UNIVERSIDAD
REY JUAN CARLOS

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

Trabajo Fin de Grado

Título del Trabajo con Letras Capitales para Sustantivos y Adjetivos

Autor : Álvaro Paniagua Tena

Tutor : Dr. Jose María Cañas Plaza

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

II

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 tambien a Jose María Cañas por darme la oportunidad de colaborar en el 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

El proyecto tiene una intención clara, la creación de un simulador de robots aprovechando el crecimiento de las tecnologías web y los nuevos estándares como HTML 5 además de el crecimiento de la realidad virtual en la web mediante el framework **AFRAME**.

El proyecto pretende crear una herramienta educativa, **WebSim**, para la plataforma JdeRobot Kids y el desarrollo de dos aplicaciones web para resolver ejercicios de **visión artificial y programación de robots** mediante programación en *JavaScript* y mediante el uso de bloques visuales con *Blockly*.

Para el desarrollo del simulador **WebSim** se han utilizado diversas herramientas.

- AFRAME y AFRAME Physics (sistema de físicas de AFRAME).
- HTML 5.
- JavaScript y jQuery.
- CSS3.
- Blockly.
- ACE Editor.

Por último para la gestión de dependencias se utiliza la tecnología NPM.

Summary

This project has a clear intention, create a robot simulator taking advantage of the growth of web technologies, the new web standard like HTML 5 and the growth of virtual reality on web using **AFRAME** framework.

The project aims to be an educational tool, **WebSim**, for the JdeRobot Kids platform and develop two web applications where students can solve **artificial vision and robotic programming** problems using *JavaScript* or using visual blocks from *Blockly*.

Different tools have been used to develop the **WebSim** simulator.

- AFRAME y AFRAME Physics (AFRAME physics system).
- HTML 5.
- JavaScript y jQuery.
- CSS3.
- Blockly.
- ACE Editor.

Finally, NPM technology is used to manage dependencies.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Robótica

La robótica es la ciencia 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 poder imitar el comportamiento de un humano. 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.

Como ejemplo cabe destacar la existencia de un robot llamado Da Vinci 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.

1.2. Tecnologías Web

La principal y creo que más importante de las ventajas que ofrecen las tecnologías web es la ausencia de necesidad de instalación de paquetes de software, configuración y dependencias. Basta con tener un navegador, en este caso Firefox, 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 binarios como C++ o Java sino interpretados, ésta desventaja cada vez va siendo menor debido a las mejoras 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 frameworks que facilitan esta tarea como pueden ser Express y Loopback para el lenguaje JavaScript y Django para el lenguaje Python.

Como conclusión, las tecnologías web están avanzando cada vez más con el objetivo de llegar al rendimiento de las aplicaciones de escritorio hasta el punto de existir frameworks para el desarrollo de aplicaciones híbridas, es decir, desarrolladas con lenguaje web pero haciendo uso del sistema operativo como es el caso de *Electron*.

1.3. Robótica educativa

La robótica educativa son 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.

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.

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.

1.4. Motivación

La motivación de este proyecto es la de sustituir el presente simulador *Gazebo* por WebSim, un simulador desarrollado íntegramente con tecnologías Web con peso computacional en el lado cliente lo que permite escalar el número de usuarios que lo utilizan de manera simultánea.

1.4.1. Motivación personal

Como motivación personal decir que, desde el inicio del grado me ha gustado el mundo de la programación y ya de pequeño tuve mis primeras experiencias en robótica haciendo robots de **Lego** usando **Lego Mindstorm.**

Además, me gusta poder ayudar al aprendizaje de los demás por tanto este proyecto creo que encaja a la perfección ambos aspectos.

1.5. 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 2 se muestran los objetivos a completar para la elaboración del proyecto y la estructura del **roadmap**.
- En el capítulo 3 se presentan las tecnólogias 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 5 se muestran ejercicios que el alumno podrá resolver usando el lenguaje *JavaScript*.
- En el capítulo 5 se muestran ejercicios a resolver por el alumno usando el 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

2.1. Objetivo general

Mi trabajo fin de grado consiste en crear la base de una herramienta educativa 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.

2.2. Objetivos específicos

El objetivo específico de el proyecto es la simulación del robot **PiBot** implementando así todo su conjunto de sensores y actuadores en los que se encuentran:

- **Motores**: son 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.

2.3. Planificación

2.3.1. 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, ésto 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.

2.3.2. Planificación temporal

A continuación se muestran tres imágenes que representan la planificación temporal

El nivel de esfuerzo para este proyecto ha sido alto debido a la necesidad de aprender diferentes tecnologías como son AFRAME, jQuery y OpenCVjs. Se dedicaban alrededor de 3-4 horas al día cada día de la semana a excepción de los fines de semana que se añadían 2 horas más al anterior intervalo.

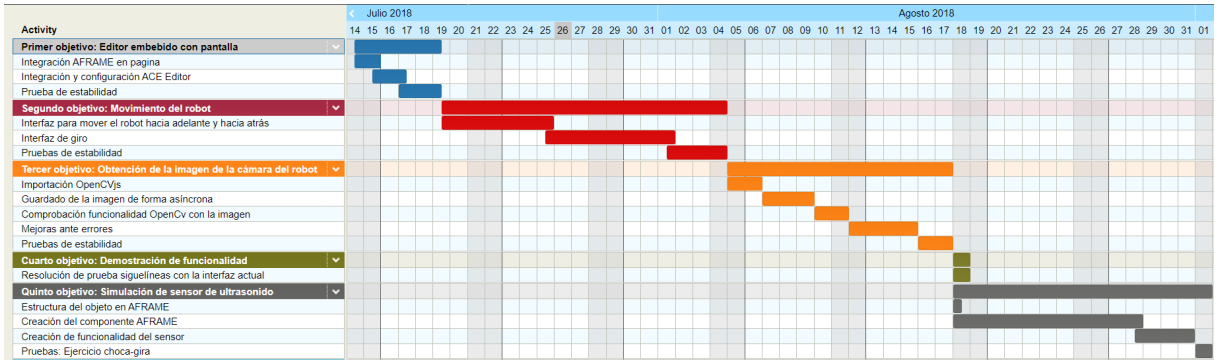

Figura 2.1: Diagrama de GANTT para el rango de fechas 14 de Julio de 2018 al 1 de Septiembre de 2018.

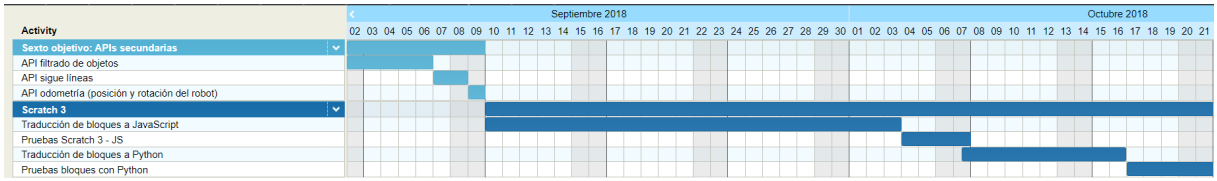

Figura 2.2: Diagrama de GANTT para el rango de fechas 2 de Septiembre de 2018 al 21 de Octubre de 2018.

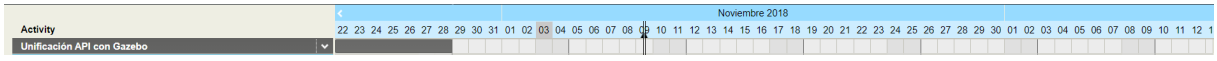

Figura 2.3: Diagrama de GANTT para el rango de fechas 22 de Octubre de 2018 al 28 de Octubre de 2018.

Capítulo 3

Herramientas

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.

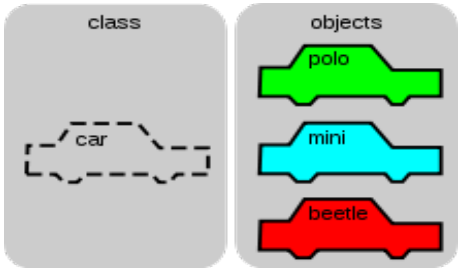

Figura 3.1: En la parte izquierda de la imagen se muestra el objeto **coche** del que heredan los 3 de la parte derecha de la imagen, los 3 coches tienen en común que son del objeto coche pero se diferencian en **marca** y **color.**

- 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*.

```
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

- 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*. 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 interprete 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*.

```
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*.

- 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 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.

- 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

```
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.

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.

```
//  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.

- 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 3.1.1 muestra una comparativa entre ambas sintaxis, en la parte superior vemos la sintaxis en estándares y en la parte inferior la sintaxis de ES6.

3.1.2. Frameworks JavaScript

Este lenguaje tiene mucho peso en el lado cliente, es decir, en el código que se ejecuta en el navegador pero gracias a la entrada de NodeJS cuando éste lenguaje ha obtenido más peso

```
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;
};
```

Figura 3.6: Estándares previos a ES6.

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

Figura 3.7: Estándar ES6.

dentro del desarrollo web ya que NodeJS permite la creación de servidores de manera sencilla.

Actualmente **JavaScript** se utiliza en multitud de frameworks 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 frameworks 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 framework 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 framework de creación de escenas de *Realidad Virtual*, se hablará de manera más extensa en el apartado 3.4. No es un framework 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 frameworks disponibles para JavaScript sirve como respaldo ante la decisión de elegir el lenguaje. Además como se ha comentado en la subsección **??** el proyecto se orienta a la creación de una aplicación con peso en el lado cliente, por tanto sabiendo ésto y teniendo en cuenta que AFRAME está creado en el lenguaje JavaScript la elección de este lenguaje gana peso por sí sola.

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 3.8 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 3.8: 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.

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 pagina 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 fines. 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.

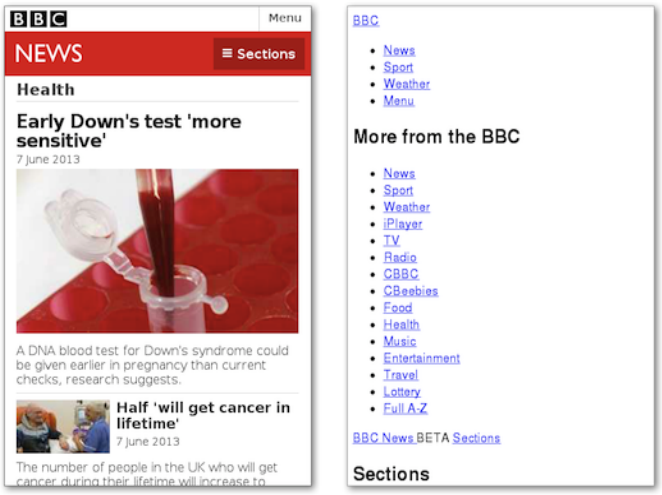

Figura 3.9: Imagen comparativa de la misma pagina web con y sin hojas CSS

Como vemos en la figura 3.9 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 framework 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 framework 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 360º, 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 framework 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 frameworks 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 3.10 únicamente se necesita una etiqueta **script** que haga referencia al código del framework 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 3.10 muestra todo el código necesario para crear la escena mostrada.

3.4.2. Sistema Entidad-Componente

AFRAME se basa en el framework ***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.

```
<html>
  <head>
    <script src="https://aframe.io/releases/0.8.0/aframe.min.js"></script>
  </head>
  <body>
    <a-scene>
      <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9"></a-box>
      <a-sphere position="0 1.25 -5" radius="1.25" color="#EF2D5E"></a-sphere>
      <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color="#FFC65D"></a-cylinder>
      <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#7BC8A4"></a-plane>
      <a-sky color="#ECECEC"></a-sky>
    </a-scene>
  </body>
</html>
```

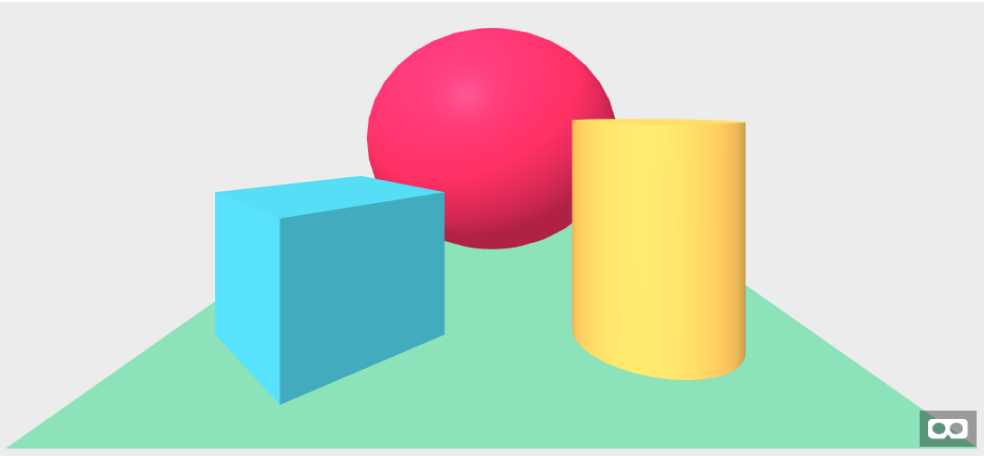

Figura 3.10: Ejemplo de código HTML que renderiza una escena básica de realidad virtual

- 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 3.11 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

Figura 3.11: La siguiente figura muestra etiquetas que serían el equivalente en AFRAME a un componente, el conjunto de varios componentes dan forma a la caja

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 3.12: 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 3.13 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

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

Figura 3.13: 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

lenguaje utilizado lo que permite registrar y suscribir eventos que permite que los elementos en la escena no se conozcan entre sí.

- 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 3.10 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.

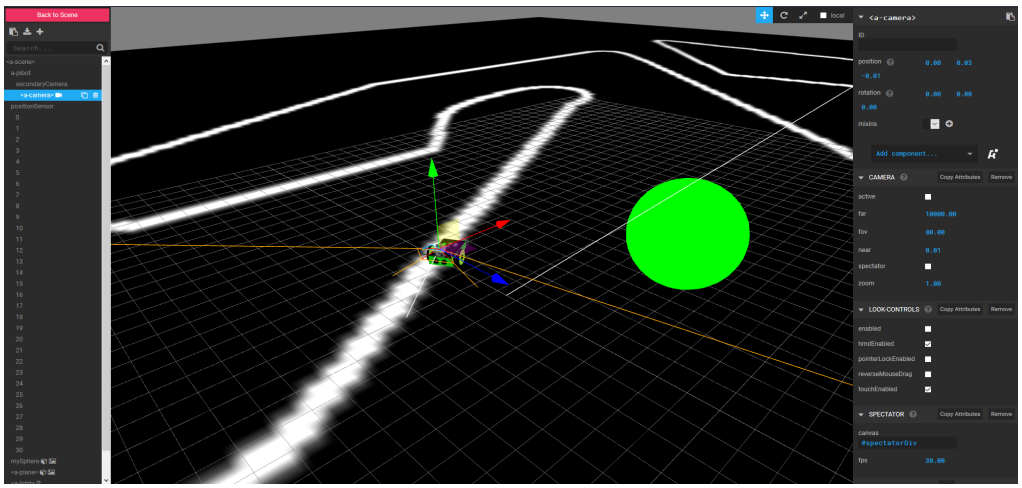

Figura 3.14: Inspector visual que ofrece el framework *AFRAME*

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 3.15: 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 3.16 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 3.16: 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 3.17 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.');
    this.setHelpUrl('http://www.w3schools.com/jsref/jsref_length_string.asp');
  }
};
```

Figura 3.17: 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 3.18

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

  }
};
```

Figura 3.18: 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 3.19 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.

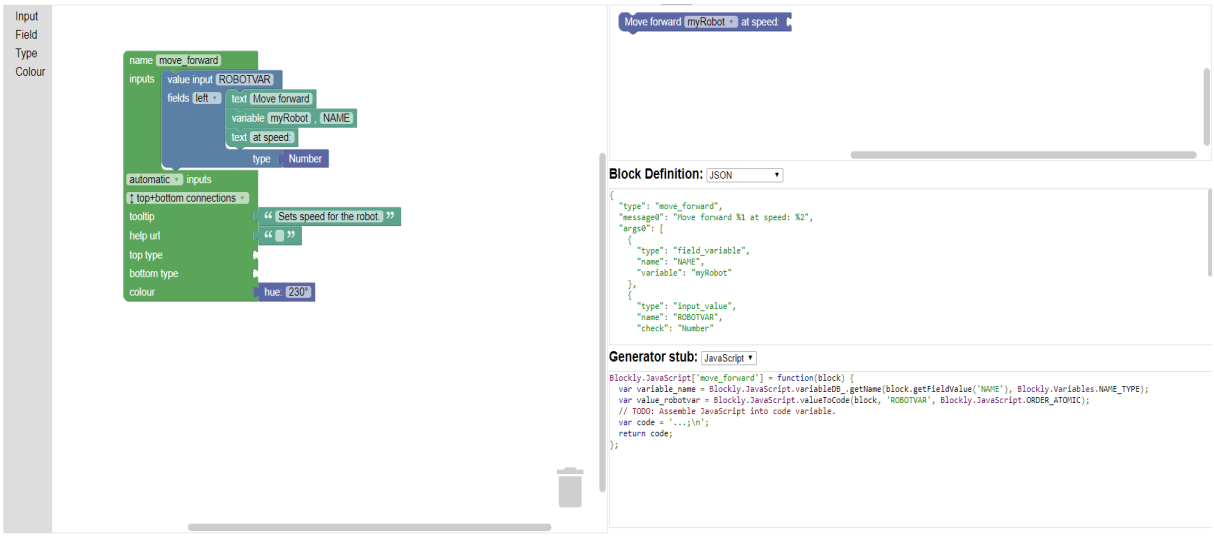

Figura 3.19: Herramienta de desarrollo para generación de bloques personalizados.

Como se ve en la figura 3.19 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.

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

Figura 3.20: Ejemplo de las etiquetas posibles de configuración de la *toolbox* del editor

Como vemos en la figura 3.20 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 (Asyncronous 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 3.21 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 3.21: 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 frameworks 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 frameworks.

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

Diseño e implementación

En este capítulo se mostrará la estructura del proyecto y se desarrollarán cada una de las partes explicando la funcionalidad de cada una.

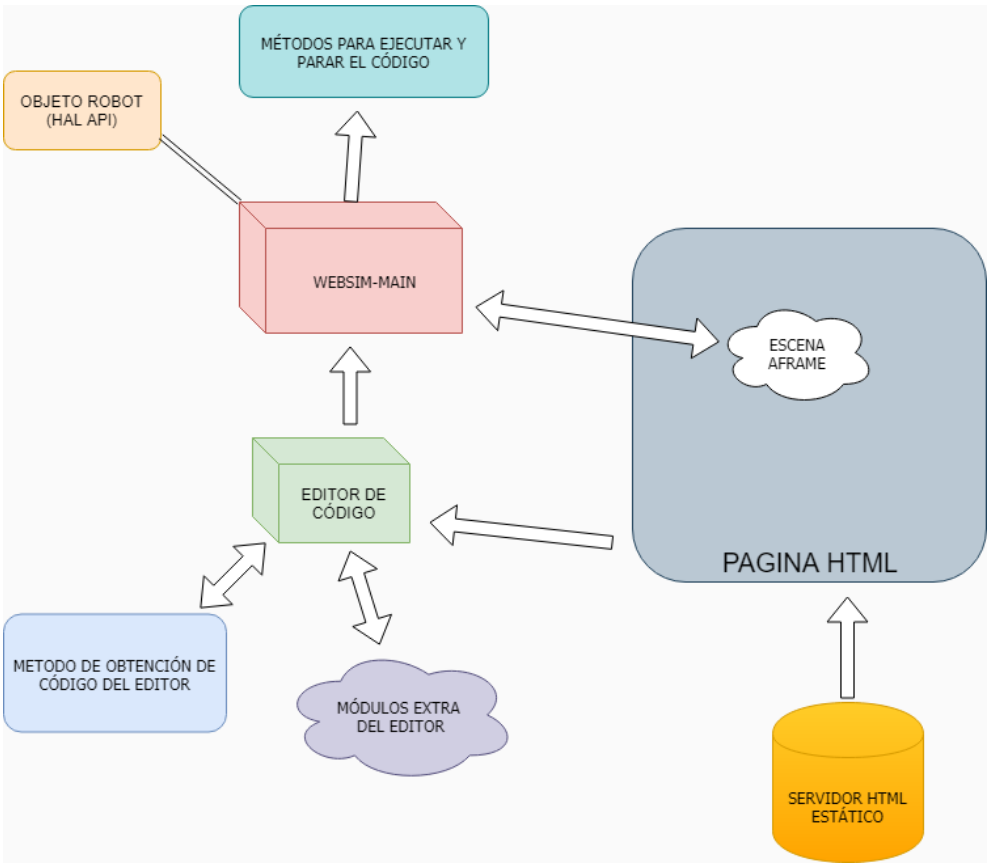

Figura 4.1: Esquema que presenta la estructura del simulador WebSim.

4.1. Arquitectura

Como se comentó en la sección 2.1 el objetivo del siguiente proyecto es crear la base de una aplicación para simular robots en el entorno *AFRAME* permitiendo conectar un editor de código para ejecutar instrucciones con el robot simulado.

En la figura 4.1 se muestra el esquema que sigue el simulador WebSim, a continuación se explicará cada una de las partes por separado y la comunicación existente entre ellas.

El diseño como se puede apreciar en la figura 4.1 sigue un diseño por módulos, la intención es poder conectar distintos tipos de entradas a WebSim como puede ser un editor, Blockly, ICE, etc. sin tener una aplicación monolítica en la que todas las partes estén completamente acopladas.

WebSim tiene varios roles:

- La funcionalidad más importante de WebSim es que se encarga de controlar la ejecución del *mundo*, es decir, sabe cuándo el robot está ejecutando un código y cuándo no y permite arrancar o pausar el robot. Se muestra en la figura 4.1 mediante el bloque **Métodos para ejecutar y parar el código**. 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 tener que refrescar la página.

- Se encarga de ofrecer la conexión simplificada con el robot simulado en el entorno AFRAME que como ya se ha explicado es la abstracción de nuestro robot real, por tanto *WebSim* ofrece un **HAL API** *(Hardware Abstraction Layer)* lo que simplifica mucho el uso del robot creado en AFRAME, podemos enviarle instrucciones que el robot ejecutará mediante una única línea de código, lo que permitirá al usuario crear una lógica para el robot de una manera limpia sin tener que comunicarse directamente con el motor *AFRAME*, 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 y segundo que

evitamos la creación de dos instancias del mismo objeto que llevaría a una incorrecta ejecución.

- Ofrece una interfaz de escucha de eventos en la cual el editor o la parte que se conecte con *Websim* recibe el código que se quiere ejecutar y llama a los métodos necesarios para ejecutar el código que hace uso del robot simulado que como ya se ha comentado en el punto anterior está contenido en la variable global de *WebSim* llamada **myRobot**. Esto permite conectar distintos tipos de 'puertas' para ejecutar código en el robot simulado como pueden ser las interfaces **ROS Y ICE**.

- Por último como se ha comentado una de las dependencias de *WebSim* es el framework *AFRAME*, para la simulación del robot ha sido necesario crear tres componentes llamados **followBody, spectatorComponent, intersectionHandler** y es *WebSim* el encargado de registrar estos componentes haciendo uso de los métodos que ofrece *AFRAME*, la figura 4.2 muestra la forma de registrar los nuevos componentes.

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

Figura 4.2: Registro de los tres componentes necesarios para el Robot simulado.

Como se puede ver se cumple el objetivo fijado en el inicio y es proveer 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, motores, sensores de ultrasonido y sensores infrarrojos.

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. Algunos ejemplos de ejercicios soportados son sigue línea, seguimiento de objeto mediante filtrado de colores y evitar obstáculos mediante el control de posición. A continuación se muestran enlaces a los ejercicios resueltos.

4.2. Simulación con AFRAME

En esta sección se explicará como 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 siguiente figura se muestra el esquema HTML necesario para generar nuestra escena junto con nuestro robot.

```
<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>
    <img id="ground" src="assets/textures/cheste-circuit.png" />
  </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">
      <!-- Create a second third-person camera which cant be controlled, the Pibot camera -->
      <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-light type="point" intensity="1" position="-2 2 4"></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 4.3: HTML que genera la escena principal del simulador

Ahora se explicarán cada una de las partes del HTML de la figura 4.3, se explicarán las que no sean entendibles de manera directa.

Como hemos comentado en el capítulo 3 la etiqueta **a-scene** es la etiqueta principal utilizada en *AFRAME* para generar la visualización, en nuestro caso como queremos una escena simple le hemos dado un color de fondo gris y hemos activado las físicas y mediciones. Las físicas permite hacer uso del motor de físicas en el paquete **aframe-physics** en el cual existe la gravedad, rozamiento y otros tipos de fuerzas. La etiqueta **stats** nos permite conocer las métricas relevantes de la escena como los *FPS* (Frames 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 de el código bajo ésta.

Las etiquetas **a-assets** 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 ha cargado correctamente.

Como hemos 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 ésto hemos podido simular nuestro robot, nuesto robot se compone de ruedas, sensores y cámara.

Para la simulación del cuerpo del robot hemos utilizado el atributo **collada-model** que nos 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 hemos tomado éste ya que era el que mejor rendimiento daba al cargarlo en la escena. Además se ha utilizado el atriburo **dynamic-body**, éste atributo lo que hace es dotar de físicas a ese elemento con la particularidad que el elemento puede ser movido por otros, en la imagen también se ve el atributo **static-body** que también es dotado de físicas pero de una manera distinta, las entidades con éste atributo no pueden ser movidos, se usa en general para el suelo, paredes y en general entidades que no han de ser movidas en la escena.

El robot además tiene la cámara y sensores de ultrasonido, para los sensores de ultrasonido se ha utilizado el atributo **raycaster** que se encapsula dentro de la entidad con el identificado

positionSensor junto con el componente **followBody**. La carga de estos elementos se hace desde WebSim, es necesario hacerlo así para poder modificarlos dinámicamente y añadir más o menos sensores al robot.

Por último tenemos la cámara, como vemos 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 incluyamos esta dentro de una entidad. La etiqueta cámara cuenta con atributos entre los que destacaremos 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 imprimir lo que está viendo el robot.
- WASD-Controls-Enabled: Permite declarar si la cámara se podrá mover con las teclas W-A-S-D, en nuestro caso no nos interesa ya que modificaríamos la posición de la cámara respecto del robot estropeando así la escena.

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 nuestro circuito en blanco y negro y el número de veces que se repite a lo largo de los ejes X y Z. Como hemos indicado que solo se repita 1 vez en ambos ejes lo que hace *AFRAME* es estirar la textura hasta ajustarla con el 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 principal de la escena.

4.3. Drivers del robot

En el siguiente apartado se explicará cada una de las partes del objeto robot *(HAL API)* su funcionalidad y cómo ha sido implementado, las partes que se implementan se encuentran enumeradas en la sección 2.2.

Estos 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 llamar a estos datos y programar una lógica en función de estos, para ello se crean una serie de métodos que se conectan con el robot simulado en AFRAME y variables internas para guardar los datos y servirlos mediante una instrucción simple.

4.3.1. Constructor

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 nuestro robot simulado para poder hacer uso de las propiedades que ofrece el entorno AFRAME como puede ser 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 lo que hace es iniciar una serie de variables internas del robot (*this* hace referencia al contexto del objeto robot):

- **defaultDistanceDetection**: Es una variable de configuración de los sensores de ultrasonido que permite declarar la distancia máxima a la cuál los sensores detectarán un objeto, en nuestro caso se ha puesto hasta 10 metros.
- **defaultNumOfRays**: Es una variable de configuración de los sensores de ultrasonidos, declara el número de láseres que se simularán para detectar objetos igual que lo haría un sensor de ultrasonidos. El arco que abarcamos es 180º por lo tanto cuantos mayor sea el número configurado mejor será la detección de objetos.

- **this.robot**: Es una variable que permite crear un link entre la abstracción de nuestro objeto robot y el robot simulado en AFRAME y nos permite acceder a los distintos métodos que ofrece AFRAME por defecto.

- **this.activeRays**: Es una variable de control de tipo *boolean* que nos 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 lo que haría 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, está dividido en tres grupos; 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 mapear (vincular un valor o serie de valores a otro distinto) un color de entrada como tipo *string* a sus *arrays* de filtros RGB (Red-Green-Blue) para poder detectarlo mediante el uso de herramientas de OpenCVjs lo que permite hacer una simplificación para detectar objetos mediante su color para usuarios que no tengan conocimientos en imagen. 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 necesitaremos conocer la velocidad actual, esto nos permite tenerla guardada para su uso tanto por los motores como por los usuarios.

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.

La figura 4.4 muestra el código del constructor del robot, como se ve en la imagen 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.understandedColors = {
      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 4.4: Código del constructor del objeto robot, la variable *this* hace referencia al contexto.

4.3.2. Motores

En el siguiente apartado se explicará la función de los motores en el robot simulado, cómo se inician y a qué submétodos llama.

La función principal de los motores es la de permitir el movimiento del robot en el plano horizontal a la velocidad configurada por el usuario. Además un objetivo secundario para esta interfaz es que los motores funcionen de manera autónoma, es decir, que no tengamos que enviar constantemente una instrucción al robot para que este se mueva sino que al configurar la velocidad al robot este se mueva de manera autónoma.

Antes de explicar la funcionalidad completa del motor hay que hacer un inciso en lo que representa las velocidades configuradas en la variable *this.velocity* en el que 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 el robot. En el eje Z tendremos la velocidad de giro.

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

- Obtenemos la rotación, esto se hace debido a que, como me encuentro en un sistema de coordenadas, necesito saber hacia donde mira el robot para poder moverlo hacia adelante y atrás correctamente.

- Calculo de la nueva posición, como se ha comentado en el punto anterior, necesito calcular la nueva posición en función de la velocidad lineal configurada en ese instante y en función del vector de vista (hacia donde mira el robot). Para calcular dicha posición es necesaria la rotación, 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 4.5: Calculo de la nueva posición en función del ángulo de rotación y la velocidad lineal.

- Establecemos la nueva posición para el objeto en la escena, se establece la nueva posición previamente calculada en la escena. Además se establece 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 ejecute así misma constantemente y no tener que estar enviando constantemente instrucciones de velocidad lo que simplifica el código.

Gracias a esta última instrucción nativa de *JavaScript* podemos crear una función autónoma, por tanto, 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 autónoma del motor la que se encargará de comprobar este registro para saber cuál es la velocidad configurada por el usuario.

Otros métodos relacionados con los motores del robot serían:

- **getV**: Método para que el usuario pueda conocer la velocidad lineal actual configurada en el robot.
- **getW**: Método para que el usuario pueda conocer la velocidad angular actual configurada en el robot.
- **getL**: Método para que el usuario pueda conocer la velocidad de elevación configurada en el robot.
- **setV**: Método para que el usuario configure la velocidad lineal del robot.
- **setW**: Método para que el usuario configure la velocidad angular del robot.
- **setL**: Método para que el usuario configure la velocidad de elevación del robot.
- **move**: Método que combina la funcionalidad de *setV* y *setW*.

A continuación en la figura 4.6 se presenta un esquema de la ejecución de la función *setVelocity* del robot.

4.3.3. Sensores

En este apartado se explicarán cómo se inician los sensores, las funciones de acceso a los datos de los sensores y cómo funcionan internamente en el objeto robot.

Se han implementado dos tipos de sensores, sensores de infrarrojos que permiten detectar una línea que se encuentre bajo el robot y el sensor de ultrasonido que permite detectar obstáculos abarcando un radio de 180º por delante del frontal del robot.

El sensor de infrarrojos se ha implementado mediante el propio uso de la cámara por simplicidad y por mejora de rendimiento, lo que se hace internamente es recortar la imagen de la cámara hasta quedarnos con los píxeles que se encuentran más abajo en la imagen, el ancho de

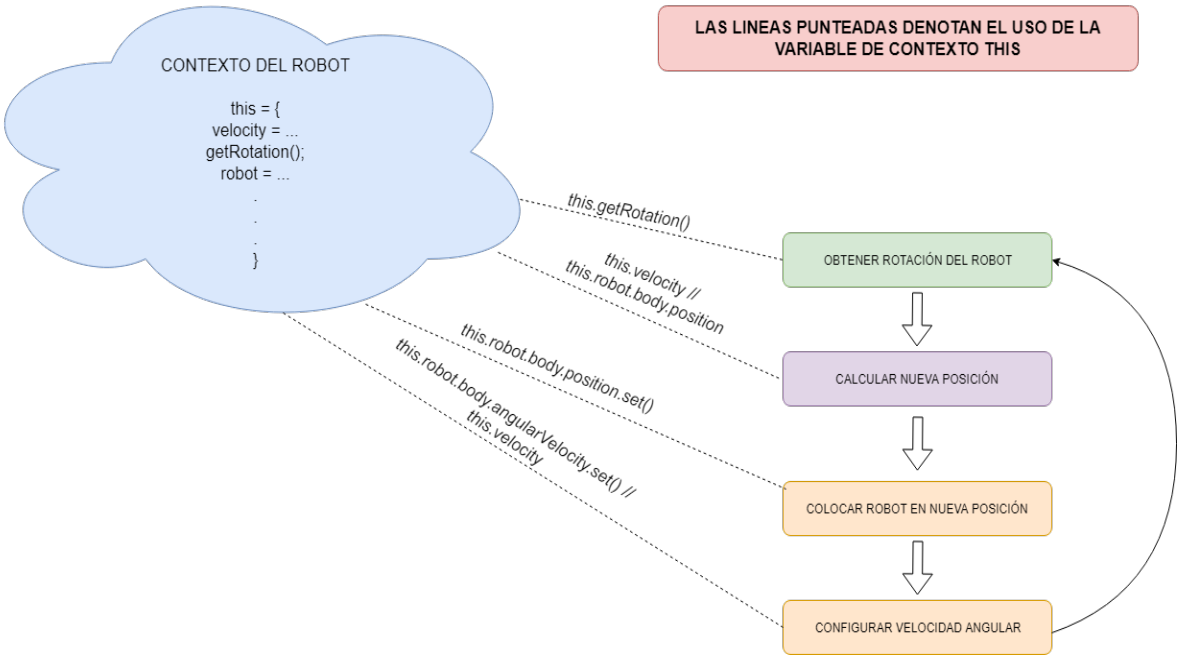

Figura 4.6: Esquema de ejecución de la función *setVelocity*, en la parte superior izquierda de la imagen se muestran algunos datos que existen en el contexto de la función.

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 *readIR* es la encargada de simular este sensor de infrarrojo, toma como parámetro de entrada un color pero como *string* y accede a la variable anteriormente mencionada *understandedColors* para obtener los filtros para el color seleccionados, después recorta la imagen para obtener una imagen de 5x150 px. Posteriormente filtra 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 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.

Esta simplificación es debida a la implementación existente en el robot real, un objetivo en el futuro es que el mismo programa creado en el robot se pueda exportar al robot real y funcione de la misma manera, por tanto ha sido necesaria crear esta adaptación.

Como ya se ha comentado en este apartado se ha creado también el sensor de ultrasonido, este sensor ha sido simulado mediante el elemento *raycaster* existente en AFRAME que se asemeja a un láser y nos permite conocer el punto de intersección entre el láser 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 respecto del robot.

- Componente AFRAME **intersectionHandler**: Este componente es el más importante ya que nos permite manejar correctamente el evento que se dispara cuando hay una intersección para un *raycaster*, los eventos tienen todos el mismo nombre más el número de identificación del *raycaster* que lanza el evento, esto se lleva a cabo para poder detectar que *raycaster* lanza el evento y poder 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 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.

- **createRaycasters**: Función que crea un *raycaster* configurando una serie de atributos como la distancia 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, para el sensor de ultrasonido.

- **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.

- **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 detectadas por los *raycasters*.

- **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 sirven para el usuario final, ambas funciones 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 ultrasonido es simple, el constructor llama a la función *startRaycasters* y esta se encarga de configurar todo lo necesario para los sensores de ultrasonido.

Como resumen, el usuario normalmente usará los métodos **getDistance, getDistances y readIR** para obtener los datos de los sensores de ultrasonido e infrarrojo respectivamente.

4.3.4. Cámara

Para la simulación de la cámara del robot se utiliza el componente *spectator* que utiliza 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 método 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

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).

- **getImage**: Esta función la usará el usuario final para obtener los datos de la cámara para realizar la lógica que se necesite.

Además de estos métodos se ofrecen una serie de métodos algo más elaborados, a continuación se enumeran y explican:

- **getObjectColor y getObjectColorRGB**: Ambas funciones resuelven el mismo problema pero con parámetros de entrada distintos, su función es la de 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. La primera función toma como parámetro de entrada el color del objeto a detectar pero como tipo *string*, la segunda función toma como valores de entrada los filtros de color que vamos a usar para detectar un elemento en la imagen.

4.4. HAL API, *Hardware Abstraction Layer*

A continuación se muestra una tabla con el API, si no es posible hacer zoom para ver la imagen ir a los links de los repositorios donde se encuentra la documentación del API en el fichero README.

Método	Descripción
.setV(velLineal)	Mueve hacia delante o atrás el robot. INPUT: - velLineal: numero con la velocidad lineal.
.setW(velAngular)	Hace girar al robot. INPUT: - velAngular: numero con la velocidad angular.
.setL(velElevacion)	Hace que el robot se mueva hacia arriba. INPUT: - velElevacion: numero con la velocidad de elevación
.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
.getImage()	Obtener la imagen de la cámara en el robot OUTPUT: cv.Mat() con la imagen de la cámara del robot.

Método	Descripción
.getObjectColor(color)	Devuelve un objeto con datos sobre el objeto que detecte la cámara con el color pasado como parámetro devuelve un objeto del siguiente tipo INPUT: -color: color como string. OUTPUT: {center: [cx, cy], area: areaInt }
.getObjectColorRGB(filtroBajo, filtroAlto)	Devuelve un objeto con datos sobre el objeto que detecte la cámara con el color pasado como parámetro devuelve un objeto del siguiente tipo INPUT: - filtroBajo: lista de longitud 4 con valores 0 a 255 (RGBA) - filtroAlto: lista de longitud 4 con valores de 0 a 255 (RGBA) OUTPUT: { center: [cx, cy], area: areaInt }

.getRotation()	Devuelve un objeto con la rotación del robot en los 3 ejes
	OUTPUT:
	{
	x: rotacionX
	y: rotacionY
	z: rotacionZ
	}

.followLine(filtroBajo, filtroAlto, velocidadLineal)	Simplificación del algoritmo del sigue lineas INPUT: Se utilizan los mismos inputs que para los metodos getObjectColorRGB() y setV()

Método	Descripción
.readIR(color)	Permite obtener valores entre 0-3 que simulan el uso de sensores infrarrojos para un color de linea pasado como parámetro INPUT: -color: color a filtrar en la imagen como string
.getDistance()	Permite obtener la distancia del objeto que tiene delante OUTPUT: number
.getDistances()	Permite obtener la distancia de los objetos detectados en un arco de 180º, devuelve 31 valores por defecto OUTPUT Lista con 31 valores de tipo number.

.getPosition()	Permite obtener la posición del robot en la escena
	OUTPUT:
	{
	x: coordenadax
	y: coordenaday
	z: coordenadaz
	theta: rotacion eje Y (horiz)
	}

```
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|.jsx)$/,
        loader: 'babel-loader',
        exclude: "/node_modules/"
      }
    ]
  },
  mode: 'development'
}
```

Figura 4.7: Fichero de configuración de la herramienta WebPack.

4.5. Empaquetado

En esta sección se explicará el empaquetado final de *WebSim* y . Para empaquetarlo 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.

Lo primero que ha sido necesario es instalar las dependencias a través de la herramienta NPM, lo que hace que pueda importar posteriormente desde el punto principal de lo que será mi empaquetado dicha dependencia. Lo que se ha hecho es desde el archivo **websim.js** importar 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 empaquetará con WebPack y gracias a esta combinación tendremos un único *bundle* en el cual se incluirán todas las dependencias. A continuación se muestra en la figura 4.7 el fichero de configuración de *WebPack* para empaquetar WebSim.

En la figura 4.7 se muestra cuál es fichero de entrada en el que WebPack buscará los *imports* y el fichero de salida de todas las dependencias empaquetadas.

Capítulo 5

Usos del simulador

Como se ha comentado en el apartado de objetivos del presente documento, los usos del simulador serán educativos. Se ha enfocado la aplicación 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 ha preparado la aplicación para desarrollar una serie de ejercicios ya establecidos previamente al proyecto. En la siguiente sección se explicarán los distintos ejercicios.

5.1. Programando 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.

5.2. Programando con *Blockly*

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 5.1 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.

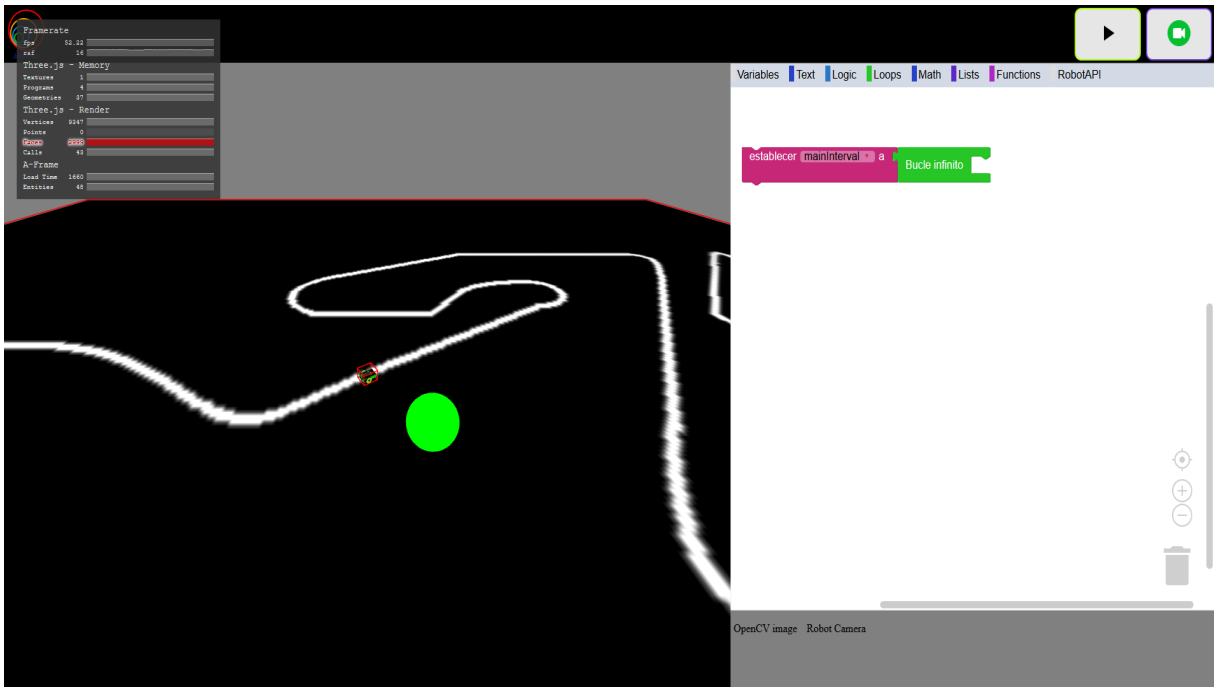

Figura 5.1: Interfaz de la aplicación WebSim + Blockly

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=iouvTDALMl8`

- Ejercicio evitar obstáculos:

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

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

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 framework AFRAME.

Apéndice A

Manual de usuario

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/`

A.1. Instalación

Para la instalación en local necesitamos tener instalados NodeJS y NPM, para ello tenemos 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

-- Para comprobar que Node está instalado:

$ nodejs --version
```

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

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

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

Muévete a la carpeta del repositorio que acabes de clonar y pon lo siguiente:

```
$ npm install
```

Esto instalará todas las dependencias en la carpeta 'node-modules', una vez finalizado esto podremos arrancar el servidor, para ello tenemos 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
$ python -m http.server [port]
-- Python v.2
$ python -m SimpleHTTPServer [port]
```

Apéndice B

Enlaces a tutoriales de uso

En el siguiente enlace se muestra el canal de Youtube de la plataforma *JdeRobot* en el que, filtrando por 'websim', se podrán encontrar vídeotutoriales de uso del simulador.

`https://www.youtube.com/channel/UCgmUgpircYAv_QhLQziHJOQ?view_as=subscriber`

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/