



INGENIERÍA EN SISTEMAS AUDIOVISUALES Y
MULTIMEDIA

Curso Académico 2018/2019

Trabajo Fin de Grado

WEBSIM
SIMULADOR DE ROBOTS CON TECNOLOGÍAS WEB VR

Autor : Álvaro Paniagua Tena

Tutor : Dr. Jose María Cañas Plaza

Co-tutor: Dr. Jesús González Barahona

Trabajo Fin de Grado

Título del Trabajo con Letras Capitales para Sustantivos y Adjetivos

Autor : Álvaro Paniagua Tena

Tutor : Dr. Jose María Cañas Plaza

Co-tutor : Dr. Jesús González Barahona

La defensa del presente Proyecto Fin de Carrera se realizó el día de
de 20XX, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 20XX

*Dedicado a
mis padres, familia, y a mi pareja Cristina*

Agradecimientos

En primer lugar dar las gracias a mis padres por apoyarme y animarme desde el primer momento. También agradecer a mis compañeros Roberto, Ángel y Ahmed por tantas horas de ayuda y clases particulares para que entendiese todo bien.

Gracias también a Jose María Cañas por darme la oportunidad de colaborar en el proyecto y a Jesús González Barahona por su ayuda para el desarrollo y mejora del proyecto.

Por último dar las gracias a mi pareja Cristina, has sido un gran apoyo en estos últimos años y sin duda me has motivado a hacer mejor las cosas y superarme.

Resumen

Este proyecto se encuadra dentro de las tecnologías web y la robótica, tiene la intención de desarrollar un simulador con peso computacional únicamente en el cliente (*WebSim*), es decir, el navegador. El simulador aprovecha el crecimiento de las tecnologías web y más específicamente el crecimiento de *AFRAME*.

Para el uso del simulador se han desarrollado además dos páginas web que permiten editar programas robóticos en el propio navegador en lenguaje JavaScript y lenguaje visual de bloques con Blockly para ejecutarlos en el propio simulador. Estas aplicaciones web son usadas por la plataforma *JdeRobot Kids* para enseñar a estudiantes las bases para resolver ejercicios de visión artificial y programación robótica.

Para el desarrollo del simulador *WebSim* se han utilizado diversas herramientas como, *AFRAME*, *AFRAME Physics* (sistema de físicas de *AFRAME*), *HTML5*, *JavaScript*, *jQuery*, *CSS3*, *Blockly* y *ACE Editor*. Por último para la gestión de dependencias se utiliza la tecnología NPM y para el empaquetado de la aplicación la herramienta *WebPack*.

Summary

This project is fits on web technologies and robotics, it intends to develop a simulator with the computational weight in the client (WebSim), the browser. The simulator takes advantage of the growth of web technologies and more specifically the growth of AFRAME.

To use the simulator I developed two web pages that allows the user to write robotic programs directly on the browser using JavaScript language and Blockly as visual blocks language to execute the program on the embedded simulator. This web applications are used by JdeRobot Kids platform to teach students the basics to solve artificial vision and robotic programming exercises.

Different tools have been used to develop the *WebSim* simulator as *AFRAME*, *AFRAME Physics system* , *HTML5*, *JavaScript*, *jQuery*, *CSS3*, *Blockly* and *ACE Editor*. Finally, NPM technology is used to manage dependencies and Webpack tool for the application's packaging.

Índice general

1. Introducción	1
1.1. Robótica	1
1.2. Robótica educativa	3
1.3. Tecnologías Web	6
1.4. Estructura de la memoria	10
2. Objetivos y planificación	13
2.1. Objetivos	13
2.2. Metodología	14
2.3. Planificación temporal	14
2.4. Control de versiones	15
3. Herramientas	17
3.1. JavaScript	17
3.1.1. Características de JavaScript	18
3.1.2. Entornos JavaScript	20
3.2. HTML	22
3.3. CSS	24
3.4. AFRAME	25
3.4.1. Primitivas y HTML	26
3.4.2. Sistema Entidad-Componente	28
3.4.3. Modelos 3-D	30
3.4.4. Herramientas de Desarrollo	30
3.5. ACE Editor	32

3.6.	Blockly	33
3.6.1.	Generadores de código	34
3.6.2.	Bloques personalizados	35
3.6.3.	Menú de bloques, <i>Toolbox</i>	36
3.7.	jQuery	37
3.8.	NPM y Webpack	39
4.	Simulador robótico web: <i>WebSim</i>	41
4.1.	Diseño	41
4.2.	Simulación con AFRAME	43
4.2.1.	Cuerpo del robot	47
4.2.2.	Sensores de distancia	48
4.2.3.	Cámara	49
4.2.4.	Escena	50
4.3.	Drivers del robot	51
4.3.1.	Constructor	52
4.3.2.	Driver de motores	54
4.3.3.	Driver del sensor de distancia a obstáculos	57
4.3.4.	Sensores de infrarrojos	60
4.3.5.	Driver de los sensores de odometría	61
4.3.6.	Driver de la cámara	62
4.4.	Control de la simulación	64
4.5.	Conexiones de <i>WebSim</i> con software externo	65
4.6.	Empaquetado e instalación	66
5.	Robótica educativa	71
5.1.	Programando con <i>JavaScript</i>	71
5.2.	Ejercicios con <i>JavaScript</i>	73
5.3.	Programando con <i>Scratch</i>	74
5.4.	Ejercicios con <i>Scratch</i>	76

<i>ÍNDICE GENERAL</i>	XI
6. Conclusiones	77
6.1. Valoración objetivo final	77
6.2. Aplicación de lo aprendido	77
6.3. Mejoras futuras	78
Bibliografía	81

Índice de figuras

1.1.	Imagen del robot <i>Da Vinci</i> usado en operaciones quirúrgicas.	2
1.2.	Imagen del robot utilizado en los centros logísticos de Amazon para acelerar el proceso de entrega de productos.	3
1.3.	Kit <i>Lego WeDo 2</i> para robótica educativa.	4
1.4.	Robot PiBot utilizado por la plataforma JdeRobot-Kids.	5
1.5.	Ejemplo de una página web que hace uso de la etiqueta canvas, webVR y aceleramiento gráfico con WebGL.	9
1.6.	Página de inicio de la web <i>appear.in</i> que hace uso de tecnologías WebRTC para hacer videollamadas.	9
1.7.	Esquema que muestra las nuevas relaciones que se establecen según evoluciona la web entre los usuarios y el contenido.	10
2.1.	Flujo de trabajo en hitos para los sprints 1 al 4	14
2.2.	Flujo de trabajo en hitos para los sprints 5 al 8.	15
2.3.	Gráfico de actividad del repositorio de <i>AFRAME</i> en Github.	16
3.1.	En la parte izquierda de la imagen se muestra el objeto <i>coche</i> del que heredan los 3 de la parte derecha de la imagen, los 3 coches tienen en común que son del objeto coche pero se diferencian en <i>marca</i> y <i>color</i>	18
3.2.	En la parte izquierda de la imagen se muestra el código que imprime la variable <i>msg</i> tras modificar su tipo	19
3.3.	Ejemplo de la declaración de variables locales con la etiqueta <i>let</i>	20
3.4.	El contexto <i>this</i> es siempre el mismo gracias a la declaración de la función de flecha.	20

3.5. La figura muestra la sintaxis para importar o exportar funciones entre distintos programas dentro de una aplicación o entre varias lo que permite reutilizar mismo código entre distintas aplicaciones.	21
3.6. Comparativa de sintaxis de declaración de un objeto antes y después del estándar ES6.	21
3.7. Estructura de una pagina HTML simple con un título y un párrafo	23
3.8. Imagen comparativa de la misma pagina web con y sin hojas CSS	25
3.9. Ejemplo de código HTML que renderiza una escena básica de realidad virtual .	27
3.10. La siguiente figura muestra etiquetas que serían el equivalente en AFRAME a un componente, el conjunto de varios componentes dan forma a la caja	28
3.11. Componente básico que imprime por la consola del navegador el mensaje que se le pase por <i>message</i>	29
3.12. Ejemplo de acceso a un elemento de AFRAME, como se aprecia se hace de la misma manera que se accede a cualquier elemento de HTML convencional . .	30
3.13. Inspector visual que ofrece el entorno <i>AFRAME</i>	31
3.14. Posibles parámetros de configuración para el editor	33
3.15. Implementación de un contador para evitar bucles infinitos al traducir lenguaje de <i>Blockly</i>	35
3.16. Modos de configuración de un bloque personalizado en <i>Blockly</i> , como se puede apreciar la declaración de las distintas partes del bloque es bastante intuitiva y los parámetros son autodescriptivos.	35
3.17. Muestra del código que inicia el bloque para que se muestre en el editor visual, <i>moveBlock</i> representa un objeto JSON con la configuración del bloque	36
3.18. Herramienta de desarrollo para generación de bloques personalizados.	37
3.19. Ejemplo de las etiquetas posibles de configuración de la <i>toolbox</i> del editor . . .	37
3.20. Instrucción para hacer un efecto de ' <i>fundido</i> ' en jQuery.	38
4.1. Estructura de WebSim.	42
4.2. HTML que genera la escena principal del simulador	45
4.3. Etiqueta <i>a-scene</i> , es la etiqueta principal del entorno <i>AFRAME</i>	46

4.4. <i>a-assets</i> es la etiqueta usada por <i>AFRAME</i> para la gestión de texturas y recursos externos.	46
4.5. Etiquetas utilizadas para la simulación del robot en el entorno <i>AFRAME</i>	47
4.6. Robot simulado en la escena, se resalta la cámara integrada en él.	48
4.7. Registro de los tres componentes necesarios para el Robot simulado.	49
4.8. Etiqueta vacía utilizada para encapsular los sensores de distancia a obstáculos. .	49
4.9. Etiquetas generadas por <i>WebSim</i> para la simulación del sensor de ultrasonido. .	49
4.10. Etiquetas utilizadas para crear diferentes elementos de la escena del simulador <i>WebSim</i>	50
4.11. Código del constructor del objeto robot, la variable <i>this</i> hace referencia al contexto.	54
4.12. Cálculo de la nueva posición en función de la orientación y la velocidad lineal.	55
4.13. Esquema de ejecución de la función <i>setVelocity</i> , en la parte superior izquierda de la imagen se muestran algunos datos que existen en el contexto de la función.	56
4.14. Sistema de ejes utilizado en las escenas del entorno <i>AFRAME</i>	63
4.15. Fichero de configuración de la herramienta <i>Webpack</i>	67
4.16. Ficheros de entrada que tomará <i>Webpack</i> para hacer el empaquetado.	68
4.17. Nombre y directorio de los ficheros de salida empaquetados por <i>Webpack</i>	68
4.18. Declaración de las extensiones que ha de buscar <i>Webpack</i>	69
4.19. Declaración de la configuración del servidor estático para desarrollo que <i>Webpack</i> permite usar.	69
4.20. Configuración de los módulos de preprocesado a usar por <i>Webpack</i>	70
5.1. Arquitectura de la aplicación que hace uso del editor ACE y <i>WebSim</i>	72
5.2. Interfaz de usuario de <i>WebSim</i> + editor JavaScript (Ace Editor) usado en <i>Jde-Robot Kids</i>	74
5.3. Interfaz de la aplicación <i>WebSim</i> + <i>Blockly</i>	75
5.4. Arquitectura de la aplicación web que hace uso de <i>Websim</i> + <i>Blockly</i>	75

Capítulo 1

Introducción

La motivación de este proyecto es la de sustituir el presente simulador *Gazebo* utilizado por la plataforma educativa *JdeRobot-Kids* ya que éste tiene un gran peso computacional en el servidor. Se busca la creación de un simulador con peso únicamente en el lado cliente lo que permite escalar a un mayor número de usuarios simultáneos.

1.1. Robótica

La robótica es la rama tecnológica que está involucrada en el diseño, fabricación y la utilización de robots. Un robot es una máquina que puede programarse para que interactúe con otros objetos y realice de manera autónoma tareas costosas para las personas. La robótica combina, entre otras, la informática, electrónica, ingeniería y en los últimos años la visión artificial, esta última es muy importante ya que la imagen representa un sensor muy potente (contiene mucha información) para el robot.

El desarrollo tecnológico ha generado cambios a nivel social facilitándonos a las personas la mayoría de las tareas que llevamos a cabo a lo largo del día tanto dentro del trabajo como pueden ser largas cadenas automáticas de fabricación de productos o en el campo de la medicina con la cirugía como en el ámbito del hogar como es el caso del aspirador Roomba, esto ocurre gracias al avance de la rama de la robótica. Este campo tiene como objetivos el simplificar las tareas de las personas tanto en la vida cotidiana como en trabajos de alto riesgo mediante el uso de autómatas cada vez más desarrollados que sean capaces de realizar tareas más complicadas

incluso de manera más eficaz a la que la haría un humano.

La robótica está presente cada vez en más campos de desarrollo, tenemos aplicaciones que hacen uso de la robótica en:

- Automoción, el campo de la automoción esta experimentando un fuerte crecimiento en el uso de la robótica para el desarrollo de coches autónomos, es decir, coches autopilotados (no necesitan un conductor). Una de las empresas más importantes en este sector es *Waymo*, es una de las empresas de Alphabet (Google) y ya está probando con coches autónomos de nivel 4 que no necesitan un conductor de seguridad. El siguiente enlace muestra un vídeo de cómo funcionan estos coches autónomos. <https://www.youtube.com/watch?v=aaOB-ErYq6Y>
- Medicina, cabe destacar la existencia del robot llamado *Da Vinci* (figura 1.1) que se ha convertido en uno de los referentes de la cirugía. Se trata de un dispositivo a través del cual se han conseguido llevar a cabo con éxito operaciones tan importantes como las de cirugía transoral, esta cirugía se basa en el uso de un brazo robótico que puede manejar el cirujano para la extracción de cáncer en partes de la boca y garganta de acceso difícil.



Figura 1.1: Imagen del robot *Da Vinci* usado en operaciones quirúrgicas.

- Centros logísticos, este campo se encuentra en desarrollo, una de las empresas que más se importancia adopta en este ámbito es Amazon con el uso de *Drones* para la entrega de

artículos y sus robots *Drives* que usan de manera interna para acelerar las entregas en sus centros logísticos. Estos robots lo que hacen es deslizarse debajo de grandes estanterías donde se encuentran los artículos y las llevan hacia los operarios o el destino indicado al robot. España es actualmente el tercer país en adoptar esta tecnología de la empresa Amazon en los centros logísticos. En la siguiente figura se muestra los robots utilizados por la empresa Amazon.



Figura 1.2: Imagen del robot utilizado en los centros logísticos de Amazon para acelerar el proceso de entrega de productos.

Actualmente la industria de la robótica está experimentando un gran crecimiento entrando en un mayor número de campos tecnológicos, se prevé un aumento de la demanda de profesionales en el campo de la robótica debido a esta diversidad y a la mayor sofisticación de los robots. Un ejemplo del grado de avance que tienen actualmente los robots se puede ver en el vídeo a continuación que presenta los prototipos de la empresa *Boston Dynamics* en el que se puede apreciar que cada vez los robots tienen funcionalidades más complejas (minuto 9 en el vídeo).

<https://www.youtube.com/watch?v=KEMt58ePNDs>

1.2. Robótica educativa

La robótica educativa ofrece entornos de aprendizaje basados en la actividad de los estudiantes, es decir, aprender el pensamiento lógico que va más allá de la programación o el diseño

de robots mediante el uso de entornos 'simplificados' para el alumno. Además fomenta la resolución de problemas y el trabajo en equipo a través de recursos tecnológicos. La robótica es un campo multidisciplinar que conjunta el conocimiento matemático, físico y tecnológico por lo tanto es un campo que conjuga muy bien con las actuales materias educativas a la vez de ser una materia de aplicación, es decir, los alumnos pueden ver el resultado de la aplicación de estos conocimientos en el robot.

Muchos proyectos han demostrado que el uso de kits de robótica para el aprendizaje de los alumnos aumenta la capacidad de reflexión de estos, Jhon Siraj-Blatchford, profesor de la universidad de Cambridge, en el libro *Nuevas tecnologías para la educación infantil y primaria* argumenta este tema. Además se ofrece un entorno educativo distinto al tradicional, más adaptado al mundo actual donde la tecnología crece a gran velocidad y donde los alumnos pueden conocer una motivación por un entorno al que no tendrían acceso hasta unos estudios superiores o especializados. La figura 1.3 muestra un kit de robótica educativa en el que se ofrecen piezas para la construcción de una serie robots simples y la herramienta para programar su funcionalidad.



Figura 1.3: Kit *Lego WeDo 2* para robótica educativa.

Dentro del marco de la robótica educativa cabe hablar de la plataforma *JdeRobot-Kids* en la cual se enmarca el presente proyecto. Esta plataforma ofrece las herramientas necesarias para la enseñanza de robótica en alumnos de secundaria. La plataforma tiene una intención de potenciar los conocimientos de robótica previos a los estudios universitarios, para ello provee de

una infraestructura software, hardware y una colección de ejercicios que permiten el desarrollo completo del curso. Tiene soporte para robots tanto reales como simulados, el lenguaje de programación usado es Python y los robots utilizan el hardware de *Raspberry PI*. La propuesta educativa llevada a cabo por la plataforma se ha puesto en práctica con éxito en la Fundación Franciscanas de Montpillier. Actualmente la plataforma cuenta como infraestructura hardware con el robot *PiBot* el cual cuenta con los siguientes componentes, cámara, sensores de ultrasonido, sensores infrarrojos y dos servomotores todo ello conectado a la *Raspberry PI*. Además cuentan con el robot simulado en el simulador Gazebo. Lo que dota al estudiante de flexibilidad a la hora de realizar los ejercicios fuera del horario lectivo. En la figura a continuación se muestran una imagen de la infraestructura hardware (PiBot).

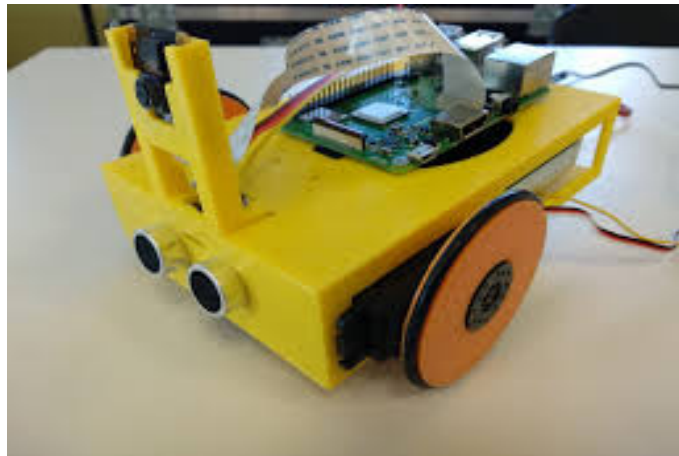


Figura 1.4: Robot PiBot utilizado por la plataforma JdeRobot-Kids.

Además como se ha mencionado anteriormente, cada vez el desarrollo y sofisticación de estos autómatas es mayor debido a que se intenta que tengan una funcionalidad similar a la de un humano dotándolos incluso de *inteligencia artificial* por tanto es necesario profesionales cada vez mejor formados lo que conlleva la necesidad de formación en esta rama desde edades más tempranas.

En el siguiente enlace se muestra un vídeo en el cual se hablan de las posibles aplicaciones de la robótica en la educación, en la charla se expone cómo aprenden los niños con la robótica los problemas de la contaminación lumínica para los animales.

<https://www.youtube.com/watch?v=FDnhMTxhddM>

1.3. Tecnologías Web

Una de las ventajas principales que ofrecen las tecnologías web es la ausencia de necesidad de instalación de paquetes de software, configuración y dependencias. Basta con tener un navegador y una conexión a internet, todos los archivos necesarios para la ejecución de la aplicación se sirven de manera automática al ingresar la URL dentro de la barra de navegación.

El cliente no tiene la necesidad de instalar actualizaciones ya que únicamente se actualiza la versión que proporciona el servidor, esto elimina las incompatibilidades entre versiones ya que todos los clientes usarán la misma versión. Desarrollo unificado, con esto hacemos referencia a que no se necesita desarrollar para los distintos sistemas operativos (Windows, MacOS, Linux/Ubuntu, etc.) así como conocer sus entornos gráficos y dependencias del sistema operativo, lo único necesario es saber HTML5, JavaScript y CSS3, el navegador se encarga de interpretar los distintos lenguajes.

No todo son ventajas, como desventaja sabemos que las aplicaciones web son algo más lentas debido a la necesidad de descargar los recursos y no ser lenguajes compilados como C++ o Java sino interpretados, ésta desventaja cada vez va siendo menor debido a las mejoras en los interpretes de JavaScript en los navegadores y protocolos como por ejemplo *AJAX (Asynchronous JavaScript and XML requests)* que trata de una técnica de peticiones ligeras para aplicaciones interactivas.

Además con los años ha aumentado el número de navegadores distintos y desarrollar la aplicación para todos ellos es costoso aunque existen entornos que facilitan esta tarea como pueden ser Express y Loopback para el lenguaje JavaScript y Django para el lenguaje Python.

El uso de las tecnologías web ha aumentado debido al lanzamiento del nuevo estándar HTML 5 en el que tenemos características muy importantes para la transmisión de contenido multimedia con la inserción de la etiqueta *video* y *canvas* en las que podemos incluir vídeos y generar escenas 2D y 3D respectivamente. A continuación se enumeran y explican algunas de las características más destacables del nuevo estándar HTML 5.

- Aceleración gráfica con WebGL y la etiqueta *canvas*, esto permite el uso de aceleración gráfica hardware basada en OpenGL evitando el uso de *plugins* (extensiones) y generando estos gráficos a partir de código JavaScript. Los elementos WebGL se pueden mezclar con otros elementos HTML y componerse con otras partes de la página o el fondo de la misma.
- WebWorkers, el nuevo estándar HTML 5 ofrece un API para generar ejecuciones de código en segundo plano en paralelo con el programa principal, esto permite operaciones por hilos de ejecución con un mecanismo de mensajes entre hilos para el control de ejecución.
- WebStorage, este API es la evolución del mecanismo de las cookies para mantenimiento de estado en el protocolo HTTP el cual sabemos que es sin estado. El tipo de almacenamiento es similar al de las cookies, se usa el par clave-valor. La principal diferencia entre el ambos mecanismos es que WebStorage ofrece mayor capacidad para guardar datos en el navegador y una mejor seguridad.
- WebSockets, este API permite abrir un canal de comunicación bidireccional con un servidor o con una aplicación de terceros. El tipo de comunicación usada es mediante eventos, ambas partes de la comunicación deben hacer uso del mismo tipo de mensajes. Los mensajes los define el desarrollador de la aplicación. El protocolo de apertura de WebSockets consiste en el protocolo *handshake* en el cual ambas partes de la comunicación se ponen de acuerdo en el canal a usar y entonces se inicia la transmisión de mensajes sobre TCP. Como características destacar que al ir sobre el protocolo TCP se ofrece redundancia, es decir, si el mensaje se pierde se vuelve a enviar ofreciendo la fiabilidad de que el mensaje llegará al destino y además son transmisiones de baja latencia.
- ServerSent Events, define un API para la apertura de una conexión HTTP en la cual el servidor puede enviar notificaciones al cliente, dotando de capacidad de iniciativa de envío de mensajes del lado servidor. Previo a HTML 5 esto no era posible, el servidor tenía que esperar una petición del cliente para responder.
- WebRTC, define un API de transmisión y recepción de contenido multimedia desde otro navegador o dispositivo que implemente los protocolos de transmisión en tiempo real. Lo más importante de esta API es que abstrae al desarrollador de toda la problemática que

trae consigo el envío de audio y vídeo como son el retardo, el jitter, el uso del mismo formato de transmisión, etc. El jitter es un problema muy importante en transmisión de audio y vídeo y es la variación del retardo de la transmisión debido a la fluctuación de la red. Una aplicación que hace uso de este API es `www.appear.in` que es un servicio de videollamada similar a Skype pero en Web.

- WebVR, especificación web que permite generar escenas de realidad virtual en aplicaciones web. Provee de un API JavaScript que da soporte a la mayoría de dispositivos de realidad virtual como *HTC Vive*. La intención del API es detectar los dispositivos, sus características, la frecuencia de refresco de funcionamiento y ofrecer control de posición. Utiliza el motor WebGL anteriormente mencionado para el renderizado de las imágenes 3D.

A continuación se muestran imágenes que presentan aplicaciones web que utilizan algunas de estas mejoras del nuevo estándar HTML 5.

Estos avances en las tecnologías web conforman lo que es conocido como web 2.0 que se centra en fomentar la interacción con el usuario final, el compartir información y diseño centrado en el usuario. Gracias a HTML 5 se está experimentando un modelo de web semántica en la cual cada elemento de la red está identificado unívocamente lo que permite enlazar la información permitiendo que el acceso a ella sea mucho más sencillo. Lo que se intenta emular son las relaciones entre elementos de la web como si una base de datos se tratase en el que todos los datos tienen una relación. En la figura ?? se muestra de manera esquemática y muy resumida la evolución de la web.

Un ejemplo de web semántica se encuentra en *dbpedia* (<https://wiki.dbpedia.org/>) que se trata de un proyecto que toma los datos de la web *Wikipedia* pero estableciendo relaciones entre todos los elementos existentes de la wikipedia.

Gracias a este crecimiento de las tecnologías y estándares web comienzan a surgir numerosos entornos de desarrollo tanto para la parte cliente como la parte servidora de un servidor web. Estos entornos desarrollar de manera más eficiente y menos costosa nuestra aplicación web. A continuación se enumeran algunos entornos de desarrollo más usados actualmente.



Figura 1.5: Ejemplo de una página web que hace uso de la etiqueta canvas, webVR y aceleramiento gráfico con WebGL.

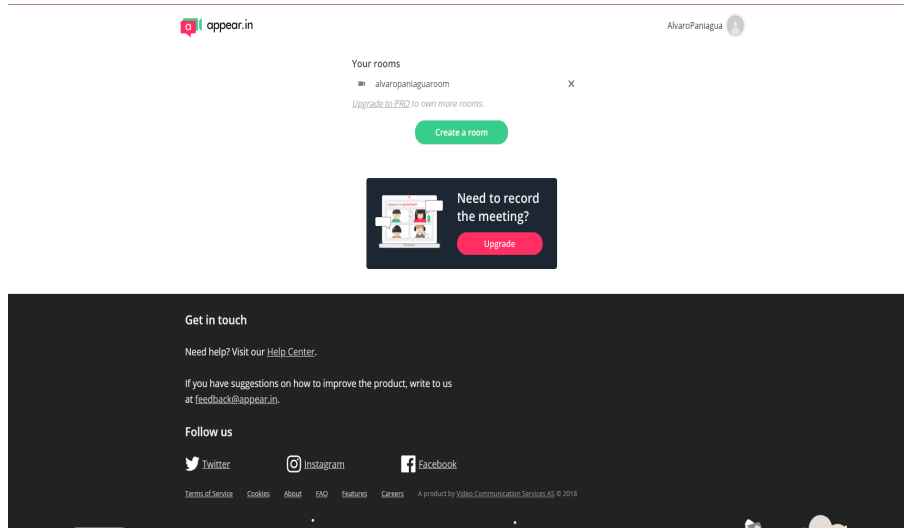


Figura 1.6: Página de inicio de la web *appear.in* que hace uso de tecnologías WebRTC para hacer videollamadas.



Figura 1.7: Esquema que muestra las nuevas relaciones que se establecen según evoluciona la web entre los usuarios y el contenido.

- NodeJS, entorno de desarrollo de servidores en lenguaje JavaScript. Provee de las herramientas básicas para el manejo de peticiones del protocolo HTTP.
- Django, entorno de desarrollo de servidores en lenguaje Python, implementa el MVC (Modelo Vista Controlador) que simplifica mucho la creación de paginas web gracias a su motor de plantillas.
- React, entorno de desarrollo de interfaces de usuario, permite desacoplar el lado cliente del lado servidor y programación orientada a componentes.
- Loopback, entorno de desarrollo de servidor en lenguaje JavaScript, se construye encima de NodeJS y se utiliza para acelerar el desarrollo de API Rest.

Como conclusión, el desarrollo de las tecnologías web está experimentando actualmente un crecimiento muy grande. Cada vez las aplicaciones web son capaces de integrar más capacidades y de manera más eficiente lo que hace que cada vez sean más los desarrolladores de software que adoptan este modelo de desarrollo.

1.4. Estructura de la memoria

En esta sección se detalla la estructura de la memoria que constará de las siguientes partes:

- El capítulo 1 es una introducción al campo en el que se desarrolla el proyecto, se explica la motivación del proyecto y la motivación personal.
- En el capítulo ?? se muestran los objetivos a completar para la elaboración del proyecto y la estructura de la hoja de ruta.
- En el capítulo 3 se presentan las tecnologías que se han utilizado para el desarrollo del proyecto y se explica el porqué de dichas tecnologías y no otras.
- En el capítulo 4 se explicarán el diseño de la aplicación, soporte del robot, es decir, qué funcionalidades tiene y la conectividad que dispone el simulador.
- En el capítulo ?? se muestran ejercicios que el alumno podrá resolver usando el lenguaje *JavaScript* y haciendo uso del lenguaje de bloques visuales *Blockly*.
- Finalmente en el capítulo 6 se hace una valoración de todo lo que ha conllevado el proyecto y se proponen futuras implementaciones o mejoras de WebSim.

Capítulo 2

Objetivos y planificación

2.1. Objetivos

Mi trabajo fin de grado consiste en crear la base de una herramienta educativa (WebSim) para simulación de robots para la plataforma JdeRobot Kids en la cual el peso computacional la lleve el lado cliente en lugar del lado servidor.

El simulador ha de soportar la infraestructura hardware utilizada actualmente por JdeRobot-Kids, el PiBot, que está formado por los siguientes sensores y actuadores:

- *Motores*: dos servomotores independientes que dotan de movimiento al robot.
- *Cámara*: una minicámara, ésto le da funcionalidad muy importante al robot como puede ser la detección de obstáculos.
- *Sensores IR*: dos sensores infrarrojos posicionados en la parte baja del chasis del robot, estos sensores permiten la detección de colores, objetos y formas.
- *Sensor de ultrasonidos*: Dos sensores de ultrasonido posicionados en la parte delantera del chasis del robot, su funcionalidad es la de sensores de proximidad lo que permite saber no solo si hay un objeto delante sino que también permite saber a qué distancia está dicho objeto.

Además existen dos objetivos más, la creación de dos aplicaciones web para el uso de WebSim a través de dos editores de código distintos en los cuales los alumnos que usen el simulador podrán resolver una serie de ejercicios educativos propuestos por JdeRobot-Kids.

2.2. Metodología

Para el desarrollo del proyecto se ha seguido la metodología *Agile*, es una forma de realizar los proyectos en la cual el proyecto al completo se parte en partes más pequeñas que se desarrollan en un par de semanas. De modo que el cliente puede ir haciendo pequeños cambios al proyecto en función de la ventaja de mercado que quiera obtener y le permite ir viendo el desarrollo del proyecto en intervalos de tiempo reducidos. Estos intervalos de tiempo se llaman *sprints* en el cual el desarrollador se centra únicamente en programar el software y si el *sprint* lo permite se genera documentación de lo hecho en el *sprint*.

Para la simulación de esta metodología se han hecho reuniones semanales con los tutores del TFG en el cual se les presentaba un prototipo y se proponían cambios, mejoras y desarrollos a llevar a cabo en la próxima semana. Por norma los *sprints* se sobredimensionaban, es decir, se proponían desarrollos que no iban a entrar en la planificación temporal, esto se ha hecho así para mantener la idea general del proyecto y no desviarnos de esta idea y además poder evitar el problema de no desarrollar nada si lo fijado en la reunión ya se había completado.

2.3. Planificación temporal

Como se ha comentado en la subsección anterior se ha llevado una metodología en *sprints*, a continuación se muestran dos imágenes con los intervalos temporales que han llevado cada uno de los *sprints*.

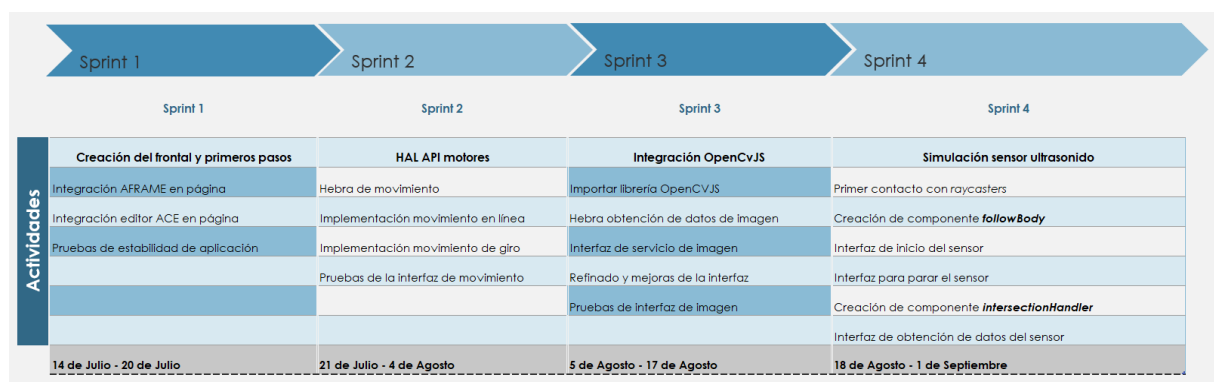


Figura 2.1: Flujo de trabajo en hitos para los sprints 1 al 4

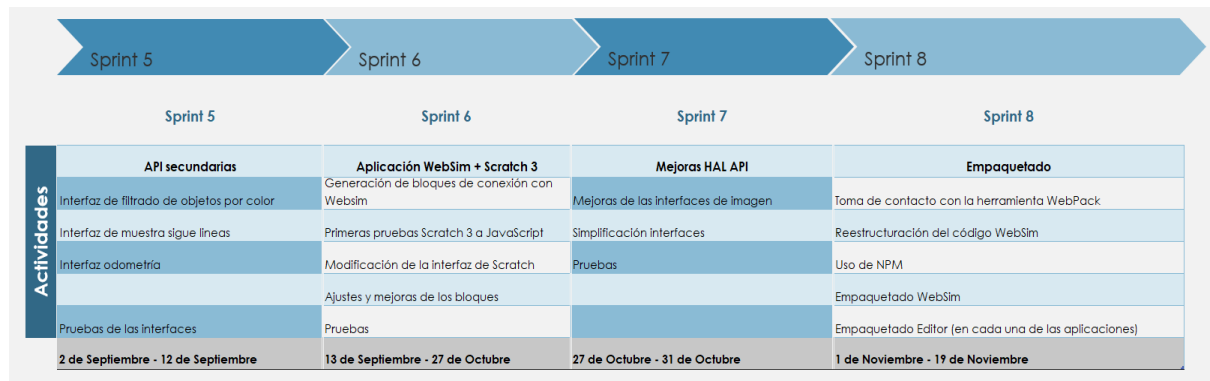


Figura 2.2: Flujo de trabajo en hitos para los sprints 5 al 8.

Como se ve en las imágenes algunos de los *sprints* han ocupado más de una reunión semanal. En esas reuniones se presentaba el estado actual del proyecto aunque no estuviesen finalizados los hitos marcados del *sprint*.

El nivel de esfuerzo para este proyecto ha sido alto debido a la necesidad de aprender diferentes tecnologías como son AFRAME, jQuery, OpenCVjs, WebPack y NPM. Se dedicaban alrededor de 4-5 horas al día cada día de la semana a excepción de los fines de semana que se añadían 2 horas más al anterior intervalo.

2.4. Control de versiones

Como en todo desarrollo de software es necesario el uso de una plataforma de control de versiones, para este proyecto se ha utilizado la plataforma de control de versiones *GitHub*. Github es una plataforma que permite crear varias versiones y volver a una versión anterior si la actual deja de funcionar. Permite crear ramas distintas con funcionalidades de código distintas lo que permite hacer pruebas de una funcionalidad sin 'estropear' el código ya existente. Además permite tener tu código actualizado en cualquier ordenador.

Github tiene una extensa comunidad de desarrolladores, los repositorios son de uso libre, es decir, cualquiera que use Github puede clonar el código de tu repositorio para usarlo, modificarlo e incluso contribuir en él ayudando a solventar errores o descubriendo nuevas incidencias. Todas estas características lo hacen muy potente y admitido por la comunidad del desarrollo de

software en general.

Como última característica pero no menos importante Github permite visualizar de forma gráfica la actividad que tiene un repositorio lo que permite indicar qué repositorios están muy activos o tienen una gran comunidad de desarrolladores que le dan soporte y cuáles están en desuso.

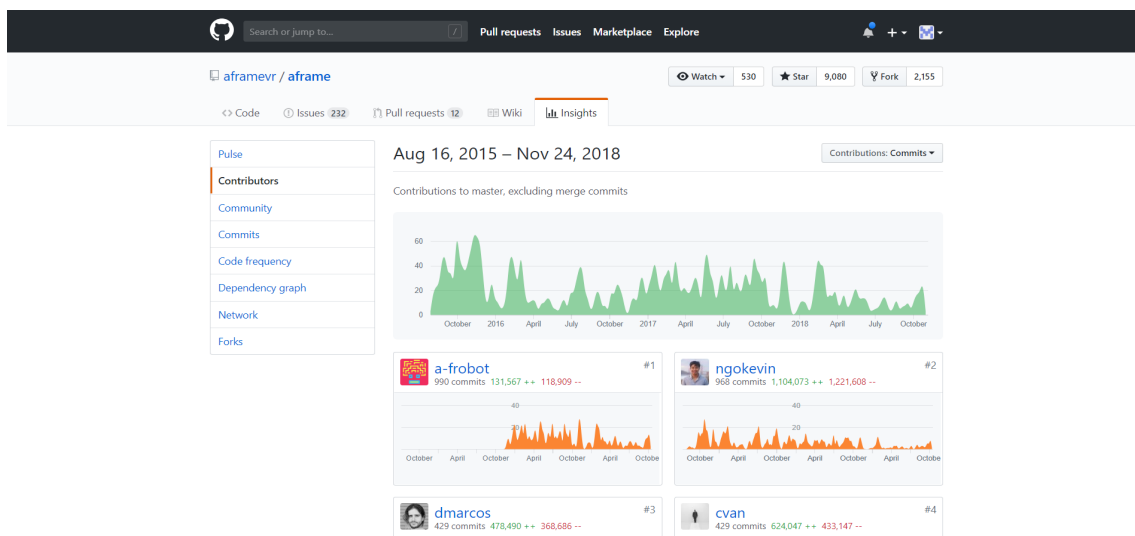


Figura 2.3: Gráfico de actividad del repositorio de *AFRAME* en Github.

Se ha trabajado en dos repositorios distintos que a continuación se mencionan.

<https://github.com/JdeRobot/WebSim>

https://github.com/RoboticsURJC-students/2018-tfg-alvaro_paniagua

En el primer repositorio la metodología de trabajo era mediante incidencias en las cuales se marcaba una incidencia importante, se creaba una rama para arreglar la incidencia y se subía un *pull request* a modo de parche para la incidencia lo que permite que el desarrollo quede muy limpio. Esta metodología es útil en repositorios en los cuales hay varios desarrolladores trabajando en partes distintas del software como era el caso.

En el segundo repositorio al haber un único desarrollador se ha trabajado únicamente con *commits*.

Capítulo 3

Herramientas

En este capítulo se desarrollarán de manera breve las herramientas utilizadas para el desarrollo del proyecto. Estas herramientas han sido elegidas debido a que eran las que mejor se ajustaban a las necesidades del proyecto.

Algunas de las herramientas utilizadas como HTML, y JavaScript se han elegido debido a que son los estándares de desarrollo para aplicaciones web.

3.1. JavaScript

JavaScript fue creado por Brendan Eich en 1995 cuando trabajaba para Netscape Communications inspirado por el lenguaje Java. Se encuentra actualmente bajo el estándar *ES6 (ECMAScript 6 o ECMAScript 2015)* que añade ciertas características al lenguaje que se explicarán más adelante.

Es el lenguaje más utilizado para el desarrollo Web, permite que las aplicaciones web sean interactivas, es decir, permite hacer actualizaciones de contenido en el momento, mostrar mapas, animaciones 3D. Es el tercer pilar del estándar de tecnologías web compuesto por HTML, CSS y JS.

3.1.1. Características de JavaScript

JavaScript es un lenguaje de *scripting* orientado al lado cliente de una aplicación como ya se ha comentado anteriormente, y cuenta con las siguientes características:

- Lenguaje de *alto nivel*, esto quiere decir que su sintaxis es similar a la escritura habitual de una persona, por ejemplo:

```
function myFunction() {  
    console.log("Hello world");  
}
```

- Lenguaje basado en *objetos*, esto es una estructura habitual en programación que se refiere a la encapsulación de operaciones y estados en un modelo de datos. Otros lenguajes orientados a objetos serían *Python*, *Ruby*, *Java*, *etc.* La figura 3.1 representa de manera visual la orientación a objetos.

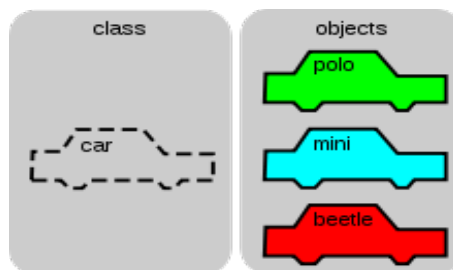
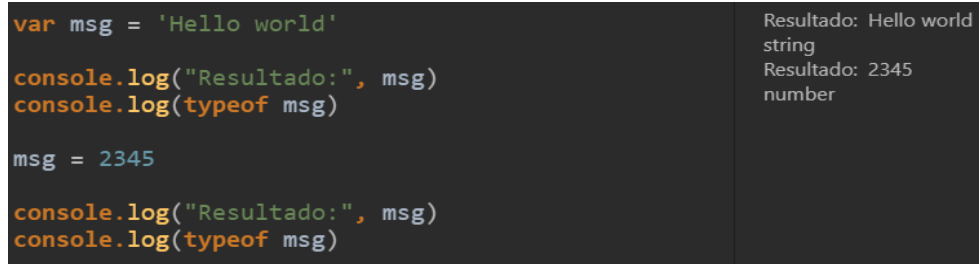


Figura 3.1: En la parte izquierda de la imagen se muestra el objeto *coche* del que heredan los 3 de la parte derecha de la imagen, los 3 coches tienen en común que son del objeto coche pero se diferencian en *marca* y *color*.

- Tipado débil, esto quiere decir que no es necesario declarar el tipo de una variable, una variable cualquiera puede ser de distintos tipos en momentos distintos pasando de un *string* a un *number* o a un *objeto*. En la figura 3.2 se muestra un ejemplo de esta característica en el que la variable *msg* es declarada inicialmente con contenido de tipo *string* y posteriormente se modifica su contenido para que sea de tipo *number*.
- Es un lenguaje *case-sensitive*, es decir, distingue las letras mayúsculas de las minúsculas, a la hora de declarar una variable o función no es lo mismo *miVariable* que *mivariabLe*.



```
var msg = 'Hello world'
console.log("Resultado:", msg)
console.log(typeof msg)

msg = 2345
console.log("Resultado:", msg)
console.log(typeof msg)
```

Resultado: Hello world
string
Resultado: 2345
number

Figura 3.2: En la parte izquierda de la imagen se muestra el código que imprime la variable *msg* tras modificar su tipo

La manera correcta para definir nombres de variables o funciones en este lenguaje es seguir la sintaxis *camel-case* en la cual si el nombre de mi variable está compuesta por varias palabras la primera de ellas estará completamente en minúsculas y las palabras que la siguen tendrán la primera letra en mayúsculas, a continuación se muestra un ejemplo: *miVariableDeMuestra*.

- Al igual que otros lenguajes como *Python*, *JavaScript* es un lenguaje *interpretado* esto quiere decir que no se necesita un compilador para crear un binario del código sino que existe un intérprete dentro del navegador que se encarga de ejecutarlo.

Como hemos comentado en la introducción de *JavaScript* éste se encuentra bajo el estándar *ES6* lo cual le dota de una serie de características importantes, a continuación se citarán algunas de las características que son más relevantes para el proyecto:

- Definición de variables con ámbito local (*Block-Scoped Variables*), esta característica permite declarar una variable que únicamente existirá en un determinado bloque y no fuera de éste, mejora la inteligibilidad del código a la hora de que lo tengan que leer distintos desarrolladores de un equipo. Esta característica existía anteriormente pero el desarrollador debía tener conocimiento del *scoping* de variables, es decir, cuando una variable afectaba a un bloque y cuándo no. En la figura 3.3 se muestra un uso de este tipo de declaración, la variable *x* únicamente existe dentro del bucle *for*.
- Funciones de flecha, en el estándar *ES6* se permite la declaración de funciones sin la creación de un contexto *this*, este contexto hace referencia al bloque en el que se encuentra. Anteriormente a *ES6* si se necesitaba usar el contexto de otra función en un determinado

```
for (let i = 0; i < a.length; i++) {
  let x = a[i]
  ...
}
```

Figura 3.3: Ejemplo de la declaración de variables locales con la etiqueta *let*.

bloque había que hacer la siguiente transformación *var self = this*; lo que permitía usar el contexto dentro de una nueva función llamando a *self*, gracias a ES6 esto ha cambiado, la figura 3.4 muestra un ejemplo de la nueva sintaxis.

```
this.nums.forEach((v) => {
  if (v % 5 === 0)
    this.fives.push(v)
})
```

Figura 3.4: El contexto *this* es siempre el mismo gracias a la declaración de la función de flecha.

- Exportar e importar módulos, esta característica es similar a los *import* de *Python* lo que dota de mucha modularidad a la aplicación ya que permite tener un programa principal que controla el uso de las distintas funcionalidades evitando así las aplicaciones monolíticas y permite que cada nueva funcionalidad sea completamente independiente de las demás lo que hace que la modificación de esta funcionalidad no afecte a las demás. La figura 3.5 muestra la sintaxis para exportar una variable y una función para ser utilizada en distintos scripts.
- Declaración de clases, es una modificación que permite mejorar la inteligibilidad del código ya que esta característica se permitía en anteriores estándares pero con una sintaxis distinta que podría dar lugar a confusiones a la hora de entender el código. La figura 3.6 muestra una comparativa entre ambas sintaxis, en la parte superior vemos la sintaxis en estándares anteriores y en la parte inferior la sintaxis de ES6.

3.1.2. Entornos JavaScript

Este lenguaje tiene mucho peso en el lado cliente, es decir, en el código que se ejecuta en el navegador pero gracias a los avances anteriormente mencionados y a la llegada del entorno *NodeJS* este lenguaje ha comenzado a tomar más importancia en el lado servidor debido a que


```
// lib/math.js
export function sum (x, y) { return x + y }
export var pi = 3.141593

// someApp.js
import * as math from "lib/math"
console.log("2π = " + math.sum(math.pi, math.pi))

// otherApp.js
import { sum, pi } from "lib/math"
console.log("2π = " + sum(pi, pi))
```

Figura 3.5: La figura muestra la sintaxis para importar o exportar funciones entre distintos programas dentro de una aplicación o entre varias lo que permite reutilizar mismo código entre distintas aplicaciones.

```
var Shape = function (id, x, y) {
  this.id = id;
  this.move(x, y);
};
Shape.prototype.move = function (x, y) {
  this.x = x;
  this.y = y;
};

class Shape {
  constructor (id, x, y) {
    this.id = id
    this.move(x, y)
  }
  move (x, y) {
    this.x = x
    this.y = y
  }
}
```

Figura 3.6: Comparativa de sintaxis de declaración de un objeto antes y después del estándar ES6.

ya no existe la necesidad de aprender dos lenguajes (javascript en cliente, ruby, python, java en el servidor) sino que comienzan a surgir los desarrolladores *fullstack JavaScript*. Debido a éste 'boom' del lenguaje se comienzan a crear distintos entornos de desarrollo que tratan de cubrir distintas problemáticas en el desarrollo web.

Actualmente *JavaScript* se utiliza en multitud de entornos tanto de lado cliente como en el lado servidor, a continuación se enumeran algunos de los más usados:

- *NodeJS*: Tecnología en el lado servidor con código completamente JavaScript asíncrono y orientado a eventos, NodeJS fué diseñado para construir aplicaciones en red escalables. Su principal característica es la capacidad de gestionar multitud de conexiones simultáneas de una manera muy eficaz gracias a su arquitectura. Se basa en un hilo que

escucha peticiones y las redirecciona a *hebras* distintas, cada una de estas hebras ejecuta el código necesario y una vez la hebra termina lanza un evento al hilo de peticiones indicando que ha terminado de ejecutar la tarea indicada entonces es el hilo de peticiones el que se encarga de devolver la respuesta.

- *Express y Loopback*: Ambos entornos son extensiones de NodeJS y permiten la creación de un API en el lado servidor. Loopback es actualmente más importante que Express ya que ofrece muchas herramientas para la creación de API Rest de manera muy sencilla así como conectores a la mayoría de bases de datos como puede ser MongoDB, mySQL, sqlite3, etc.
- *AngularJS* Es un entorno basado en el *Modelo Vista Controlador* para el desarrollo en la parte cliente que permite la creación de aplicaciones web *SPA (Simple-Page-Application)*. AngularJS permite extender el lenguaje HTML con directivas y atributos sin perder la semántica.
- *AFRAME* Es un entorno de creación de escenas de *Realidad Virtual*, se hablará de manera más extensa en el apartado 3.4. No es un entorno muy extendido debido a su juventud pero cuenta con una gran comunidad de desarrolladores y es el pilar central del proyecto que se lleva a cabo.

Esta imagen creada de los entornos disponibles para JavaScript sirve como respaldo ante la decisión de elegir el lenguaje. Además como se ha comentado en el capítulo 1 el proyecto se orienta a la creación de una aplicación con peso en el lado cliente, por tanto sabiendo esto y teniendo en cuenta que AFRAME está creado en el lenguaje JavaScript la elección de este lenguaje adquiere importancia por sí misma.

3.2. HTML

HTML fue creado por *Tim Berners-Lee* en 1990 y es el acrónimo para *HyperText Markup Language* (Lenguaje de marcas de hipertexto).

Se utiliza para la creación de documentos electrónicos que se envían a través de la red global (internet). Cada documento tiene una serie de conexiones a otros documentos llamados *hyperlinks* que permiten la navegación entre distintos recursos.

HTML asegura el formato correcto de texto, imágenes y estilos para poder leer un documento con el navegador con la forma original con la que se generó el documento. En la figura 3.7 se muestra una página HTML muy simple y se explican sus distintas partes y su función.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset=utf-8 />
    <title> Mi primera pagina WEB</title>
  </head>
  <body>
    <h1> Hola Mundo! </h1>
    <p> Mi primera pagina HTML</p>
  </body>
</html>
```

Figura 3.7: Estructura de una pagina HTML simple con un título y un párrafo

Como vemos en la figura anterior, un documento HTML tiene una estructura de árbol donde la etiqueta *html* es el elemento raíz y cada nuevo elemento es una rama del anterior. Estas 'ramas' se pueden ir extendiendo según la necesidad del proyecto web.

Como elementos principales del documento HTML tenemos la declaración documento *DOCTYPE html* que en este caso indica que estamos ante un documento HTML 5 que es última versión de HTML. Como hemos indicado en el párrafo anterior la etiqueta *html* marca la raíz del documento, dentro de ésta etiqueta tenemos dos etiquetas importantes:

- **HEAD** es la cabecera del documento, contiene los metadatos del documento como el título, la codificación de caracteres utilizada y links a otros recursos adicionales como pueden ser *scripts* y *hojas de estilos*.
- **BODY** es el contenido que se mostrará del documento, puede contener imágenes, enlaces a otros documentos, vídeos, menús de navegación, formularios, botones e incluso escenas animadas como la que se mostrará en el presente proyecto.

Las combinaciones de elementos son muy amplias, no existe una única estructura válida para un documento HTML sino que se genera una estructura en función de la aplicación.

Dentro del estándar HTML se ha escogido el estándar HTML5, a continuación se enumeran las nuevas características de éste nuevo estándar que tienen relación con el proyecto:

- *VIDEO*, esta etiqueta es una de las nuevas características de HTML5 y una de las más importantes, permite embeber vídeos dentro de una página web de manera nativa sin el uso de *plugins*.
- *NAV*, esta etiqueta declara un elemento de tipo *barra de navegación* en el cual se encuentra el menú con enlaces a otros tipos de recursos y secciones tanto dentro como fuera de la página. En nuestro caso esta etiqueta se ha utilizado para encapsular los botones de arranque/pare del código creado por el alumno y el botón que permite mostrar u ocultar la cámara del robot.
- *CANVAS*, permite la renderización de escenas gráficas a través de JavaScript. Es la etiqueta más importante dentro de nuestro proyecto ya que es la etiqueta que nos permite la creación de la escena en la cual tenemos nuestro robot simulado.

Existe una característica importante que afecta a todo el documento de HTML5 pero que, en general, no se está respetando en el desarrollo web y es que HTML5 tiene una tendencia semántica, es decir, las etiquetas como *SECTION*, *NAV* y *FOOTER* marcan claramente zonas dentro del documento HTML con el fin de poder conocer la estructura del documento de manera clara.

3.3. CSS

CSS o *Cascading StyleSheet* es un lenguaje que se usa para definir el aspecto visual de una página HTML. Su principal misión es la de separar la estructura y contenido del aspecto de la página HTML.

Con CSS podemos controlar incluso cómo se van a ver todos los documentos HTML de mi aplicación, es comúnmente utilizado por las empresas y diseñadores gráficos para crear de manera visual una identificación de la aplicación mediante tipos de letra, paleta de colores utilizada.

Deja atrás la gran necesidad de uso de JavaScript para fines de representación visual lo que hace que el rendimiento de la página se mejore al usar código JavaScript para otros fi-

nes. Además reduce la dependencia de software de edición gráfica como *Photoshop*, que sigue siendo utilizado pero su función es la edición más avanzada.

A continuación se muestra una imagen con una comparativa de la misma página web con y sin CSS.

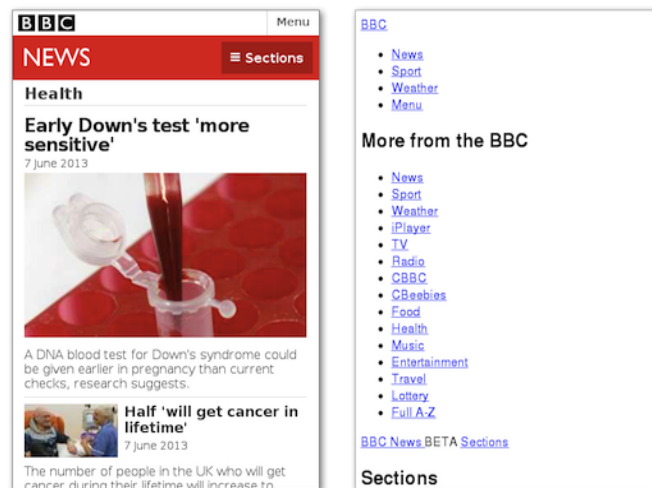


Figura 3.8: Imagen comparativa de la misma pagina web con y sin hojas CSS

Como vemos en la figura 3.8 las hojas de estilo CSS permite modificar completamente el aspecto de la pagina así como esconder menús y delimitar de manera visual las distintas partes de la página lo que permite mejorar la experiencia de usuario haciendo más accesible las partes importantes de la página.

Ésta última mención a la interfaz de usuario es importante ya que es uno de los puntos que más se cuidan en las empresas e incluso se hacen estudios para mejorar las interfaces , por ejemplo, debido al tamaño de los nuevos *smartphones* el menú de navegación ha cambiado su posición debido a que era difícil alcanzar la parte superior de la pantalla y por tanto se hacía molesto el uso de la aplicación.

3.4. AFRAME

AFRAME es un entornos web para la construcción de escenas de realidad virtual, se creó con la intención de facilitar la creación de contenido de realidad virtual. Es un entorno de código libre y tiene una de las comunidades de creadores de realidad virtual más grandes actualmente.

Soporta la mayoría de gafas de realidad virtual como *Vive*, *Rift*, *GearVR*, *etc.* además se puede usar no solo para realidad virtual sino para realidad aumentada. AFRAME fomenta la creación de escenas inmersivas completas de realidad virtual y va más allá de únicamente generar contenido en 360°, también implementa el uso de control de posición y controladores (mandos) que permiten interactuar con la escena, estos controles permiten al usuario tener una experiencia más inmersiva en la escena.

AFRAME además está soportado en los siguientes escenarios:

- Realidad virtual en aplicaciones de escritorio con *gadgets*.
- Realidad virtual en aplicaciones móviles con *gadgets*.
- Aplicaciones de escritorio convencionales.
- Aplicaciones de móvil convencionales.

A continuación se explican en subsecciones las características más importantes del entorno AFRAME.

3.4.1. Primitivas y HTML

AFRAME está basado en HTML y el DOM (*Document Object Model*), HTML es un lenguaje sencillo de leer y conocer la estructura, además no requiere de instalaciones únicamente se compone de texto y un navegador que muestre la página. AFRAME es compatible con la mayoría de entornos que se utilizan actualmente en el desarrollo web como pueden ser Vue.js, React, AngularJS y jQuery.

Crear escenas de realidad virtual de manera muy simple, como se ve en la figura 3.9 únicamente se necesita una etiqueta *script* que haga referencia al código del entorno y una etiqueta *a-scene* dentro del cuerpo del documento para crear una simple escena.

AFRAME ofrece un conjunto de elementos básicos para la escena llamados primitivas, estos elementos son figuras básicas como *cajas*, *esferas*, *cilindros*, *planos*, *cielo*, *etc.* AFRAME no solo ofrece figuras básicas, como hemos comentado anteriormente su intención es la de crear escenas inmersivas completas, por ello ofrece además etiquetas para la inyección de sonidos y vídeos dentro de la escena.

Este tipo de primitivas son bastante útiles para su uso en escenas simples, todas ellas heredan de la primitiva *a-entity* que, equiparandolo con HTML convencional, equivaldría con la etiqueta *div* del estándar HTML, la cual se utiliza de muchas maneras distintas.

Esta etiqueta *a-entity* representa por tanto el punto de partida de cualquier tipo de elemento de la escena que queramos crear al cual se le irán añadiendo *componentes* que le dotarán de cierta funcionalidad específica.

AFRAME permite además crear nuestras propias primitivas lo que nos permite seguir el principio de programación *DRY (Don't Repeat Yourself)* por el cual si tenemos un complicado elemento en la escena que está compuesto de varias entidades distintas no tenemos que copiar ese código *N* veces sino que podemos registrar la primitiva y hacer referencia a este elemento mediante el nombre de etiqueta que más convenga.

La figura 3.9 muestra todo el código necesario para crear la escena mostrada.

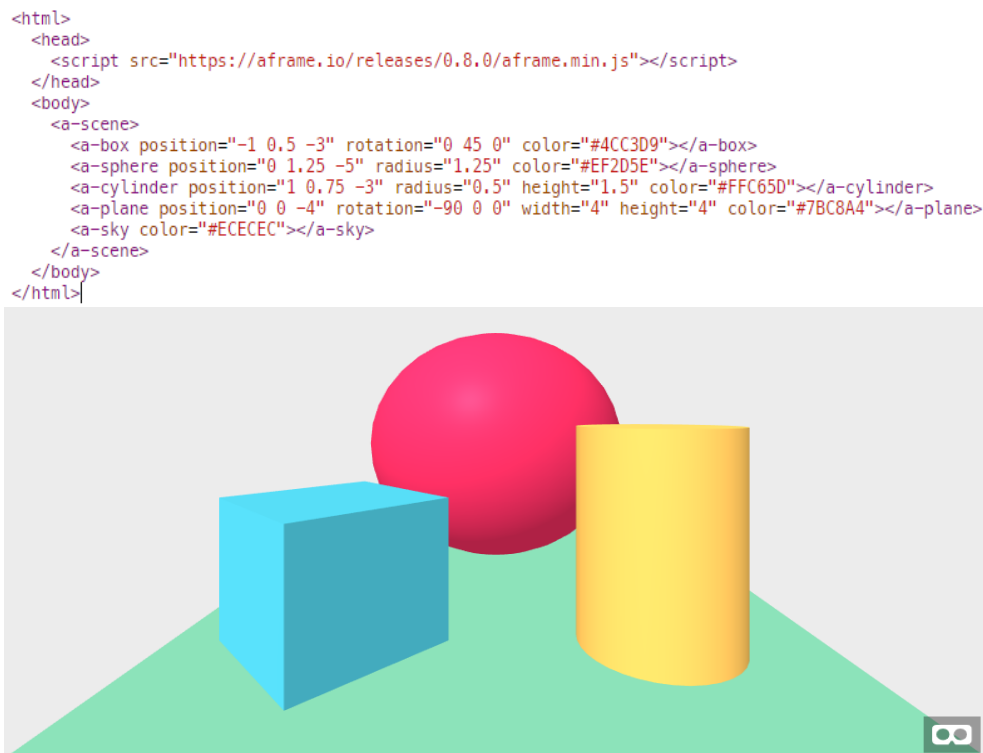


Figura 3.9: Ejemplo de código HTML que renderiza una escena básica de realidad virtual



Figura 3.10: La siguiente figura muestra etiquetas que serían el equivalente en AFRAME a un componente, el conjunto de varios componentes dan forma a la caja

3.4.2. Sistema Entidad-Componente

AFRAME se basa en el entorno three.js y provee una estructura reutilizable de entidad-componente en la que un componente puede ser utilizado en distintas entidades de distinta clase. Se pueden generar componentes personalizados y vincularlos a cualquier tipo de entidad dándole una funcionalidad distinta. Esto permite una gran flexibilidad a la hora de generar distintos integrantes en la escena con funcionalidades diferentes pero heredando todos de una misma entidad.

La arquitectura entidad-componente es común en el desarrollo 3D y en el desarrollo de videojuegos y sigue el principio de composición por herencia. Los beneficios de este tipo de arquitectura son:

- Gran flexibilidad a la hora de crear objetos debido a la reutilización de componentes y el mezclado de distintos componentes.
- Elimina el problema de largas cadenas de herencia, cada componente es independiente.
- Diseño limpio gracias al desarrollo por módulos.
- Es la manera más escalable de generar complejas escenas de realidad virtual.
- Permite reutilizar y compartir componentes no solo en un mismo proyecto sino con la comunidad de desarrolladores.

En la figura 3.10 se muestra un esquema del *Sistema Entidad-Componente* en el cual tenemos una figura final con forma de caja la cual estaría compuesta de varios componentes distintos. Estos componentes son: *Posición*, *Geometría*, *Material* y *Color*.

A continuación se mostrará la API (*Application Program Interface*) que ofrece AFRAME para implementar el *Sistema Entidad-Componente*:

- *Entidad*: se representa en AFRAME mediante la etiqueta *a-entity*.
- *Componentes*: se representa en AFRAME como atributos de la etiqueta HTML. Estos componentes son objetos que contienen un esquema, manejadores y métodos. Éstos se registran mediante el método `AFRAME.registerComponent(nombre, definición)`. A continuación se muestra un componente que imprime un mensaje en la consola del navegador.

```
AFRAME.registerComponent('log', {  
  schema: {  
    message: {type: 'string', default: 'Hello, World!'}  
  },  
  init: function(){  
    console.log(this.data.message);  
  }  
});
```

Figura 3.11: Componente básico que imprime por la consola del navegador el mensaje que se le pase por *message*

- *Sistema*: representado por la escena mediante la etiqueta *a-scene*. Los sistemas son similares a los componentes a la hora de definirlos, se registran mediante `AFRAME.registerSystem(nombre, definición)`.

Además como hemos comentado AFRAME tiene dos tipos de implementaciones que le dan características adicionales al sistema entidad-componente y es que al implementarse sobre HTML y JavaScript tenemos dos características importantes:

- Referenciar una entidad mediante el método *querySelector* implementado en JavaScript lo que permite acceder a una entidad por su ID, clase o atributos. En la figura 3.12 se muestra un ejemplo de código JavaScript que accede a un elemento de AFRAME con ID *rightHand*.
- Comunicación entre las entidades mediante eventos, esta característica es hereditaria del lenguaje utilizado lo que permite registrar y suscribir eventos que permite que los elementos en la escena no se conozcan entre sí.

```
var rightHandElement = document.querySelector("#rightHand");
```

Figura 3.12: Ejemplo de acceso a un elemento de AFRAME, como se aprecia se hace de la misma manera que se accede a cualquier elemento de HTML convencional

- Crear, eliminar y modificar atributos mediante el API del DOM, podemos utilizar los métodos `.setAttribute`, `.removeAttribute`, `.createElement` y `.removeChild` para modificar los elementos.

Por último pero no más importante, los componentes pueden hacer cualquier cosa, tienen acceso completo a *three.js*, *JavaScript* y *APIs Web* como pueden ser *WebRTC*, *AJAX*, *etc.*

3.4.3. Modelos 3-D

AFRAME ofrece la posibilidad de cargar modelos 3D más sofisticados en los formatos *glTF*, *OBJ*, *COLLADA*. Se recomienda el uso del formato *glTF* ya que es el modelo estándar para transmitir modelos 3D en la WEB. Los componentes se pueden escribir de manera que se pueda manejar cualquier tipo de formato que tenga un objeto en *three.js* para cargar el modelo.

Los modelos son archivos en texto plano y contiene vértices, caras, texturas, materiales y animaciones.

Como se ha repetido en varias ocasiones se trata de crear escenas, para ello es necesario implementar animaciones. Estas animaciones se implementan con el paquete de componentes creado por *Don McCurdy*, se puede localizar en <https://github.com/donmccurdy/aframe-extras/blob/master/src/loaders/animation-mixer.js>.

3.4.4. Herramientas de Desarrollo

AFRAME como hemos comentado se construye sobre JavaScript y HTML por tanto utiliza las mismas herramientas de desarrollo ya disponibles dentro del navegador. Además al crear escenas 3D se hace complicado depurar la escena y saber que todo está siendo representado en su posición correcta, para esta problemática AFRAME incluye un inspector visual que permite conocer la posición y los valores de los atributos para cada entidad de la escena. La figura 3.9 muestra la escena de nuestro simulador y los atributos de la cámara incluida dentro del robot, como se ve el inspector muestra el ángulo de visión de la cámara del robot y muestra los ejes

de la escena respectivo al punto en el que se encuentra de la cámara.

Ofrece una representación en árbol de la escena siguiendo la estructura del documento HTML, el inspector ofrece la posibilidad de mover, rotar, añadir y borrar elementos de la escena así como copiar la etiqueta una vez movida para usarla en nuestro documento HTML. Un ejemplo de esto sería mover nuestro robot y orientarlo de cara a la pelota verde, sin el inspector tendríamos que ir haciendo pruebas modificando manualmente el atributo de la posición dentro del documento HTML pero con el inspector visual basta usar las herramientas para mover el elemento y copiar las coordenadas que aparecen en la ventana de la derecha.

Por último el inspector permite hacer capturas de movimiento lo que permite:

- Test más rápidos, no se necesitan utilizar los *gadgets* cada vez que se quiera hacer un test lo que acelera mucho el desarrollo de la aplicación.
- Múltiples desarrolladores pueden utilizar el mismo *gadget*, puedes grabar el movimiento y dejar de usar el *gadget* para que otros desarrolladores del mismo proyecto puedan usarlo.
- Mostrar errores del código.

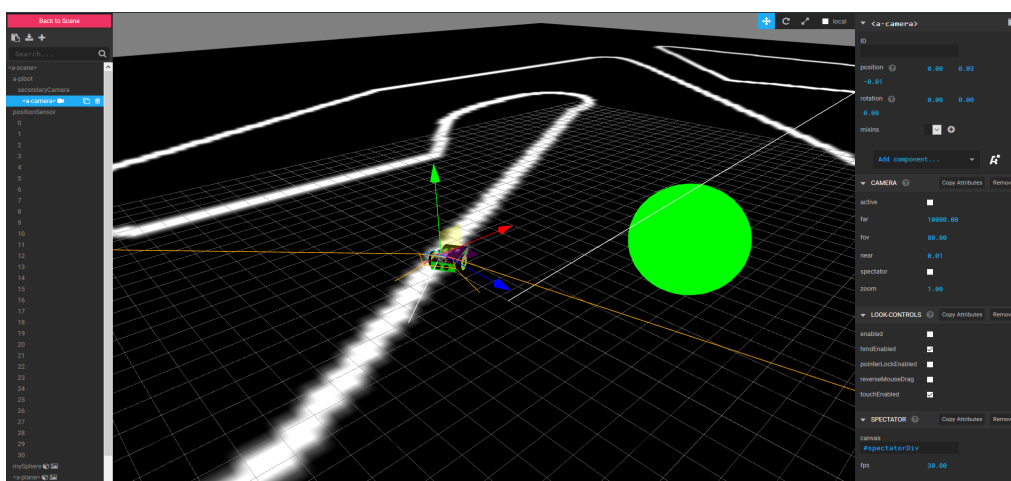


Figura 3.13: Inspector visual que ofrece el entorno *AFRAME*

3.5. ACE Editor

ACE Editor es un editor de código embebido creado en *JavaScript*, implementa las características de los editores nativos como *Sublime Text*, *Vim*, *etc.* ACE es el editor usado en el servicio *AWS Cloud9 IDE*.

El editor ofrece la siguiente funcionalidad:

- Mantener el estado de la sesión como por ejemplo el *scroll*, selección de texto, *etc.*
- Resaltado de sintaxis para la mayoría de lenguajes de programación como *JavaScript*, *CSS*, *Python*, *Java*, *etc.*
- Permite crear tus propias reglas de resaltado.
- Indentación automática del código.
- Manejo de ficheros grandes, maneja miles de líneas sin problema.
- Resaltado de paréntesis.
- *Drag and Drop* de texto dentro del editor de código.
- Comprobación de sintaxis del lenguaje, esta característica es bastante útil ya que nos permite descartar errores en ejecución debido a la sintaxis del programa lo que acelera el desarrollo.

La característica más importante para nuestro proyecto es sin duda la facilidad para embeberlo dentro de nuestra aplicación, simplemente hace falta una etiqueta *script* y un pequeño código para configurar su carga en la página, el detalle del código se mostrará en el capítulo 4. El editor además provee de una sencilla *API* para poder obtener el código escrito en el editor a través de nuestro programa para poder manejarlo.

En la figura a continuación se muestra las posibles configuraciones que puede adoptar el editor.

El editor utiliza el DOM (*Document Object Model*) para el renderizado, concretamente la etiqueta *canvas* y no depende de librerías externas. La característica más destacable de el editor es que no es necesario instalar nada tu ordenador, el editor reside completamente en la página web lo que permite la creación de aplicaciones interactivas como la del presente proyecto en la

```
selectionStyle: "line"|"text"
highlightActiveLine: true|false
highlightSelectedWord: true|false
readOnly: true|false
cursorStyle: "ace"|"slim"|"smooth"|"wide"
mergeUndoDeltas: false|true|"always"
behavioursEnabled: boolean
wrapBehavioursEnabled: boolean
// this is needed if editor is inside scrollable page
autoScrollEditorIntoView: boolean (defaults to false)
// copy/cut the full line if selection is empty, defaults to false
copyWithEmptySelection: boolean
useSoftTabs: boolean (defaults to false)
navigateWithinSoftTabs: boolean (defaults to false)
enableMultiselect: boolean # on by default
```

Figura 3.14: Posibles parámetros de configuración para el editor

cual podemos programar el robot 'en vivo' sin la necesidad de tener que exportar el código de nuestro editor para hacer pruebas en la simulación.

3.6. Blockly

Blockly es una librería que permite aprender programación mediante el uso de bloques visuales que se pueden combinar y traducir posteriormente a distintos lenguajes. Permite a los usuarios programar en distintos lenguajes sin tener que conocer la sintaxis del lenguaje al detalle.

Es una forma de aprender a programar orientada a estudiantes de temprana edad, el objetivo es que el alumno pueda aprender la lógica de los algoritmos de programación pero sin tener que aprender todo el lenguaje como puede ser declaración de variables, tipo y en general la sintaxis propia del lenguaje que según el tipo de lenguaje puede ser pesado al principio.

Blockly es la base de otros entornos de programación visual como Scratch 3 ya que es código libre lo que permite a cualquier desarrollador contribuir y utilizarlo en su propia aplicación. Ofrece métodos para importar y exportar el código de bloques además de métodos para modificar y personalizar el aspecto de los bloques para que sigan el estilo de la interfaz de la aplicación.

Es una tecnología orientada completamente al lado cliente y se puede utilizar mediante el fichero comprimido del repositorio oficial de *Blockly* llamado *blockly-compressed*, no utiliza

dependencias y como ya hemos comentado es código libre.

3.6.1. Generadores de código

Blockly como hemos comentado provee de generadores de código para los siguientes lenguajes:

- JavaScript.
- Python.
- PHP.
- Lua.
- Dart

Estos generadores proveen las herramientas básicas para crear funciones, expresiones lógicas, bucles, etc. La problemática de esto es que a veces nuestras aplicaciones necesitan usar la API de otras dependencias como en nuestro caso, para solventar esto *Blockly* permite generar bloques personalizados que se traducirán a la instrucción necesaria dotando de mucha flexibilidad al entorno.

Además *Blockly* permite añadir palabras reservadas dentro de cada tipo de lenguaje lo cual nos permite un control de colisiones con las variables propias de la aplicación ya que en lugar de eliminar esta variable lo que hace *Blockly* es renombrar todas las apariciones de la variable en el código generado dinámicamente.

Se puede apreciar que el código generado a través de los bloques será correcto en su sintaxis pero hay una problemática presente en programación y es la aparición de *bucles infinitos*, la librería dota de un método para intentar aplacar esta problemática, en la figura 3.15 se muestra un simple código que cuenta el número de iteraciones del bucle, no es el mejor método para solventar el problema ya que, como es nuestro caso, necesitaremos bucles de ejecución continua pero dota a la aplicación de control de ejecución de código.

```

window.LoopTrap = 1000;
Blockly.JavaScript.INFINITE_LOOP_TRAP = 'if(--window.LoopTrap == 0) throw "Infinite loop.";\\n';
var code = Blockly.JavaScript.workspaceToCode(workspace);

```

Figura 3.15: Implementación de un contador para evitar bucles infinitos al traducir lenguaje de *Blockly*

3.6.2. Bloques personalizados

Como se ha hecho referencia en la sección anterior, *Blockly* permite además de generar código en distintos lenguajes también crear bloques personalizados lo que permite generar código a través de bloques y conectarlos con cualquier tipo de API.

Para generar bloques personalizados *Blockly* hemos de configurar varios aspectos:

- Configuración de los parámetros de entrada y salida del bloque, conectores o parámetros en línea y color. La configuración del bloque se permite mediante dos formas distintas a través de un JSON o mediante JavaScript registrando un nuevo bloque en el objeto *Blockly*. La figura 3.16 muestra ambos métodos para generar el mismo bloque.

```

{
  "type": "string_length",
  "message0": 'length of %1',
  "args0": [
    {
      "type": "input_value",
      "name": "VALUE",
      "check": "String"
    }
  ],
  "output": "Number",
  "colour": 160,
  "tooltip": "Returns number of letters in the provided text.",
  "helpUrl": "http://www.w3schools.com/jsref/jsref_length_string.asp"
}

Blockly.Blocks['string_length'] = {
  init: function() {
    this.appendValueInput('VALUE')
      .setCheck('String')
      .appendField('length of');
    this.setOutput(true, 'Number');
    this.setColour(160);
    this.setTooltip('Returns number of letters in the provided text.\\n');
    this.setHelpUrl('http://www.w3schools.com/jsref/jsref_length_string.asp');
  }
};

```

Figura 3.16: Modos de configuración de un bloque personalizado en *Blockly*, como se puede apreciar la declaración de las distintas partes del bloque es bastante intuitiva y los parámetros son autodescriptivos.

- Configuración de la traducción del bloque a la instrucción de interés en los distintos lenguajes necesarios.
- Iniciar el bloque para que se renderice en el editor de bloques visual. La figura 3.17

```
Blockly.Blocks['move_combined'] = {  
  init: function() {  
    this.jsonInit(moveBlock);  
  }  
};
```

Figura 3.17: Muestra del código que inicia el bloque para que se muestre en el editor visual, *moveBlock* representa un objeto JSON con la configuración del bloque

Como se ve aunque los parámetros del JSON son autodescriptivos y sencillos de entender es complicado y lento generar un bloque desde cero por tanto *Google* ofrece unas herramientas para desarrolladores en línea lo que permite acelerar esta generación de código.

En la figura 3.18 se muestra una imagen del entorno de creación de bloques personalizados que provee *Blockly* en el cual se puede observar en la parte de la derecha el archivo en formato *JSON (JavaScript Object Notation)* de configuración del bloque en el que se definen las entradas que toma, el color con el que se mostrará entre otra serie de parámetros. Además, muestra la función con la configuración básica para obtener los parámetros que se utilizaran para generar código real JavaScript.

Como se ve en la figura 3.18 en la parte superior derecha la herramienta te muestra cómo se verá el bloque final en tu aplicación bajo la configuración visual por defecto que trae *Blockly*

3.6.3. Menú de bloques, *Toolbox*

El editor de *Blockly* provee además de una barra de herramientas en la cual se muestran los bloques que podrán ser usados a través del editor, estos bloques se configuran a través de un fichero XML en el cual podemos tener distintos tipos de etiquetas que se muestran a continuación.

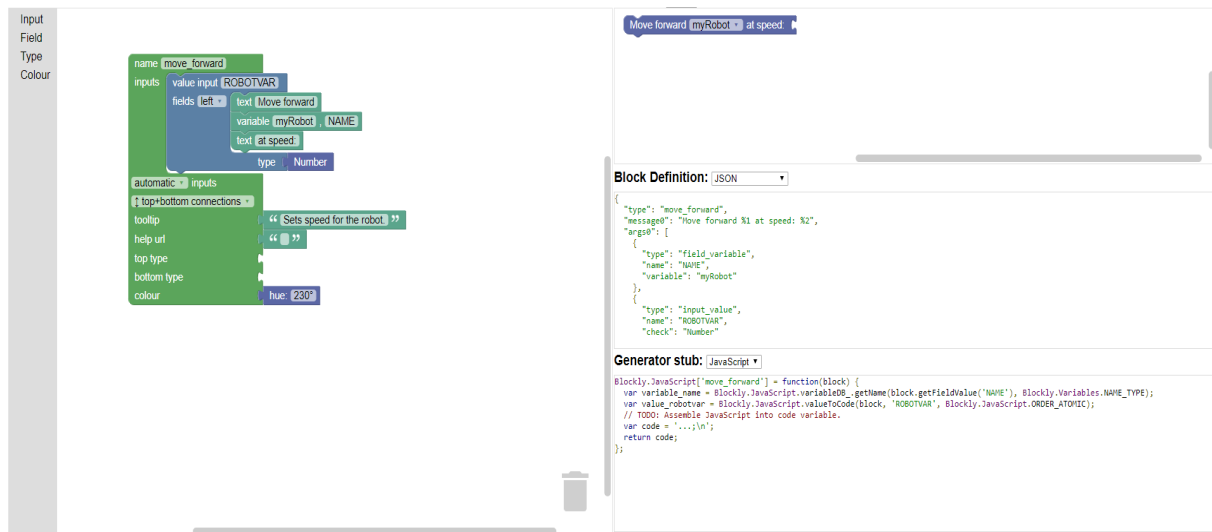


Figura 3.18: Herramienta de desarrollo para generación de bloques personalizados.

```
<xml id="toolbox" style="display: none">
  <category name="Variables" custom="VARIABLE"></category>
  <category name="Text" colour="{BKY_MATH_HUE}">
    <block type="text"></block>
  </category>
</xml>
```

Figura 3.19: Ejemplo de las etiquetas posibles de configuración de la *toolbox* del editor

Como vemos en la figura 3.19 tenemos tres bloques distintos, la raíz del XML marcada por la etiqueta *xml*, la etiqueta *category* que nos permite hacer divisiones de bloques por tipos distintos, en la imagen tenemos dos categorías *variables* y *texto*, la categoría *variables* como vemos no tiene bloques dentro ya que se generan dinámicamente. En la categoría *texto* vemos declarado un bloque de tipo *text* que representaría un *String* en lenguaje JavaScript. Como se puede apreciar la configuración del menú de bloques no es fija, se puede crear distintas categorías en función del diseño de la interfaz de usuario.

3.7. jQuery

jQuery es una librería multiplataforma de *JavaScript*, creada inicialmente por John Resig y permite simplificar la forma de interactuar con los documentos HTML, el DOM, manejo de eventos, desarrollo de animaciones y agregar interacción ligera entre el cliente y servidor con el mecanismo *AJAX* (*Asynchronous JSON And XML*). Es una biblioteca ampliamente utilizada

debido a las características que ofrece y sigue la filosofía ' *Code less, do more* '.

jQuery es software libre de código abierto, posee una licencia doble MIT y una licencia pública general de GNU v.2 lo que permite su uso en proyectos libres y privados. *jQuery* se basa en simplificar funcionalidad que se repite a menudo en las páginas web como por ejemplo manejo de evento 'click' de un botón, ocultar o mostrar un elemento del DOM, etc.

Basados en *jQuery* existen una gran cantidad de plugins (extensiones) gratuitos y de pago que permiten disminuir el tiempo de desarrollo de la interfaz de usuario como por ejemplo hacer que tu pagina web sea *responsive* (que el contenido se adapte bien al tipo de dispositivo y navegador), crear una galería de fotos, carrusel de imágenes, etc.

Una de las características de *jQuery* más importante es su facilidad de uso, la curva de aprendizaje de *jQuery* es sencilla ya que como hemos comentado ofrecen métodos para manejo incluso de CSS. La figura 3.20 muestra cómo se hace un efecto de '*fundido*' con *jQuery*, como se ve en la figura el método puede tomar parámetros de entrada para regular la velocidad a la que se quiere realizar el efecto.

```
$( "div:hidden:first" ).fadeIn( "slow" );
```

Figura 3.20: Instrucción para hacer un efecto de '*fundido*' en *jQuery*.

En la parte de la derecha de la instrucción tenemos el método a ejecutar y la velocidad, en la parte de la izquierda tenemos el acceso completo al dom para un elemento en concreto.

Para hacer este tipo de efectos es necesario conocer bien el uso del lenguaje CSS, *jQuery* permite a desarrolladores novatos en el campo de maquetación de interfaces de usuario hacer que sus páginas web sean algo más elaboradas en cuanto a efectos e interfaz se refiere sin la necesidad de conocimiento profundo de CSS3.

Actualmente en el punto en el que se encuentra el desarrollo web *jQuery* ha disminuido su uso debido a la aparición de entornos como *React*, *VueJS* y *AngularJS* que implementan funcionalidad similar a la de *jQuery* pero además permiten implementación de patrones de diseños como el *MVC (Modelo Vista-Controlador)*. Pero este tipo de modelos no son necesarios en el proyecto que nos concierne de ahí la elección de *jQuery* por encima de estos otros entornos.

3.8. NPM y Webpack

NPM es la abreviatura de (Node Package Management), es una tecnología de gestión de dependencias del entorno NodeJS lo cual permite la simplificación de instalación de las dependencias de un determinado software.

La declaración de dependencias de la aplicación se hace en el fichero *package.json* en el cual se pone los distintos paquetes NPM de los que hará uso nuestra aplicación tanto para un entorno de desarrollo como para un entorno de producción. Haciendo referencia a esto mencionamos WebPack que es una herramienta de empaquetado de aplicaciones lo que permite generar un *bundle* con todo lo necesario de nuestra aplicación haciendo que el uso de nuestra aplicación en el HTML se simplifique necesitando únicamente una etiqueta *script* que referencie a nuestro *bundle*.

La instalación con NPM es simple, basta con moverse al directorio en el que se encuentra nuestro fichero *package.json* y ejecutar *npm install*, esto descargará todas las dependencias de nuestra aplicación e incluso las dependencias de nuestras dependencias (si existieran) y las instalará bajo la carpeta *node-modules*. La referencia en el nombre de *Node* es debido a que el entorno *Node* tiene una simplificación para hacer uso de las librerías en la carpeta *node-modules* mediante la instrucción *require* la cual busca dentro de la carpeta la que se llame igual que la que pasamos como parámetro a la función.

Para la utilización de estas librerías en el lado cliente tenemos que hacer uso de ES6 y la instrucción *import*. Ésto unido con *Webpack* permite que todas las dependencias en el lado cliente (imports) se junten en un único fichero.

Capítulo 4

Simulador robótico web: *WebSim*

En este capítulo se muestra la estructura diseñada para el simulador robótico web y se detallan sus partes explicando la funcionalidad de cada una de ellas. El simulador se ha llamado *WebSim*.

4.1. Diseño

Como se comentó en el capítulo 1, el objetivo del presente proyecto es crear una aplicación para simular robots en el entorno *AFRAME* permitiendo conectar un editor de código para ejecutar instrucciones con el robot simulado. El primer componente de esta aplicación es el propio simulador robótico. En la figura 4.1 se muestra el diseño que sigue el simulador *WebSim*. A continuación se explicará cada una de las partes por separado y la comunicación existente entre ellas.

La arquitectura sigue un diseño por módulos, la intención es poder conectar distintos tipos de entradas a *WebSim* como puede ser un editor externo de código en JavaScript, un editor de código en el lenguaje visual Blockly, una aplicación externa vía comunicaciones ICE, etc. sin tener una aplicación monolítica en la que todas las partes estén completamente acopladas.

WebSim tiene cinco funcionalidades principales:

- Registra los tres componentes constituyentes del robot en *AFRAME*. Para la simulación del robot en el entorno ha sido necesario crear tres componentes llamados *followBody*,

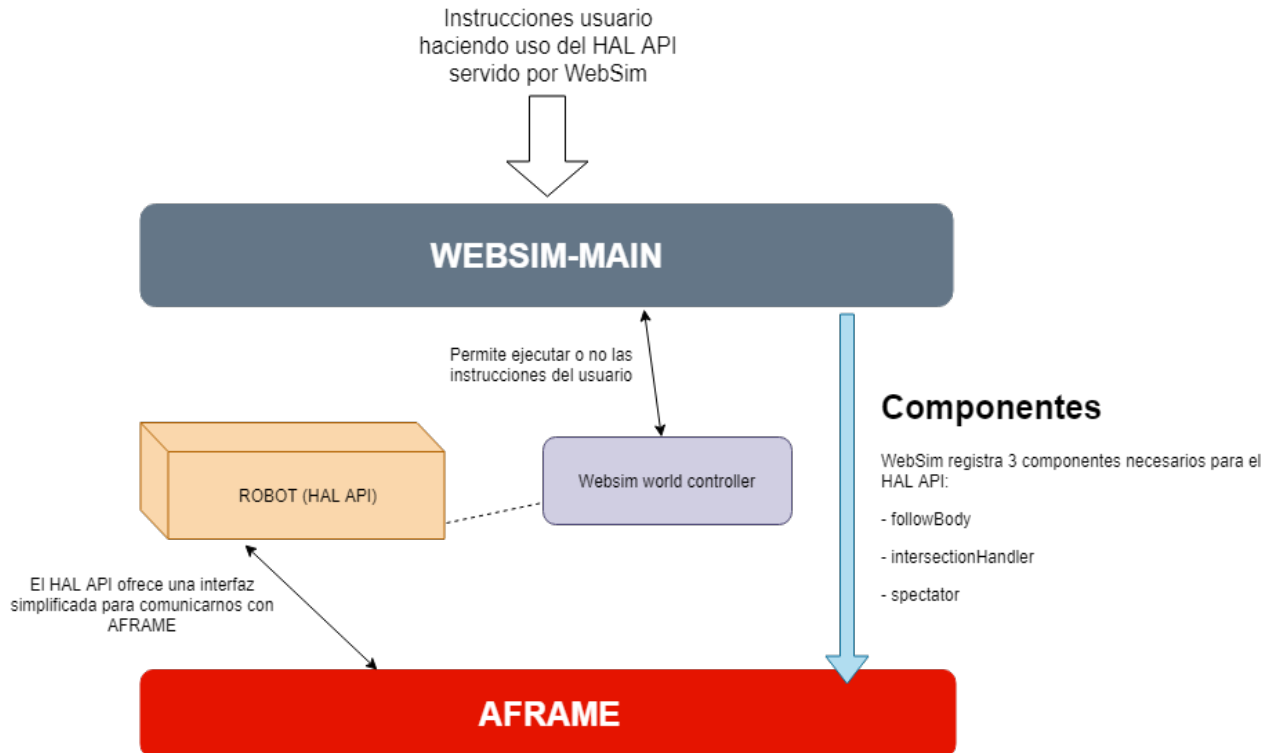


Figura 4.1: Estructura de WebSim.

spectatorComponent, *intersectionHandler* que se encargan de simular una cámara contenida en el robot, manejar los eventos de intersección de láseres en el robot simulado y anclar al cuerpo del robot simulado un elemento respectivamente.

- Ofrece la interfaz de programación en JavaScript para manejar el robot simulado, ofrece un interfaz en lenguaje JavaScript, llamada *HAL API (Hardware Abstraction Layer)* lo que simplifica mucho el uso del robot creado en AFRAME. Se le pueden enviar instrucciones que el robot ejecutará mediante una única línea de código, lo que permitirá al usuario crear una lógica para el robot de manera limpia, sin tener que comunicarse directamente con el motor AFRAME ya que de eso se encarga WebSim.
- Ofrece la instancia que contiene el objeto robot (*myRobot*). De manera que el usuario no tiene que instanciar ningún tipo de variable de la clase *RobotI* ya que se le ofrece para el uso directo. Esto conlleva una doble funcionalidad. Primero, simplifica el uso del simulador aún más, ya que el usuario solo tiene que enviar instrucciones haciendo uso del objeto *myRobot*. Y segundo, evita la creación de dos instancias del mismo objeto que

llevaría a una incorrecta ejecución.

- Controla la ejecución del *mundo*, es decir, permite arrancar o pausar la simulación del robot en sí misma. Se muestra en la figura 4.1 mediante el bloque *websim world controller*. Una de las funcionalidades del control del entorno es evitar que el usuario cambie el valor de la variable *myRobot* lo cual haría que se perdiese el objeto robot y tendría que refrescar la página.
- Permite la conexión del robot simulado con software externo mediante intercambio de mensajes. De este modo el software externo al navegador web puede acceder a los sensores y actuadores del robot simulado y gobernar su comportamiento. El software externo por tanto se encargará únicamente de pedir datos de los sensores simulados y acceder a los actuadores simulados. Es un segundo interfaz de programación del robot simulado, no ya como funciones JavaScript sino como mensajes.

Con este diseño el simulador *WebSim* provee de una capa de simplificación y abstracción para los usuarios de modo que ellos solo se tienen que concentrar en generar un código para ejecutar una serie de instrucciones en el robot accediendo de forma simple a los distintos sensores como son la cámara, sensores de ultrasonido, sensores infrarrojos y a los motores.

Principalmente el usuario se encargará de programar la lógica del robot, es decir, el pensamiento (código) para resolver un ejercicio determinado con el objetivo de aprender programación de robots.

4.2. Simulación con AFRAME

En esta sección se explica cómo se ha creado la simulación 3D de nuestra escena con el entorno *AFRAME*.

Como hemos explicado en la sección 3.4, *AFRAME* simplifica en gran parte la generación de escenas en el navegador, se monta sobre *WebGL* y *three.js* y permite crear la escena y sus integrantes (entidades) haciendo uso únicamente de etiquetas HTML. En la Figura 4.2 se muestra el esquema HTML necesario para generar nuestra escena simulada junto con el robot.

El mundo simulado, es un conjunto de elementos con atributos y el motor *AFRAME* se encarga de su funcionamiento tridimensional.


```

<a-scene id="scene"
  background="color: gray;"
  stats
  embedded physics="debug: true">

  <a-assets>
    <a-asset-item id="model-pibot"
      src="assets/models/jrobotF.dae">
    </a-asset-item>
    
  </a-assets>
  <!-- Pibot, which body is the asset item with ID = model-pibot -->
  <a-entity dynamic-body="mass: 50;" id="a-pibot"
    collada-model="#model-pibot"
    scale="20 20 20"
    rotation="0 -149.63 0"
    position="0 0.1 -6.0">
    <a-entity id="secondaryCamera"
      position="0 0 0"
      rotation="-20 -90 0">
      <a-camera position="0 0.03 -0.01"
        spectator="canvas:#spectatorDiv;"
        active="false"
        wasd-controls-enabled="false"
        look-controls-enabled="false">
      </a-camera>
    </a-entity>
  </a-entity>
  <a-entity id="positionSensor"></a-entity>

  <!-- Scenario -->
  <a-sphere id="mySphere"
    class="collidable"
    dynamic-body="mass:10000;"
    scale="1.1 1.1 1.1"
    position="2.75 0.01 -2.27"
    rotation="0 90 0"
    color="#00ff00">
  </a-sphere>
  <a-plane static-body position="0 0 0"
    rotation="-90 0 0"
    width="100"
    height="100"
    repeat="1 1"
    src="#ground">
  </a-plane>

  <!-- Illumination -->
  <a-light type="ambient" color="white"></a-light>

  <a-entity id="primaryCamera" position="0 8 6" rotation="-45 0 0">
    <a-camera fov="100"></a-camera>
  </a-entity>

</a-scene>

```

Figura 4.2: HTML que genera la escena principal del simulador

Como se ha comentado en el capítulo 3, la etiqueta *a-scene* (Figura 4.3) es la etiqueta

```
<a-scene id="scene"  
  background="color: gray;"  
  stats  
  embedded physics="debug: true">
```

Figura 4.3: Etiqueta *a-scene*, es la etiqueta principal del entorno *AFRAME*.

principal utilizada en *AFRAME* para generar la visualización. Como queremos una escena simple le hemos dado un color de fondo gris y hemos activado las físicas y las mediciones. Las físicas permiten hacer uso del motor de físicas en el paquete *aframe-physics* en el cual existe la gravedad, rozamiento y otros tipos de fuerzas haciendo que la simulación sea más realista. La etiqueta *stats* (mediciones) permite conocer las métricas relevantes de la escena como los *FPS* (Fotogramas por segundo), el número de vértices que está manejando la escena, las texturas cargadas en la escena y los *RAF* (*Request Animation Frame*) que es la latencia de la escena. Estas medidas aparecen en rojo cuando el valor no es el adecuado para cada una de ellas, permiten al desarrollador medir la eficiencia de su escena y del código bajo ésta.

```
<a-assets>  
  <a-asset-item id="model-pibot"  
    src="assets/models/jrobotF.dae">  
  </a-asset-item>  
    
</a-assets>
```

Figura 4.4: *a-assets* es la etiqueta usada por *AFRAME* para la gestión de texturas y recursos externos.

Las etiquetas *a-assets* (Figura 4.4) son un modo de *AFRAME* de gestionar de manera más eficiente las texturas y modelos 3D a usar en las entidades de la escena. Podemos usar la etiqueta *a-asset-item* con cualquier tipo de archivo de entrada, llama al cargador de ficheros de *three.js* y 'avisa' del estado de la carga del archivo, tiene tres estados:

- *Error*: Fallo al cargarlo.
- *Progress*: lo emite cuando está en proceso de carga del archivo, devuelve un evento en el

cual en el campo *detail* tenemos un objeto de tipo *XMLHttpRequest* con la cantidad de bytes cargados en total.

- *Loaded*: Indica que el archivo se ha cargado completa y correctamente.

En el mundo simulado de *WebSim* se tiene: el cuerpo del robot, sus sensores de distancia, su cámara y la escena que lo rodea. Todos estos objetos se detallan en las secciones siguientes.

4.2.1. Cuerpo del robot

Como se ha explicado en el capítulo 3, el 'objeto' básico en una escena en *AFRAME* es *a-entity* que es un objeto vacío al que le podemos conectar y dar la funcionalidad que requiramos. Haciendo uso de esta etiqueta se ha simulado el robot *Pibot*, que se compone de ruedas con motores, sensores y cámara. La Figura 4.5 muestra las etiquetas y atributos utilizados para la configuración inicial en la simulación del robot.

```
<a-entity dynamic-body="mass: 50;" id="a-pibot"
  collada-model="#model-pibot"
  scale="20 20 20"
  rotation="0 -149.63 0"
  position="0 0.1 -6.0">
  <a-entity id="secondaryCamera"
    position="0 0 0"
    rotation="-20 -90 0">
    <a-camera position="0 0.03 -0.01"
      spectator="canvas:#spectatorDiv;"
      active="false"
      wasd-controls-enabled="false"
      look-controls-enabled="false">
    </a-camera>
  </a-entity>
</a-entity>
<a-entity id="positionSensor"></a-entity>
```

Figura 4.5: Etiquetas utilizadas para la simulación del robot en el entorno *AFRAME*.

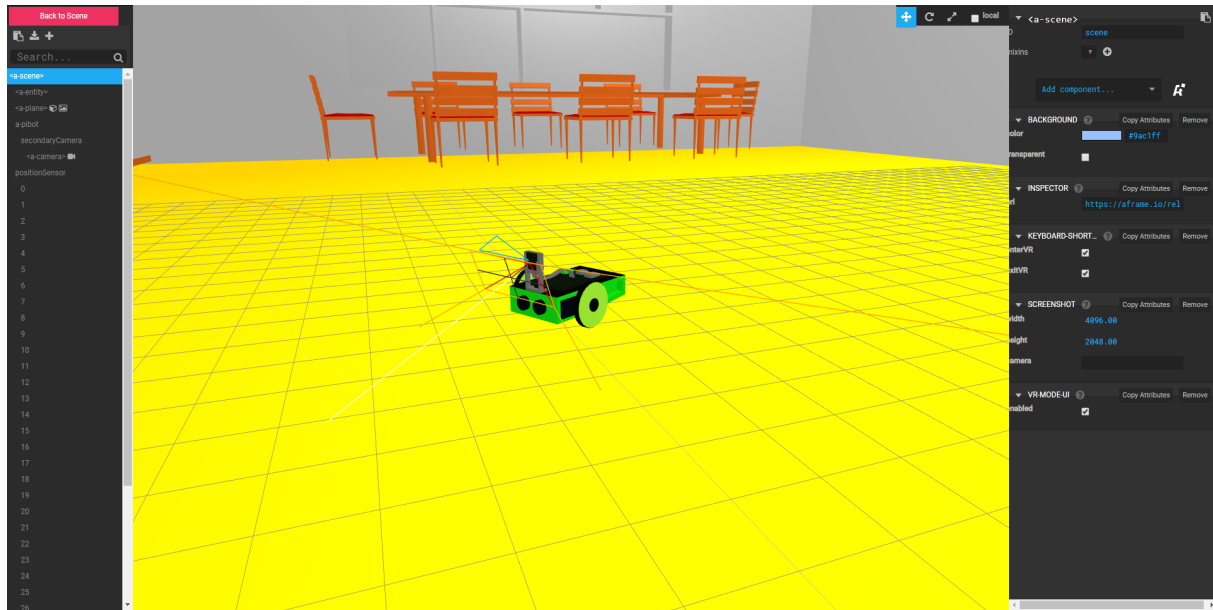


Figura 4.6: Robot simulado en la escena, se resalta la cámara integrada en él.

Para la simulación del cuerpo del robot hemos utilizado el atributo *collada-model* que permite hacer uso de un modelo 3D en formato *.dae* y anclarlo a nuestra entidad de la escena. Se pueden usar otro tipo de formatos pero se ha elegido éste ya que es el que mejor rendimiento daba al cargarlo en la escena. Además, se ha utilizado el atributo *dynamic-body* que dota de físicas al elemento con la particularidad que entonces el elemento puede ser movido por otros.

4.2.2. Sensores de distancia

El robot tiene sensores de obstáculos, para los que se ha utilizado el atributo *raycaster* que se encapsula dentro de la entidad con el identificador *positionSensor* (Figura 4.8) junto con el componente *followBody* que se muestra en la figura 4.9. Estos sensores de distancia a obstáculos representan a sensores reales de ultrasonidos o de láser. La carga y registro (Figura 4.7) de estos elementos se hace desde WebSim. Es necesario hacerlo así para poder modificarlos dinámicamente y añadir más o menos sensores de distancia al robot en tiempo de ejecución. Se ha utilizado un array de sensores de distancia que apuntan a distintas direcciones desde el cuerpo del robot.

```

AFRAME.registerComponent('spectator', spectObject);
AFRAME.registerComponent("intersection-handler", intersectionHandlerObj);
AFRAME.registerComponent("follow-body", followBodyObj);

```

Figura 4.7: Registro de los tres componentes necesarios para el Robot simulado.

```

<a-entity id="positionSensor"></a-entity>

```

Figura 4.8: Etiqueta vacía utilizada para encapsular los sensores de distancia a obstáculos.

```

<a-entity raycaster line follow-body intersection-handler class="center" id="0"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="1"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="2"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="3"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="4"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="5"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="6"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="7"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="8"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="9"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="10"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="11"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="12"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="13"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="14"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="15"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="right" id="16"></a-entity>
<a-entity raycaster line follow-body intersection-handler class="left" id="17"></a-entity>

```

Figura 4.9: Etiquetas generadas por *WebSim* para la simulación del sensor de ultrasonido.

4.2.3. Cámara

La cámara, como se ve en la Figura 4.6 es un elemento hijo encapsulado dentro de una entidad. Esto se hace así debido a las recomendaciones de *AFRAME* que indican que para modificar la posición de la cámara en tiempo de ejecución se incluya ésta dentro de una entidad. La etiqueta cámara cuenta con atributos entre los que destacan dos: *spectator* y *wasd-controls-enabled*.

- *Spectator*: Este atributo ha sido creado específicamente para la aplicación y permite imprimir el contenido de la cámara en una etiqueta *canvas* dentro de la página HTML lo que permite mostrar lo que está viendo el robot en tiempo real.
- *WASD-Controls-Enabled*: Permite declarar si la cámara se podrá mover con las teclas

W-A-S-D, en nuestro caso no interesa ya que modificaríamos la posición de la cámara respecto del robot estropeando así la escena.

En la Figura 4.6 se muestra el robot simulado y una muestra del foco de visión que está tomando la cámara del robot en la escena, es decir, lo que está viendo el robot.

4.2.4. Escena

En esta subsección se explica qué etiquetas se han utilizado para la generación de los elementos que intervienen en la escena pero que no forman parte del cuerpo del robot. En la Figura 4.10 se muestran las etiquetas y atributos utilizados para el resto de elementos de la escena.

```
<a-entity id="positionSensor"></a-entity>

<!-- Scenario -->
<a-sphere id="mySphere"
  class="collidable"
  dynamic-body="mass:10000;"
  scale="1.1 1.1 1.1"
  position="2.75 0.01 -2.27"
  rotation="0 90 0"
  color="#00ff00">
</a-sphere>
<a-plane static-body position="0 0 0"
  rotation="-90 0 0"
  width="100"
  height="100"
  repeat="1 1"
  src="#ground">
</a-plane>

<!-- Illumination -->
<a-light type="ambient" color="white"></a-light>

<a-entity id="primaryCamera" position="0 8 6" rotation="-45 0 0">
  <a-camera fov="100"></a-camera>
</a-entity>
```

Figura 4.10: Etiquetas utilizadas para crear diferentes elementos de la escena del simulador *WebSim*.

La etiqueta *a-sphere* declara la existencia de una esfera en nuestra escena, los atributos

declaran las siguientes propiedades de la esfera, es de la clase '*collidable*'. Esto se utiliza para indicar al robot que los sensores de ultrasonido van a detectar intersecciones con este tipo de objetos. La esfera también tiene físicas aplicadas y tiene una masa de 1000kg.

La etiqueta *a-plane* indica la existencia de un plano horizontal. Este plano tiene físicas pero es un cuerpo estático, es decir, no puede ser movido por otros objetos de la escena. Además, se declara el tamaño del plano, la textura a cargar en el plano que en este caso será un circuito en blanco y negro y el número de veces que se repite a lo largo de los ejes X y Z. Como se indica que solo se repita 1 vez en ambos ejes entonces *AFRAME* estira la textura hasta ajustarla con al tamaño del plano.

Por último tenemos dos etiquetas, *a-light* que indica que se va a utilizar una luz en la escena de tipo ambiental de color blanco y la etiqueta *a-entity* que declara una entidad vacía que, como se ha explicado antes, se usa para posicionar la cámara observadora principal desde la que se ve la escena.

4.3. Drivers del robot

El robot consta de varios sensores y actuadores, simulados por *AFRAME*. Los drivers permiten a código externo al simulador acceder a los sensores y actuadores simulados a través de un interfaz de programación llamado *HAL-API (Hardware Abstraction Layer)*. Están escritos en JavaScript y también materializan la simulación del robot (por ejemplo su movimiento) manejando la representación web del robot en *AFRAME*.

Por ejemplo, los sensores implementados en el robot recogen una serie de datos que han de ser guardados de manera interna en el robot para que el usuario pueda acceder a ellos y programar una lógica. Para ello se crean una serie de métodos en JavaScript que se conectan con el robot simulado en *AFRAME* y se crean variables internas para guardar los datos y servirlos mediante una instrucción simple.

En el robot dentro de *WebSim* tenemos dos hebras independientes, la hebra de los motores y la hebra de obtención de datos de la cámara. Son funciones autónomas periódicas y extraen o añaden datos en otras variables internas del robot, lo que permite hacer uso de estos datos de

manera asíncrona.

4.3.1. Constructor

Esos métodos JavaScript y variables internas mencionados se materializan como un objeto de la clase *RobotI*. El constructor de la clase *RobotI* es un método obligatorio en *JavaScript*, es el primer método al que se llama una vez se crea una instancia del objeto. El constructor de nuestro objeto toma como parámetro de entrada un *string* con el identificador de la etiqueta HTML (etiqueta del entorno AFRAME) en la cual tenemos el robot simulado. Gracias a esto se puede hacer uso desde *JavaScript* de las propiedades que ofrece el entorno AFRAME, como la posición en la escena, añadir o eliminar elementos en la escena, obtener la imagen de la cámara, etc.

Además de acceder al robot simulado de AFRAME, el constructor inicia una serie de variables internas de configuración del robot (*this* hace referencia al contexto del objeto robot):

- *defaultDistanceDetection*: Es una variable de configuración de los sensores de ultrasonido que declara la distancia máxima a la cuál los sensores detectarán un objeto. Se ha elegido hasta 10 metros.
- *defaultNumOfRays*: Es una variable de configuración de los sensores de distancia a obstáculos. Declara el número de rayos que se simularán para detectar objetos. El arco que abarcamos es 180° por lo tanto cuanto mayor sea el número más precisa angularmente será la detección de objetos.
- *this.robot*: Es una variable que permite crear un enlace entre la abstracción del objeto robot y el robot simulado en AFRAME y permite acceder a los distintos métodos que ofrece AFRAME por defecto.
- *this.activeRays*: Es una variable de control de tipo *boolean* que permite saber si los láseres del sensor de ultrasonidos están activos o no. Esta variable permite hacer un apagado o encendido de los láseres en caliente, cosa que no suele ser habitual en robótica pero ya que nos encontramos en un entorno de simulación puede ser interesante la posibilidad de

apagar estos láseres con el fin de reducir la cantidad de código a ejecutar por el robot y hacer así que el rendimiento mejore.

- *this.distanceArray*: Se trata de un objeto *JavaScript* que contiene tres variables de tipo *Array* en las cuales se guardan las distancias que detectan los sensores de ultrasonidos agrupados en tres conjuntos; centro, izquierda y derecha lo que permite conocer la ubicación del objeto que se está detectando.
- *this.understandedColors*: Es una variable interna del robot que permite asociar un color de entrada como tipo *string* a sus valores de filtros de color RGB (Red-Green-Blue) para detectarlo mediante el uso de herramientas de OpenCVjs. Esto permite simplificar el uso de la detección de objetos mediante el nombre de color para usuarios que no tengan conocimientos en visión artificial. Actualmente se implementan las componentes primarias y el color blanco.
- *this.velocity*: Es una variable de configuración de la velocidad inicial del robot en los distintos ejes (por defecto cero en todos ellos). Esta variable es importante ya que a la hora de programar la lógica del robot es necesario conocer la velocidad actual. Esto permite tenerla guardada para su uso tanto por los motores como por los usuarios. Tenemos 3 velocidades distintas en función de cada eje (X-Y-Z). En el eje X tendremos la velocidad en el plano horizontal, la velocidad lineal. De manera simplificada sería la velocidad a la que se mueve el robot en la dirección en la que mira. En el eje Y tendremos la velocidad en la cual se eleva el robot (debido al sistema de coordenadas tomado en AFRAME donde Y es la altura). Esta velocidad no es utilizada en la implementación actual del robot. En el eje Z tendremos la velocidad de giro.

Por último, el constructor del robot llama a los métodos de arranque de motores, cámara y sensor de ultrasonido que se explicarán más detalladamente en sus correspondientes subsecciones a continuación.

La figura 4.11 muestra el código del constructor del robot. Como se ve, la variable *this.understandedColors* es una simplificación de los filtros a aplicar en una imagen para detectar un objeto con dicho color.

```

constructor(robotId){
    const defaultDistanceDetection = 10;
    const defaultNumOfRays = 31;

    this.myRobotID = robotId;
    this.robot = document.getElementById(robotId);
    this.activeRays = false;
    this.raycastersArray = [];
    this.distanceArray = {
        center: [],
        left: [],
        right: []
    };
    this.understandColors = {
        blue: {low: [0, 0, 235, 0], high: [0, 0, 255, 255]},
        green: {low: [0, 235, 0, 0], high: [0, 255, 0, 255]},
        red: {low: [235, 0, 0, 0], high: [255, 0, 0, 255]},
        white: {low: [230, 230, 230, 0], high: [255, 255, 255, 255]}
    };
    this.velocity = {x:0, y:0, z:0, ax:0, ay:0, az:0};
    this.motorsStarter(this.robot)
    this.startCamera();
    this.startRaycasters(defaultDistanceDetection, defaultNumOfRays);
}

```

Figura 4.11: Código del constructor del objeto robot, la variable *this* hace referencia al contexto.

4.3.2. Driver de motores

La función principal de los motores es permitir el movimiento del robot en el plano horizontal a la velocidad configurada por el usuario, por ejemplo mediante el método `setV`. Además, un objetivo secundario para esta interfaz es que los motores funcionen de manera autónoma, es decir, que no tengamos que enviar constantemente instrucciones al robot para que este se mueva sino que al configurar la velocidad al robot este se mueva de manera autónoma hasta nueva orden. Se tienen unas variables internas de velocidades comandadas que se consultan para materializar el movimiento simulado y se actualizan desde los métodos del HAL API a través de los cuales se ordenan comandos de movimiento.

Inicialmente se llama desde el constructor del objeto robot y, una vez arrancado, los pasos que ejecuta son los siguientes:

- Se obtiene la orientación actual, esto se hace debido a que, como el robot se encuentra

en un sistema de coordenadas, es necesario saber hacia donde mira el robot para poder moverlo hacia adelante y atrás correctamente.

- Cálculo de la nueva posición. Se calcula la nueva posición en función de la velocidad lineal configurada en ese instante y en función del vector de vista (hacia dónde mira el robot). Para calcular dicha posición es necesaria por tanto la orientación, la velocidad lineal y la posición actual del robot. Se ha utilizado una descomposición de vectores en sus componentes en el eje X-Z para calcular la nueva posición

```
let x = velocity.x/10 * Math.cos(rotation.y * Math.PI/180);  
let z = velocity.x/10 * Math.sin(rotation.y * Math.PI/180);  
  
robotPos.x += x;  
robotPos.z -= z;
```

Figura 4.12: Cálculo de la nueva posición en función de la orientación y la velocidad lineal.

- Se establece la nueva posición para el objeto en la escena. En cada iteración se establece la nueva posición previamente calculada en la escena y la velocidad angular del robot.
- Se establece un temporizador para que la función se llame así misma. Esto se hace para que la función se invoque así misma constantemente y no tener que estar enviando continuamente instrucciones de velocidad, lo que simplifica el código.

Gracias a esta última instrucción nativa de *JavaScript* se puede crear una función iterativa. Con ello el modo de usar los motores se simplifica: el usuario únicamente llama a la función *setV* que guarda en la variable interna *this.velocity* la velocidad pasada como parámetro a la función y será la función iterativa del motor la que se encargará de comprobar periódicamente este registro para saber cuál es la velocidad especificada por el usuario. Esto permite simular una hebra dentro del navegador web y conjuga el funcionamiento típico de eventos del entorno web con el funcionamiento iterativo típico de los controladores robóticos.

Otros métodos relacionados con los motores del robot son:

- *getV*: Método para que el usuario conozca la velocidad lineal actual ordenada al robot.
- *getW*: Método para que el usuario conozca la velocidad angular actual ordenada al robot.

- *getL*: Método para que el usuario conozca la velocidad de elevación ordenada al robot.
- *setV*: Método para que el usuario comande la velocidad lineal del robot.
- *setW*: Método para que el usuario comande la velocidad angular del robot.
- *setL*: Método para que el usuario comande la velocidad de elevación del robot.
- *move*: Método que combina la funcionalidad de *setV* y *setW* simultáneamente.

A continuación en la Figura 4.13 se presenta un esquema de la ejecución de la función *setVelocity* del robot.

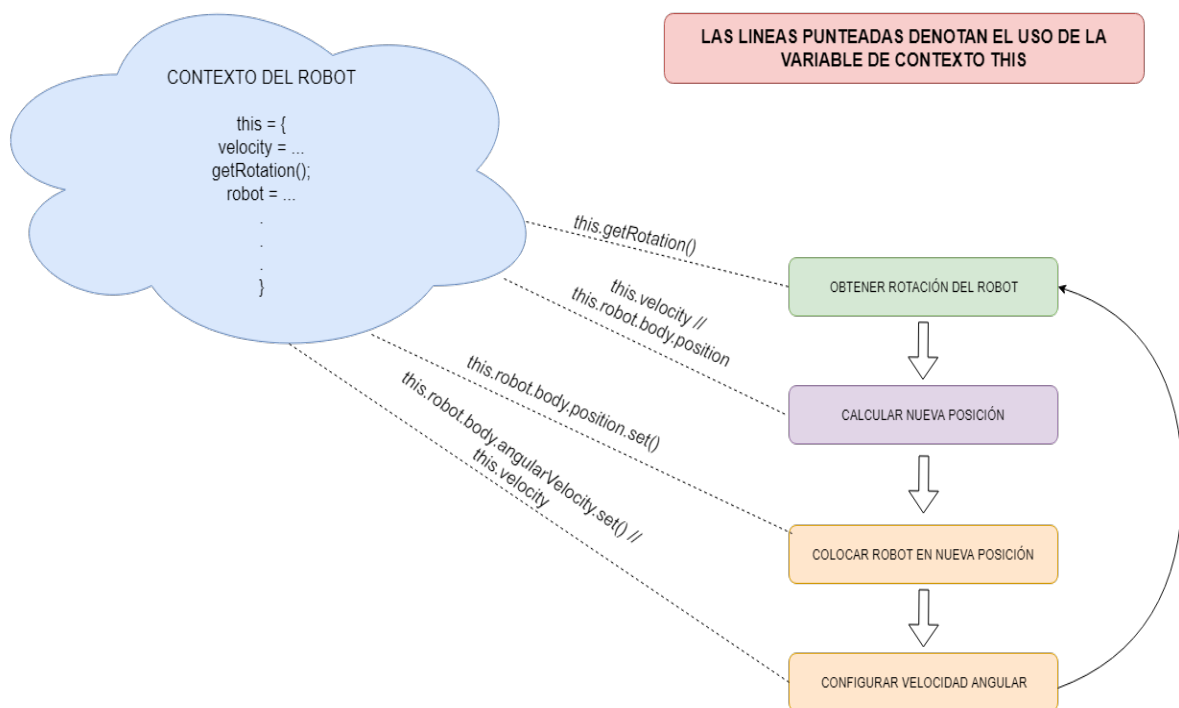


Figura 4.13: Esquema de ejecución de la función *setVelocity*, en la parte superior izquierda de la imagen se muestran algunos datos que existen en el contexto de la función.

Finalmente se muestra en la Tabla 4.1 el resumen de las funciones que se ofrecen en el *HAL API* para usar los motores en el robot simulado.

Cuadro 4.1: Métodos (API) de los motores del robot.

Método	Descripción
.setV(velLineal)	Mueve hacia delante o atrás el robot. INPUT: - velLineal: número con la velocidad lineal(m/s).
.setW(velAngular)	Hace girar al robot. INPUT: - velAngular: número con la velocidad angular (rad/s).
.setL(velElevacion)	Hace que el robot se mueva hacia arriba. INPUT: - velElevacion: numero con la velocidad de elevación (m/s)
.move(velLineal, velAngular)	Mueve el robot hacia delante/atrás y gira al mismo tiempo. INPUT: Los mismos parámetros que en setV y setW.
.getV()	Obtener la velocidad lineal configurada en el robot. OUTPUT: number
.getW()	Obtener la velocidad angular configurada en el robot. OUTPUT: number
.getL()	Obtener la velocidad de elevación configurada en el robot. OUTPUT: number

4.3.3. Driver del sensor de distancia a obstáculos

El sensor de ultrasonido o láser permite detectar obstáculos abarcando un radio de 180° por delante del frontal del robot. Este sensor ha sido simulado mediante el atributo *raycaster* existente en AFRAME que se asemeja a un láser y nos permite conocer el punto de intersección entre un rayo 3D y un determinado objeto.

Para la simulación de este sensor tenemos dos partes:

- Componente *AFRAME followBody*: Este componente tiene una función muy simple, anclar al robot un componente de tipo *raycaster*. En este caso sin tener que añadirle físicas, ya que al aplicarle físicas a una entidad se le aplican también a todos sus elementos 'hijo' dentro del HTML. Este componente permite que los *raycasters* sigan la posición del robot manteniendo su orientación relativa respecto del robot.
- Componente *AFRAME intersectionHandler*: Este componente es el más importante ya que permite manejar correctamente el evento que se dispara cuando hay una intersección para un *raycaster*. Los eventos tienen todos el mismo nombre y además el número de identificación del *raycaster* que lanza el evento. Esto permite detectar qué *raycaster* lanza el evento y guardar correctamente la distancia que detecta.
- *startRaycasters*: Función interna del objeto robot que sirve para iniciar los *raycasters* dándoles su ángulo de rotación respecto al robot y agrupándolos en función de su posición (izquierda - centro - derecha). Les da un identificador numérico unívoco que permite crear un evento individual para cada uno de ellos y además, una vez creados los *raycasters*, llama a la función que declara los escuchadores de eventos individuales para cada uno de ellos. Para conocer si los *raycasters* están activos o no se hace uso de una variable de estado *this.activeRays* que indica si los *raycasters* están en funcionamiento o no.
- *createRaycasters*: Función que crea un *raycaster* configurando una serie de atributos como la distancia máxima a la que se detectarán intersecciones. Además se le añade los componentes anteriormente mencionados *followBody* e *intersectionHandler*.
- *stopRaycasters*: Función que elimina todos los elementos *raycaster* del robot, es decir, detiene el sensor de obstáculos.
- *setListener*: Declara dos escuchadores de eventos, uno para el evento *intersection-detected-'id del raycaster'* y otro para el evento *intersection-cleared-'id del raycaster'* junto con las funciones a las que llamar cuando ocurra cada uno de los dos eventos. Cuando el evento *intersection-detected-'id del raycaster'* ocurre se llama a la función *updateDistance* y cuando ocurre el evento *intersection-cleared-'id del raycaster'* se llama a la función *eraseDistance*. Ambas funciones se explican en sus respectivos apartados.

- *removeListener*: Función que elimina el escuchador de eventos, es llamada desde la función *stopRaycasters*.
- *updateDistance*: Función que se llama cuando se detecta el evento *intersection-detected- 'id del raycaster'*. Esta función actualiza el array de distancias (*this.distanceArray*) detectadas por los *raycasters*. En cada posición del array de distancias se guarda un objeto JavaScript con dos campos: *id*, es un identificador numérico vinculado al *raycaster* que ha emitido el evento anteriormente mencionado, *d*, es la distancia a la que se encuentra el objeto detectado por el *raycaster* que ha emitido el evento.
- *eraseDistance*: Función que se lanza cuando se detecta el evento *intersection-cleared- 'id del raycaster'*. Esta función elimina el registro del array de distancias para el *raycaster* que lanza el evento y elimina dicho registro.
- *getDistance* y *getDistances*: Estas dos funciones son las que se proporcionan al usuario final. Ambas funciones del HAL API devuelven las distancias detectadas por los *raycasters*. La diferencia principal es que la primera función únicamente devuelve la distancia que detecta el *raycaster* central mientras que la segunda devuelve las distancia para todos ellos.

El inicio de los sensores de obstáculo es simple, el constructor llama a la función *startRaycasters* y ésta se encarga de configurar todo llamando a las funciones mencionadas anteriormente.

La Tabla 4.2 muestra las funciones que se ofrece al usuario para hacer uso de los datos recogidos por los sensores de obstáculos.

Cuadro 4.2: Métodos (API) de los sensores de obstáculos del robot.

Método	Descripción
<code>.getDistance()</code>	Permite obtener la distancia del objeto que tiene delante OUTPUT: number (metros)
<code>.getDistances()</code>	Permite obtener la distancia de los objetos detectados en un arco de 180°, devuelve 31 valores por defecto OUTPUT Lista con 31 valores de tipo number (metros).

4.3.4. Sensores de infrarrojos

El robot simulado está equipado con dos sensores ficticios de infrarrojos apuntando hacia el suelo. Se han implementado mediante el uso de una cámara específica (Figura 4.6) integrada en el cuerpo del robot por simplicidad y por mejora de rendimiento. Lo que se hace internamente es recortar la imagen de la cámara hasta quedarse con los píxeles que se encuentran más abajo en la imagen. El ancho de la imagen se mantiene, la imagen recortada tiene unas dimensiones de 5px de alto y 150px de ancho lo que permite que el robot detecte únicamente lo que tiene inmediatamente debajo.

La función del HAL API *readIR* es la encargada de simular este sensor de infrarrojo. Toma como parámetro de entrada un color como *string* y accede a la variable anteriormente mencionada *understandedColors* para obtener los valores de los filtros para el color seleccionados. Después recorta la imagen para obtener una imagen de 5x150 px. Posteriormente filtra por color la imagen para obtener únicamente la línea a seguir y calcula el centro de la línea mediante las funciones *findContours* y *moments* de la librería OpenCVjs.

La salida de la función son valores entre 0 y 3 que representan lo siguiente:

- 0: Los dos sensores ficticios infrarrojos están detectando la línea.
- 1: Únicamente el sensor infrarrojo de la izquierda detecta la línea.
- 2: Únicamente el sensor infrarrojo de la derecha detecta la línea.

- 3: Ninguno de los sensores detecta la línea.

Este formato de salida se debe a la implementación existente en el robot real para ser compatible con ella. Un objetivo es que el mismo programa creado en el robot simulado se pueda exportar al robot real y funcione de la misma manera, por tanto convenía mantener compatibilidad para que las aplicaciones funcionen en ambos sin que sean necesarios grandes cambios. En la Tabla 4.4 se explica la función que hace uso de los sensores infrarrojos ficticios y explica su parámetro de entrada y salida.

Cuadro 4.3: Métodos (API) de los sensores infrarrojos del robot simulado.

Método	Descripción
.readIR(color)	<p>Obtiene valores entre 0-3 en función de lo que detecte cada uno de los sensores ficticios</p> <p>INPUT:</p> <p>-color: color a filtrar en la imagen como string</p>

4.3.5. Driver de los sensores de odometría

Los sensores de odometría se encargan de obtener la posición absoluta del vehículo durante la navegación. Para la simulación de estos sensores no ha sido necesario ningún componente extra ni entidad, se ha hecho uso del sistema de coordenadas y rotación de *AFRAME* que permite la obtención de estos datos a través de su API JavaScript. De esta manera se han podido desarrollar dos funciones que simplifican el acceso a los datos, y se describen en la Tabla ??.

Cuadro 4.4: Métodos (API) de los sensores infrarrojos del robot simulado.

Método	Descripción
.getRotation()	<p>Devuelve un objeto con la orientación del robot en los 3 ejes</p> <p>OUTPUT:</p> <pre>{ x: rotacionX (radianes) y: rotacionY (radianes) z: rotacionZ (radianes) }</pre>
.getPosition()	<p>Permite obtener la posición del robot en la escena</p> <p>OUTPUT:</p> <pre>{ x: coordenadax (metros) y: coordenaday (metros) z: coordenadaz (metros) theta: rotacion eje Y (horiz, radianes) }</pre>

Cabe comentar el sistema de coordenadas del que hace uso *AFRAME* para entender los datos que se obtienen en estas funciones del HAL API. La Figura 4.14 muestra los ejes utilizados.

4.3.6. Driver de la cámara

Para la simulación de la cámara del robot se utiliza el componente *spectator* que emplea un *renderer* de la librería *three.js* para obtener la imagen de la escena. Posteriormente, en el objeto robot se crean una serie de métodos que permiten configurar y acceder a los datos de la cámara. A continuación se enumeran los métodos y se explica su función.

- *startCamera*: Esta función comprueba si la etiqueta con ID 'spectatorDiv' tiene una etiqueta 'hijo' que es una etiqueta de tipo *canvas* donde se representa la cámara. Una vez

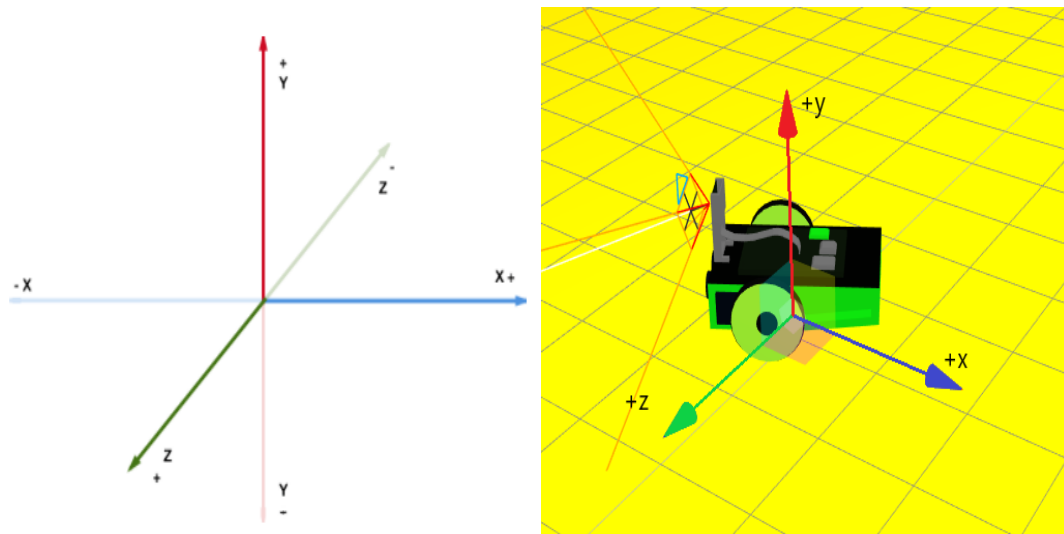


Figura 4.14: Sistema de ejes utilizado en las escenas del entorno *AFRAME*

haya cargado la etiqueta se accede al DOM para tomar el contenido de la etiqueta y se llama a la función *getImageData-async*.

- *getImageData-async*: Función que se ejecuta de manera similar a la función *setVelocity*, toma la imagen de manera autónoma haciendo uso de la función *imread* de la librería OpenCVjs y guarda los datos de la imagen en la variable *this.imagedata*. Por último se llama a si misma la función pasados 33 milisegundos (30 FPS). Sería una hebra más del simulador.
- *getImage*: Esta función del HAL API la invocará el usuario final para obtener los datos de la cámara y con ello materializar la lógica que se necesite.

Además de estos métodos 'crudos' se ofrecen en el HAL API de *WebSim* una serie de métodos algo más elaborados, 'cocinados', que se enumeran y explican a continuación:

- *getObjectColor* y *getObjectColorRGB*: Ambas funciones resuelven el mismo problema pero con parámetros de entrada distintos. Su función es filtrar un objeto que detecta la cámara del robot mediante su color. Devuelve un objeto con las coordenadas del centro del objeto en la imagen y el área del objeto en la imagen. *getObjectColor* toma como parámetro de entrada el color del objeto a detectar como cadena de caracteres, *getObjectColorRGB* toma como valores de entrada los filtros de color que se van a usar para detectar un elemento en la imagen.

En la Tabla 4.5 se resumen las funciones que hacen uso de la cámara y se muestran sus parámetros de entrada y salida.

Cuadro 4.5: Métodos (API) para obtener datos de la cámara del robot simulado.

Método	Descripción
<code>.getImage()</code>	Obtener la imagen de la cámara en el robot OUTPUT: <code>cv.Mat()</code> con la imagen de la cámara del robot.
<code>.getObjectColor(color)</code>	Devuelve un objeto JavaScript con datos sobre el objeto que detecte la cámara con el color pasado como parámetro. INPUT: color: color como string. OUTPUT: { center: [cx, cy], area: areaInt }
<code>.getObjectColorRGB(filtroBajo, filtroAlto)</code>	Devuelve un objeto JavaScript con datos sobre el objeto que detecte la cámara con el color pasado como parámetro. INPUT: filtroBajo: lista de longitud 4 con valores 0-255 (RGBA) filtroAlto: lista de longitud 4 con valores de 0-255 (RGBA) OUTPUT: { center: [cx, cy], area: areaInt }

4.4. Control de la simulación

Una de las funcionalidades de *WebSim* es el control de ejecución de código en el robot simulado. Para esto se utiliza una variable dentro del simulador de tipo *boolean* que indica si el robot ya se está usando, si ya tiene un código ejecutando en ese momento.

Si el robot está ejecutando lo que hace *WebSim* es parar el robot, es decir, ponerle la velocidad a 0 tanto lineal como angular y parar el hilo de ejecución del código que ha programado el usuario. De esta manera implementamos la 'pausa académica' que permite detener el robot, modificar una parte del código de su aplicación y continuar la ejecución con el algoritmo de la aplicación modificada sin tener que refrescar la página del simulador ni rehacer el código desde el principio.

Si el robot no tiene ningún código ejecutando actualmente entonces *WebSim* permite la ejecución del código mediante el uso de la instrucción nativa de JavaScript *eval()*. Es conocido que esta instrucción tiene una vulnerabilidad que permite la inyección de código maligno. Se ha estudiado esta vulnerabilidad y en el contexto que nos atañe no habría problema ya que es una aplicación totalmente local y todo código malicioso que se quisiese inyectar se inyectaría únicamente en la máquina local.

4.5. Conexiones de *WebSim* con software externo

Como se ha señalado en la sección 4.1, *WebSim* permite también la conexión con software externo para manejar el robot simulado mediante mensajes recogiendo medidas sensoriales y enviando órdenes a los motores. Esta funcionalidad se implementa aprovechando una de las características principales de JavaScript y es su orientación a *eventos*. Esto permite lanzar un evento (suceso) y escucharlo desde otra parte o método de la aplicación pudiendo ejecutar un código o incluso enviar datos en dicho evento.

En este caso se ha aprovechado esta funcionalidad para que *WebSim* ofrezca un método de entrada en el cual espera que se le pase un código en formato 'string' que posteriormente él se encargará de interpretar y ejecutar comprobando previamente como se ha comentado en secciones anteriores si el robot simulado ya tiene un código en ejecución. Esto permite por tanto la conexión con editores embebidos en la página e incluso conectar *WebSim* con una aplicación externa al navegador conectándose a través de *Websockets* como puede ser ROS o otro tipo de protocolos de comunicación como ICE.

Ambas interfaces (ROS y ICE) se ha programado en WebSim pero no forman parte del núcleo de este Trabajo Fin de Grado, por lo que apenas se describe un esbozo de este bloque software del simulador robótico. La forma de hacer esta transmisión de mensajes hacia *WebSim* es un programa JavaScript que se suscribe a los canales tanto de ICE como de ROS y traduce los mensajes a un código que *WebSim* interpretará y ejecutará en el robot simulado.

4.6. Empaquetado e instalación

Para empaquetar *WebSim* se ha hecho uso de los *import* de ES6 y la herramienta *WebPack* que permite generar un único fichero con todas las dependencias de un aplicación y, en modo producción, permite también minificar (reducir) el código de la aplicación una vez empaquetado para optimizar el rendimiento al cargar la página por parte de un usuario. Se ha programado en *WebSim* pero no forma parte del núcleo de este Trabajo Fin de Grado, por lo que apenas de describe un esbozo de este bloque software del simulador robótico.

Lo primero que ha sido necesario es instalar las dependencias a través de la herramienta NPM, lo que hace que se pueda importar posteriormente desde el punto principal lo que será el empaquetado de dicha dependencia. Desde el archivo *websim.js* se importan explícitamente las dependencias de *AFRAME*, *AFRAME-PHYSICS* y *jQuery*, además de exportar desde otros archivos las clases y funciones necesarias en el archivo principal.

La aplicación se empaqueta con WebPack y gracias a esta combinación se tiene un único *bundle* en el cual se incluirán todas las dependencias. A continuación se muestra en la figura 4.15 el fichero de configuración de *WebPack* para empaquetar WebSim.

En la Figura 4.15 se muestra el archivo 'webpack.config.js', este archivo declara la configuración de la herramienta *WebPack*. A continuación se explica con más detalle cada una de las partes y el papel que desempeña en el empaquetado final.

La Figura 4.16 muestra cuáles son los archivos en los cuales *WebPack* comenzará el empaquetado. En esos archivos existe una instrucción 'import' que declara una dependencia del código, por tanto, lo que hace *WebPack* es buscar esa instrucción y juntar en un mismo fichero

```
const path = require('path');

module.exports = {
  entry: {
    websim: './websim.js',
    editor: './editor.js'
  },
  output: {
    path: path.resolve(__dirname, 'build'),
    filename: '[name].bundle.js',
    publicPath: '/build/'
  },
  resolve: {
    extensions: ['.js']
  },
  devServer: {
    host: '0.0.0.0',
    port: '8080',
    inline: true
  },
  module: {
    rules: [
      {
        test: /\.js$/,
        loader: 'babel-loader',
        exclude: "/node_modules/"
      }
    ]
  },
  mode: 'development'
}
```

Figura 4.15: Fichero de configuración de la herramienta WebPack.

todas las dependencias (import) que se utilicen a partir de los ficheros iniciales.

La figura 4.17 muestra dónde y qué nombre tendrán los ficheros empaquetados por la herramienta *WebPack*. Cabe explicar la propiedad '[name]', que declara que el nombre del fichero de salida ha de ser el nombre de la etiqueta usado en *entry* que se muestra en la figura 4.16. En este caso tenemos *websim.bundle.js*.

La Figura 4.18 muestra la configuración en *WebPack* que declara cómo han de ser resueltos los módulos. Por ejemplo si hacemos uso de la instrucción *import 'aframe'* lo que hace es buscar la dependencia con el nombre 'aframe' y extensión declarada en la configuración.

La Figura 4.19 muestra la configuración de puertos y la dirección IP de la máquina donde se va a levantar un servidor estático de desarrollo por la herramienta *WebPack*. Esto permite hacer cambios en el 'core' de la aplicación y realizar el empaquetado y servicio de la aplicación

```
entry: {  
  websim: './websim.js',  
  editor: './editor.js'  
},
```

Figura 4.16: Ficheros de entrada que tomará *WebPack* para hacer el empaquetado.

```
output: {  
  path: path.resolve(__dirname, 'build'),  
  filename: '[name].bundle.js',  
  publicPath: '/build/'  
},
```

Figura 4.17: Nombre y directorio de los ficheros de salida empaquetados por *WebPack*.

de manera automática. Permite acelerar el proceso de desarrollo de cualquier aplicación. Como se ha mencionado, es un servidor estático por lo tanto tenemos que tener una página *index.html* que será la que se servirá en el navegador por defecto y ésta ha de hacer uso de los ficheros empaquetados por *WebPack*.

La Figura 4.20 muestra la configuración para los módulos de preprocesado de los ficheros a empaquetar, permite declarar cuál va a ser el paquete utilizado (que en este caso será *babel-loader*) y las extensiones de los ficheros que se van a usar, en este caso *.js* y *.jsx* (*React*). La extensión *.jsx* no va a ser utilizada. Esto nos permite compilar y transformar archivos en *TypeScript* por ejemplo a JavaScript para ser empaquetado, hay más tipos de *loaders* para hacer empaquetado incluso de archivos CSS.

Para la instalación en local es necesario tener instalados NodeJS y NPM, para ello hay que abrir una consola (Ubuntu/Linux) o un CMD (Windows)

- Ubuntu/Linux:

```
$ sudo apt update  
$ sudo apt get install nodejs  
$ sudo apt get install npm
```



```
resolve: {  
  extensions: ['.js']  
},
```

Figura 4.18: Declaración de las extensiones que ha de buscar *WebPack*.

```
devServer: {  
  host: '0.0.0.0',  
  port: '8080',  
  inline: true  
},
```

Figura 4.19: Configuración del servidor estático para desarrollo que *WebPack* permite usar.

— Para comprobar que Node está instalado:

```
$ nodejs --version
```

- Windows: Descarga el instalador de NodeJS de su página oficial.

Una vez instalado se copia el repositorio que se quiera utilizar de los que se muestran a continuación:

```
https://github.com/RoboticsURJC-students/2018-tfg-alvaro_paniagua  
https://github.com/JdeRobot/WebSim
```

Moverse a la carpeta del repositorio que se acaba de clonar y poner lo siguiente:

```
$ npm install
```

Esto instalará todas las dependencias en la carpeta 'node-modules', una vez finalizado esto se podrá arrancar el servidor, para ello hay dos opciones.

- Python: Necesitamos tener instalado python en nuestro equipo, si lo tenemos instalado ejecutamos la siguiente instrucción en la línea de comandos:

— Python v.3

```
module:{
  rules: [
    {
      test: /\.js|\.jsx$/,
      loader: 'babel-loader',
      exclude: "/node_modules/"
    }
  ]
},
mode: 'development'
```

Figura 4.20: Configuración de los módulos de preprocesado a usar por *WebPack*.

```
$ python -m http.server [port]
— Python v.2
$ python -m SimpleHTTPServer [port]
```

- **NodeJS:** En ambos repositorios anteriormente mencionados se ofrece bajo el nombre *server.js* un servidor que sirve recursos estáticos, basta con ejecutar lo siguiente tanto en Windows como Ubuntu.

```
$ node server.js
```

Una vez ejecutadas cualquiera de las dos opciones abrimos el navegador, se recomienda Firefox debido a que AFRAME tiene compatibilidad total con éste e ingresamos la siguiente URL.

```
localhost:8000/
```

En general, cualquier servidor de ficheros estático valdría para servir las aplicaciones ya que se han creado de tal manera que el servidor usado no tenga ninguna funcionalidad.

Capítulo 5

Robótica educativa

Como se ha comentado en el capítulo 1 del presente documento, los usos del simulador irán orientados a la robótica educativa. Se han creado dos aplicaciones de manera que los usuarios aprenderán a programar la lógica del robot usando una simplificación de acceso a los sensores y actuadores del robot simulado.

Se han preparado las aplicaciones para desarrollar una serie de ejercicios ya establecidos previamente al proyecto. En la siguiente sección se explicarán los distintos ejercicios.

Para el uso del proyecto podemos instalarlo en local que se explicará a continuación o bien se puede acceder a la siguiente URL. Es necesario acceder con el navegador *Firefox*.

```
https://jderobot.github.io/WebSim/  
https://roboticsurjc-students.github.io/2018-tfg-alvaro_paniagua/
```

5.1. Programando con *JavaScript*

La primera de las dos aplicaciones anteriormente mencionadas es una aplicación web que hace uso del simulador *WebSim* y el editor ACE Editor en la cual el alumno puede hacer uso del HAL API del robot simulado en lenguaje JavaScript para programar la lógica del robot y resolver un ejercicio propuesto.

El módulo del editor se encarga principalmente de crear los manejadores de eventos para

los eventos de los botones embebidos en la página web, configurar el renderizado del editor y la obtención de código escrito por el usuario.

En la figura 5.1 se muestra la arquitectura de la aplicación web y cómo el editor de código se comunica con WebSim que se encarga de interpretar el código del programador.

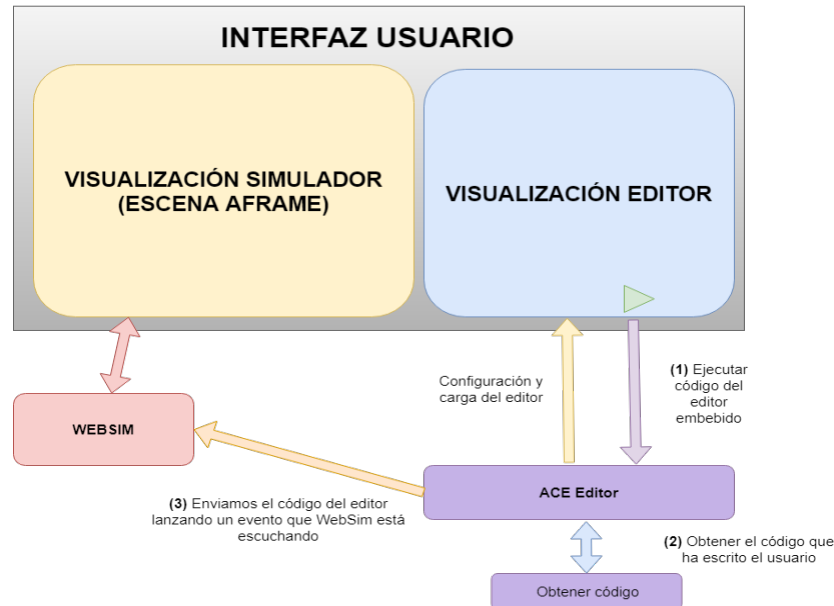


Figura 5.1: Arquitectura de la aplicación que hace uso del editor ACE y WebSim.

Como se ve en la figura la arquitectura tiene dos módulos, uno para el editor y el módulo de WebSim lo que dota de flexibilidad a la hora del desarrollo permitiendo tener distintos editores distintos o interfaces de entrada distintas.

Se explica a continuación el flujo de funcionamiento de la aplicación en los pasos marcados en la figura.

- (1): El usuario crea un código haciendo uso directo de la variable *myRobot* y pulsa el botón de ejecutar en la aplicación.
- (2): El evento 'click' es recogido por el módulo del editor que llama a la funcionalidad necesaria para obtener el código que el usuario ha escrito en el editor embebido y lo almacena en una variable.
- (3): Una vez se ha recogido el código en el paso (2) el módulo editor envía un evento que el módulo WebSim se encuentra escuchando. El código llega al módulo WebSim y este

se encarga de ejecutarlo o no dependiendo de si ya existe un código corriendo en el robot simulado.

5.2. Ejercicios con *JavaScript*

Como se ha comentado en el inicio del capítulo existen 3 ejercicios predefinidos que utiliza la plataforma *JdeRobot Kids* para enseñar a los alumnos programación de robots y visión artificial. Como lo que tenemos es una aplicación con un editor de código *JavaScript* se pueden incluso crear otro tipo de ejercicios ampliando el espectro de aprendizaje para los alumnos como por ejemplo conseguir mover el robot mediante teclado lo que les lleva a conocer la orientación a eventos de *JavaScript* y el manejo de estos, en concreto los eventos del teclado.

Una muestra de este ejercicio se enseña en el siguiente enlace (mover el robot con el teclado):

<https://www.youtube.com/watch?v=LlGeu95gEtk&t=1s>

A continuación se muestra una lista de enlaces a los ejercicios propuestos por *JdeRobot Kids* resueltos:

- Seguir un objeto detectado por su color:

<https://www.youtube.com/watch?v=9JIZO5E3jUo>

- Ejercicio sigue líneas:

<https://www.youtube.com/watch?v=tzxxEyA-LWs>

- Ejercicio evitar obstáculos:

<https://www.youtube.com/watch?v=VDW9FZcwA0g&t=8s>

Estos ejercicios lo único que tienen de diferente es el escenario de *AFRAME* (página HTML) por lo que se pueden crear distintas paginas para plantear tantos ejercicios como se quiera con el mismo robot tal y como se muestra en la figura 5.2 en la cual tenemos una escena 3D generada para la plataforma *JdeRobot Kids* en la cual tenemos un mayor número de obstáculos con el fin de aprender a programar la lógica para que el robot no se choque con los elementos.

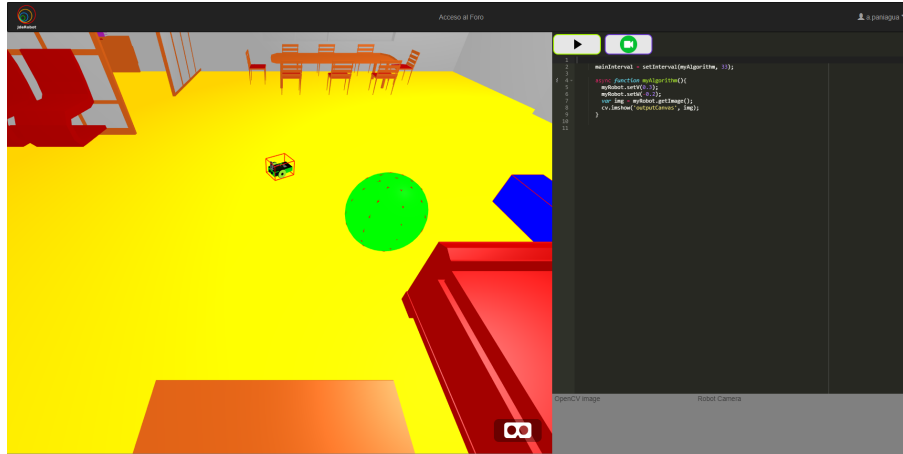


Figura 5.2: Interfaz de usuario de WebSim + editor JavaScript (Ace Editor) usado en *JdeRobot Kids*.

5.3. Programando con *Scratch*

Haciendo uso del mismo core de *WebSim* se ha creado una pequeña aplicación para la resolución de los ejercicios planteados en *JdeRobot Kids* mediante el uso de bloques visuales con *Blockly*.

Esta aplicación se ha creado para una toma de contacto con la lógica de programación de robots para usuarios no introducidos en el lenguaje *JavaScript*. En la figura 5.3 se muestra la interfaz de bloques visuales en la que se ve que no es necesario programar, es una interfaz de tipo *Plug and Play* en la que se conectan los bloques unos con otros para generar código.

A continuación se muestran enlaces a vídeos que muestran los ejercicios resueltos:

- Seguir un objeto detectado por su color:

<https://www.youtube.com/watch?v=GOaxPyp0Lk4>

- Ejercicio sigue líneas:

<https://www.youtube.com/watch?v=iouvTDALM18>

- Ejercicio evitar obstáculos:

<https://www.youtube.com/watch?v=yKXP3UIAxtg>

Como se ve los bloques son autodescriptivos, cada bloque indica su propia funcionalidad.

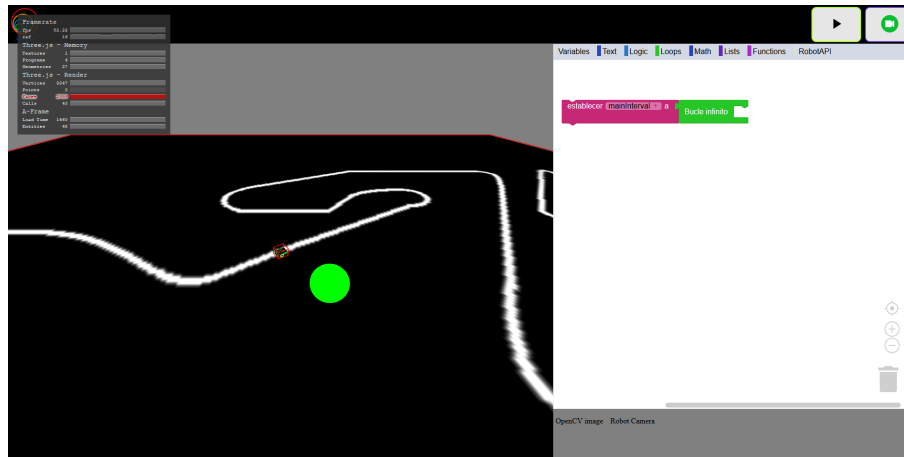


Figura 5.3: Interfaz de la aplicación WebSim + Blockly

En la figura 5.4 se muestra la arquitectura usada para la construcción de la aplicación web. Como se puede apreciar tanto esta como la arquitectura en la figura 5.1 son similares, los únicos módulos que cambian son los referentes al editor que cada editor necesitará una configuración diferente en función de sus características.

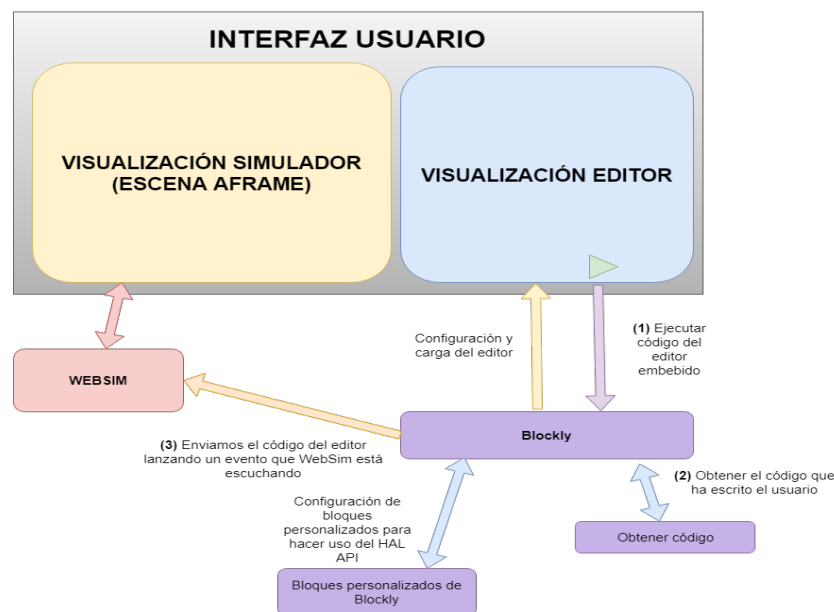


Figura 5.4: Arquitectura de la aplicación web que hace uso de Websim + Blockly.

Al estar usando Blockly necesitamos crear los bloques personalizados que, al ser usados por el usuario, se traducirán a las instrucciones JavaScript que hacen uso del HAL API que ofrece WebSim.

El flujo de ejecución de código es el mismo que el mencionado en la sección 5.2 pero modificando el módulo del editor a los necesarios para configurar Blockly y traducir los bloques al lenguaje JavaScript.

5.4. Ejercicios con *Scratch*

Los ejercicios propuestos son similares a los mencionados en la sección 5.2 a excepción del ejercicio de manejar el robot mediante el teclado.

La diferencia en los ejercicios reside principalmente en la escena que se va a usar, los métodos necesarios para su resolución y la plantilla de bloques visuales que se proporciona al usuario.

En la figura 5.3 se muestra la escena necesaria para la propuesta del ejercicio en el cual se pide al usuario programar la lógica para perseguir la esfera de color verde. Se menciona ésta en concreto debido a que para que la esfera se mueva necesitamos importar un script en el navegador en el cual dotemos de movimiento aleatorio a la esfera.

Estos ejercicios nos permiten validar el desarrollo del proyecto ya que nos permiten comprobar la estabilidad y flexibilidad del simulador y más concretamente del HAL API desarrollada para el robot.

Capítulo 6

Conclusiones

6.1. Valoración objetivo final

Al repasar los objetivos del capítulo 2 concluimos que se ha conseguido llevar a cabo los puntos establecidos ya que el principal punto era crear un simulador en lado cliente. Como se ha podido intuir a lo largo del documento no existe un servidor con una gran funcionalidad, al contrario, el servidor utilizado es un servidor estático que ofrece la plataforma de *Github*.

Además como se comentó al inicio, este simulador es la base funcional, hay muchas características interesantes que se podrán ir implementando a posteriori como podría ser el soporte de distintos prototipos de robots como por ejemplo drones y habrá que seguir en detalle también las nuevas características que se irán añadiendo en el entorno *AFRAME* que permitirán mejorar el rendimiento y funcionalidad de la plataforma.

6.2. Aplicación de lo aprendido

Para llevar acabo el proyecto me han sido imprescindibles los conocimientos adquiridos en las asignaturas relacionadas con la programación y tratamiento de imagen, a continuación se enumeran las asignaturas relacionadas con estos campos cursadas en el Grado.

- Informática I con el lenguaje 'Picky' en el cual tuve una primera toma de contacto con los fundamentos de programación.

- Informática II, avance de Informática I en el cual se llevaron a cabo proyectos algo más elaborados y se profundizó en el aprendizaje de 'punteros'.
- Protocolos de Transmisión de Audio y Vídeo en Internet, esta asignatura fue la primera aproximación con un lenguaje de programación orientado a objetos como *Python*.
- Graficos y visualización 3D, esta asignatura ha sido una de las claves de aprendizaje para el desarrollo del proyecto ya que fue mi primera toma de contacto con el lenguaje *JavaScript* y el *canvas*.
- Laboratorios en Tecnologías y Aplicaciones Web, el segundo punto clave de aprendizaje ya que estableció la base de conocimiento sobre tecnologías web como NodeJS y Django.
- Tratamiento digital de la imagen, esta asignatura ha sido importante debido a que ha establecido la base de conocimiento sobre filtrado de imagen necesario para su uso con el robot.

Durante el proyecto he aprendido muchísimo sobre tecnologías web como el uso de herramientas de empaquetado como Webpack, he mejorado mucho en el uso de control de versiones en la plataforma Github. Además he aprendido cómo estructurar una aplicación completa mediante el uso de módulos de código separados. Por último he aprendido cómo usar los paquetes *NPM* como dependencias de código lo que simplifica mucho la utilización de la aplicación e instalación de dependencias.

6.3. Mejoras futuras

Como futuras mejoras hay muchas por abarcar, a continuación se enumeran algunas de ellas:

- Permitir que *WebSim* utilice un fichero de configuración para crear el robot.
- Añadir funcionalidad de guardado de código en el servidor para los alumnos.
- Extender el soporte para otros robots con distinta funcionalidad.
- Mejora general de la escena de AFRAME.
- Integrar con el robot real, esto se hará en un futuro inmediato.

- Explorar mejoras de rendimiento.
- Explorar portabilidad a otros navegadores como Chrome, Safari, etc. Aunque algunos de ellos no soportan el entorno AFRAME.

Bibliografía

Google Blockly:

<https://developers.google.com/speed/docs/insights/MinifyResources?hl=es-419>

Historia JavaScript:

<https://medium.com/@benastontweet/lesson-1a-the-history-of-javascript->

Documentación Webpack:

<https://webpack.js.org/>

Vídeo de Youtube para configurar Webpack:

<https://www.youtube.com/watch?v=PakrjWSD6Mo>

Documentación HTML:

<https://www.computerhope.com/jargon/h/html.htm>

Configuración Webpack:

<https://jgbarah.github.io/aframe-playground/figures-02/>

NPM para principiantes:

<https://www.impressivewebs.com/npm-for-beginners-a-guide-for-front-end>

Documentación completa AFRAME:

<https://aframe.io/>

Documentación jQuery:

<https://jquery.com/>