definir las estructuras definitivas que se utilizaran para el envío y recepción de los puntos y segmentos y, que como se explico en el momento de describir el formato de los mensajes, contendrá los buffers y un booleano para indicar el refresco o no del visor. Las dos siguientes estructuras corresponden a la definición de los mensajes del modelo 3D y del movimiento del mismo, utilizando la clase Pose3DData creada en la interface anterior y los parámetros explicados en secciones anteriores (el archivo con el modelo, id, formato, escala, etc.). En el caso de la estructura del mensaje del movimiento, al enviarse más de un movimiento por mensaje, se crea una secuencia para definir el buffer que los contendrá.

Esta interface, a diferencia de los dos anteriores, contiene la definición de las cuatro funciones que serán las encargadas de realizar las peticiones y respuestas para cada tipo de elemento. Estas funciones son realmente la interface. En el momento de crear la conexión, como se vera más tarde, esta interfaz sera invocada por parte del visor y se conectara con el servidor para verificar que ambos están usando la misma, de no ser así, la conexión no se realizará.

### 5.3.2. Conexión y peticiones al servidor

Lo primero que hay que indicar es que para la conexión e intercambios con el servidor, el visor hace uso de los Web Workers de HTML5 <sup>1</sup> que permite realizar ejecuciones 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.

Una vez que el visor se ha lanzado y se ha terminado de cargar, se crear el Web Worker y se envía el mensaje para que se establezca la conexión, esto se realiza utilizando el siguiente código:

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

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 poder realizar

---

<sup>1</sup>[https://www.w3schools.com/html/html5\\_webworkers.asp](https://www.w3schools.com/html/html5_webworkers.asp)

la conexión. También se inicializan las variables para posteriormente lanzar el Promise (permite manejar la naturaleza asincrónica de ICE) y la variable donde se guardará el proxy con la conexión realizada con el servidor. Esto se realiza mediante las siguientes sentencias:

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

Una vez inicializada 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 que proxy se está intentando conectar el cliente) y el endpoint. El proxy que nos 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 lo que hace es preguntar al servidor si el proxy que nos ha devuelto es un proxy para el objeto de la interfaz de “visualization.ice”, si lo es nos devuelve un proxy del tipo “jderobot.VisualizationPrx” que guardamos en la variable creada para almacenar la conexión, sino devolverá un error. Finalmente, el Worker manda un mensaje al hilo principal indicando que la conexión se ha realizado correctamente. El siguiente código muestra como se realiza todo lo indicado anteriormente:

```
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"});
    });
}
```

```
}
```

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()”<sup>2</sup>, si no lo ha sido elimina el Worker e imprime el mensaje por consola. Esto se realiza de la siguiente manera:

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

Cuando se pase el tiempo, se activa 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 modelos 3D, que como se ha explicado anteriormente, enviara el identificador que se dará al modelo en caso de que haya uno. Este identificador sera una cadena de “obj” y un contador de modelos que hay en la escena, es decir, si no hay modelos en la escena, el siguiente tendrá de identificador “obj1”, el siguiente “obj2” y así sucesivamente. Fianlamente, esta función termina realizando una llamada a la función que se encargara de gestionar las respuestas del servidor. A continuación se muestran estas funciones:

```
function setPoint(){
```

---

<sup>2</sup>[https://www.w3schools.com/jsref/met\\_win\\_setinterval.asp](https://www.w3schools.com/jsref/met_win_setinterval.asp)

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

En el Worker, cuando se reciba los mensajes para que se soliciten 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. A continuación se muestran estas peticiones:

```
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});
    });
}
```

Como se puede ver, solo se esta realizando las peticiones para los puntos, los segmentos y los modelos, pero no para los movimientos de los modelos, ya que solo se realiza una vez que se ha recibido un modelo para mostrar. Una vez que se tiene un modelo, 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.

### 5.3.3. Recepción y tratamiento de los mensajes recibidos

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

En el hilo principal hay un manejador de mensajes que será la función “getData()” que se invoca cada vez que se realiza una petición, como se indico anteriormente. 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 sobre la interfaz gráfica. El siguiente condicional 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") {
            ...
        }
    }
}
```



### 5.3.3.1. Tratar los puntos

Si el mensaje enviado por el Worker es del tipo “drawPoint”, el manejador indicara que se deben ejecutar las sentencias correspondientes a la visualización de los puntos. Una vez concluido que se deben mostrar los puntos, lo primero que se realiza es verificar si el servidor ha solicitado que haya refresco del visor o no, si el servidor ha indicado que se debe refrescar el visor (y si el mensaje trae puntos que mostrar, ya que sino se interpreta el mensaje como erróneo), se llama al método encargado de eliminar los elementos que se muestran en ese momento en el visor y que se describió en la sección sobre la interfaz gráfica. 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 como vimos anteriormente, el método ya se encarga de diferenciar cada uno y realizar las tareas necesarias para mostrarlos. El siguiente código es el encargado de realizar estas funciones:

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

### 5.3.3.2. Tratar los segmentos

El tratamiento de los mensajes que contienen los segmentos es muy similar a los mensajes con los puntos. Se revisa si es necesario refrescar el visor o no, si así lo fuera se elimina los elementos que hay en el visor (si los hubiese), posteriormente se recorre el buffer de segmentos mediante el bucle “for” y finalmente se llama al método para mostrar segmentos en cada iteración. Al igual que en el caso de los puntos, se pasa como parámetro el segmentos completo y ya se encarga el método de separarlo y realizar

las tareas correspondientes. El siguiente código muestra como realizar esto, pudiendose apreciar que es muy similar al caso de los puntos:

```
if (event.data.func == "drawLine"){
    if (event.data.segments.refresh & (event.data.segments.buffer.length !=0)){
        deleteObj();
    }
    segments = event.data.segments.buffer;
    for (var i = 0; i < segments.length; i+=1) {
        addLine(segments[i], "segments");
    }
}
```

#### 5.3.3.3. Tratar los modelos 3D

El tratamiento de los modelos 3D comienza de la misma forma que los otros dos elementos, es decir se revisa si requiere refresco o no del visor. Sin embargo, una vez que se ha borrado o no el contenido del visor, se actualiza el contador que forma 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 modelos sea capaz de entender.

Para realizar esta conversión, lo primero que se realiza es crear una clase de JavaScript cuya estructura contendrá los parámetros que se necesitan para mostrar el modelo, es decir la posición en el eje de coordenadas (coordenadas “x”, “y” y “z”), las orientación en cada eje de coordenadas (orientación “rx”, “ry” y “rz”) y el identificador del modelo 3D. La clase queda por tanto del siguiente modo:

```
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;
    }
}
```

```
}  
}
```

Para realizar la conversión, se han creado tres métodos diferentes que devolverán cada uno de ellos la orientación en cada uno de los ejes de coordenadas. Estos métodos usarán las formulas matemáticas que transforman los cuaterniones en los ángulos de navegación usados para describir la orientación de un objeto tridimensional y que son un tipo de ángulos de Euler <sup>3</sup>. 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 “x”. Estas formulas matemáticas incorporadas a nuestro código origina los siguientes métodos:

```
function getYaw(q0,q1,q2,q3) {  
    var rotateZa0=2.0*(q1*q2 + q0*q3);  
    var rotateZa1=q0*q0 + q1*q1 - q2*q2 - q3*q3;  
    var rotateZ=0.0;  
    if(rotateZa0 != 0.0 && rotateZa1 != 0.0){  
        rotateZ=Math.atan2(rotateZa0,rotateZa1);  
    }  
    return rotateZ;  
}  
  
function getRoll(q0,q1,q2,q3){  
    rotateXa0=2.0*(q2*q3 + q0*q1);  
    rotateXa1=q0*q0 - q1*q1 - q2*q2 + q3*q3;  
    rotateX=0.0;  
  
    if(rotateXa0 != 0.0 && rotateXa1 !=0.0){  
        rotateX=Math.atan2(rotateXa0, rotateXa1);  
    }  
    return rotateX;  
}  
  
function getPitch(q0,q1,q2,q3){  
    rotateYa0=-2.0*(q1*q3 - q0*q2);
```

---

<sup>3</sup>[https://en.wikipedia.org/wiki/Conversion\\_between\\_quaternions\\_and\\_Euler\\_angles](https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles)

```
    rotateY=0.0;
    if(rotateYa0>=1.0){
        rotateY=Math.PI/2.0;
    } else if(rotateYa0<=-1.0){
        rotateY=-Math.PI/2.0
    } else {
        rotateY=Math.asin(rotateYa0)
    }

    return rotateY;
}
```

Finalmente, se crea una función que devuelve un objeto de la nueva clase creada y que tienen ya la conversión. A esta función se le pasa por parámetro el modelo 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 orientación en cada eje convertidas y que están en radianes. Esta función es la siguiente:

```
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;
}
```

Una vez que ya hemos realizado la conversión, se llama al método encargado de cargar los modelos en el visor y que se explico en la sección correspondiente. A este método, se le pasa por parámetro el modelo completo recibido del servidor y el objeto de la nueva clase que incorpora la conversión de la orientación, para que pueda mostrar el modelo en el visor. Tras ello, se arranca la petición periódica de movimientos para el modelo (si no estaba arrancada previamente) al tener ya mínimo un modelo para mover. Las siguientes sentencias hacen que tenga lugar todo lo indicado:

---

```
else if (event.data.func == "drawObj") {
    if (event.data.refresh & (event.data.obj != "")){
        deleteObj();
    }
    cont += 1
    var pos = getPose3D(event.data.obj);
    addObj(event.data.obj,pos);
    if (posInterval == null){
        posInterval = setInterval(function(){
            setPose3D();
        }, config.updatePose3D);
    }
}
```

#### 5.3.3.4. Tratar el movimiento de los modelos

En el caso de los mensajes con el movimiento, al no indicar si hay que refrescar el visor o no, únicamente se recorrerá buffer que contiene todos los movimientos realizando la conversión de Pose3D al formato admitido explicado anteriormente y invocando al método que realiza el movimiento de los modelos explicado en la sección sobre la interfaz gráfica. Cabe indicar que en la clase que se ha creado para la conversión se incluye el identificador del modelo que se va a mover, ya que, a diferencia de cuando invocamos al método que representa el modelo en el visor que se pasaba por parámetro el mensaje completo recibido del servidor y la nueva clase, en este caso solo pasamos por parámetro la nueva clase por lo que es necesario incluir el identificador. El código que se muestra este código:

```
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);}
}
```

## 5.4. Ejecución del visor 3D

El visor esta preparado para ser usado principalmente con Electron, ya que la lectura del fichero de configuración solo esta habilitada para sí se usa con Electro. Se puede utilizar en el navegador, pero la configuración sera la que se indica al visor por defecto, no pudiendo modificarse lo que limita en gran manera las funciones del mismo.

Ejecutarlo con Electron se realiza de la misma manera que para el Servidor de imágenes explicado en el capitulo anterior, por lo que se considera que no es necesario repetir el proceso de adaptación de la aplicación para ser usada con Electron.

Para ejecutar el visor 3D y realizar pruebas, se ha creado un servidor de test escrito en python que enviara todos los tipos de elementos y movimientos, enviando valores aleatorias. En cada mensaje podrá envía entre 1 y 5 puntos y rectas, esta preparado para enviar 3 modelos (uno en forma de texto plano y dos desde una url) y movimientos aleatorios para esos modelos.

1. En un terminal ejecutar el servidor de prueba:

```
python server_test.py
```

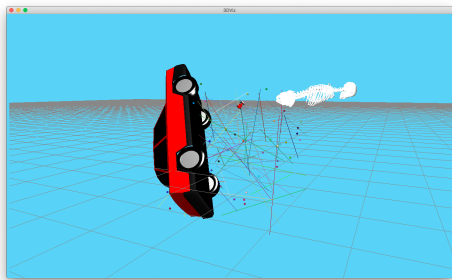
2. En otro terminal, instalar Electron y las dependencias del visor:

```
npm install
```

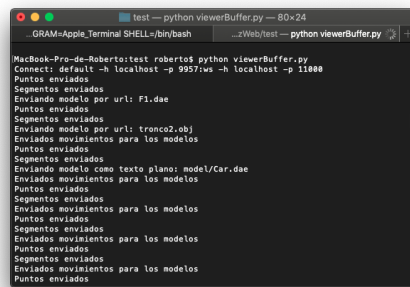
3. En el mismo terminal, ejecutar el visor 3D con Electron:

```
npm start
```

A continuación se muestra la prueba realizada con el servidor:



(a) Visor 3D



(b) Servidor de prueba

Figura 5.7: Ejemplo de ejecución del visor con el servidor de prueba

## Capítulo 6

# Visores web modificados para ser usados con Electron



## Capítulo 7

## Conclusiones