



GRADO EN INGENIERÍA TELEMÁTICA

Curso Académico 2019/2020

Trabajo Fin de Grado

Mejoras en entorno de robótica educativa para niños

Autor : Rubén Álvarez Martín

Tutor : Dr. José María Cañas Plaza

Resumen

Este proyecto está muy relacionado con las tecnologías web y la robótica y está enfocado a la mejora de la plataforma *Kibotics* en la cual se dispone de un simulador robótico que tiene todo su peso computacional en el navegador web.

Esta plataforma está enfocada a enseñar robótica y programación a estudiantes desde muy corta edad hasta estudiantes de secundaria, empleando el lenguaje *Scratch* en cortas edades y *Python* en más avanzadas. Con ella, además de ejecutar el código programado, se puede enviar el mismo a un robot físico haciendo la traducción a *Python*.

Como aportes genuinos de este TFG se han añadido a *Kibotics* soporte a nuevos robots tales como drones o mBot, nuevos ejercicios a partir de la funcionalidad existente y ejercicios competitivos para poder programar dos inteligencias en el mismo escenario. Estos ejercicios incorporan evaluadores automáticos que evalúan el comportamiento de los *robots* simulados y con ello la eficacia del código del estudiante. Además se ha desarrollado una página web para poder teleoperar los robots sin necesidad de programarlos y comprobar sus sensores y actuadores.

Para el desarrollo de software de todas las mejoras se han empleado herramientas como *A-Frame*, *HTML5*, *JavaScript*, *Blender* o *Blockly*. Para la gestión de dependencias del proyecto se utiliza NPM y para el empaquetado de la aplicación, WebPack.

Índice general

Lista de figuras	9
Lista de tablas	11
1. Introducción	1
1.1. Tecnologías web	1
1.1.1. HTTP	2
1.1.2. Tecnologías en cliente	2
1.1.3. Tecnologías en servidor	4
1.2. Robótica	4
1.2.1. Aplicaciones robóticas	5
1.2.2. Software en robótica	7
1.3. Robótica educativa	9
2. Objetivos y metodología	15
2.1. Objetivos	15
2.2. Metodología	16
2.2.1. Repositorios GitHub	16
2.3. Plan de trabajo	18
3. Herramientas	19
3.1. JavaScript	19
3.2. A-Frame	20
3.2.1. HTML y primitivas	21
3.2.2. Entidad, Componente y Sistema	22

3.3.	Blockly	23
3.3.1.	Traductor de código	23
3.3.2.	Bloques personalizados	24
3.4.	Gestores de paquetes	26
3.4.1.	NPM	26
3.4.2.	Webpack	27
3.5.	Blender	27
3.5.1.	COLLADA	29
3.5.2.	glTF	29
3.6.	Simulador WebSim	30
3.6.1.	Diseño	31
3.6.2.	Robot en WebSim	32
3.6.3.	Drivers de sensores	34
3.6.4.	Drivers de actuadores	36
4.	Mejoras a WebSim	39
4.1.	Soporte a drones y nuevos robots	39
4.1.1.	Driver del drone e interfaz de programación en <i>JavaScript</i>	39
4.1.2.	Modelo 3D, apariencia	42
4.1.3.	Bloques Scratch para programación gráfica del drone	44
4.1.4.	Nuevos robots	47
4.2.	Teleoperadores en WebSim	48
4.2.1.	Interfaz gráfica	48
4.2.2.	Arquitectura	51
4.2.3.	WebSim y sus ficheros de configuración	54
4.3.	Nuevos ejercicios individuales	56
4.3.1.	Sigue-líneas visión	56
4.3.2.	Sigue-líneas infrarrojos	57
4.3.3.	Choca-gira	59
4.3.4.	Sigue-pelota	60
4.3.5.	Atraviesa-bosque	62

ÍNDICE GENERAL	7
4.3.6. Cuadrado con drone	64
4.4. Ejercicios competitivos	64
4.4.1. Arquitectura de cómputo	65
4.4.2. Atraviesa-bosque competitivo	70
4.4.3. Sigue-líneas competitivo	73
4.4.4. Gato-ratón	74
5. Conclusiones	79
5.1. Conclusiones	79
5.2. Mejoras futuras	80
Bibliografía	83

Índice de figuras

1.1.	Comunicación cliente-servidor en HTTP	3
1.2.	Mapa generado por un robot <i>Roomba</i>	5
1.3.	Robot médico <i>Da Vinci</i>	6
1.4.	Sensores en conducción semi-autónoma de un coche <i>Tesla</i>	6
1.5.	Robots de logística de <i>Amazon</i>	7
1.6.	Robot <i>Curiosity</i> de la <i>NASA</i>	7
1.7.	Interfaz gráfica de <i>Scratch</i>	10
1.9.	Interfaz gráfica de <i>Kodu</i>	11
1.10.	Interfaz gráfica de <i>Snap!</i>	12
1.11.	Interfaz gráfica de <i>AppInventor</i>	12
1.12.	Ejemplo de <i>robot</i> montado con <i>Mindstorms</i>	13
1.13.	mBot de la plataforma <i>Makeblock</i>	13
2.1.	Representación gráfica de las ramas del proyecto	17
3.1.	Escenario a-frame	22
3.2.	Bloque personalizado	25
3.3.	Herramienta para creación de bloques personalizados	26
3.4.	Interfaz gráfica de <i>Blender</i>	28
3.5.	Modelo glTF añadido en un escenario de A-Frame	30
3.6.	Interfaz gráfica de <i>WebSim</i>	32
4.1.	Sistema de ejes en A-Frame	40
4.2.	Drone en Blender	43
4.3.	Escenario de WebSim con drone integrado	44

4.4.	Modelos de <i>drone</i> de distintos colores	44
4.5.	Bloque de velocidad de ascenso	44
4.6.	Bloque de velocidad de descenso	45
4.7.	Bloque de aterrizaje	45
4.8.	Bloque de despegue	45
4.9.	Espacio de trabajo de <i>Scratch</i> con los bloques del drone incorporados	47
4.10.	Modelos de coches Fórmula 1	47
4.11.	Modelo mBot	48
4.13.	Interfaz que permite acceder a cada uno de los teleoperadores	50
4.14.	Arquitectura de la aplicación teleoperadores	52
4.15.	Escenario para el ejercicio <i>piBot</i> sigue-líneas con cámara	57
4.16.	Solución en <i>Scratch</i> para el ejercicio sigue-líneas visión	57
4.17.	Escenario para el ejercicio para el robot <i>piBot</i> sigue-líneas infrarrojo	58
4.18.	Solución en <i>Scratch</i> para el ejercicio sigue-líneas infrarrojos	58
4.19.	Escenario para el ejercicio choca-gira	59
4.20.	Solución en <i>Scratch</i> para el ejercicio choca-gira	59
4.21.	Secuencia de la animación de una pelota para el ejercicio <i>drone</i> sigue-pelota . .	61
4.22.	Solución en <i>Scratch</i> para el ejercicio sigue pelota drone	62
4.23.	Escenario para el ejercicio atraviesa bosque	63
4.24.	Solución en <i>Scratch</i> para el ejercicio atraviesa bosque	63
4.25.	Escenario de WebSim para el ejercicio drone cuadrado	64
4.26.	Solución en <i>Scratch</i> para el ejercicio cuadrado drone	64
4.27.	Editor de <i>JavaScript</i> para ejercicios competitivos	65
4.28.	Editor de <i>Scratch</i> para ejercicios competitivos	67
4.29.	Escenario y evaluador para el ejercicio atraviesa-bosque	70
4.30.	Ejercicio y evaluador sigue-líneas competitivo	74
4.31.	Evaluador y escenario con dos robots para ejercicio gato-ratón	78

Índice de cuadros

3.1. Métodos (HAL API) de los sensores del robot.	36
3.2. Métodos (HAL API) de los actuadores del robot.	37
4.1. Métodos (HAL API) de los actuadores implementados para el drone.	42

Capítulo 1

Introducción

En este capítulo se introducen los conceptos básicos en los que se apoya este proyecto. Se explican y se ponen en contexto las tecnologías web y el estado actual de la robótica y su expansión hasta llegar al punto en el que se encuentra, haciendo especial mención a la robótica educativa en la que se centra el proyecto.

1.1. Tecnologías web

Las tecnologías web han ido evolucionando a lo largo de los últimos años. Se basan fundamentalmente en un modelo cliente-servidor. Actualmente lo más común son las *aplicaciones web*, que son herramientas ejecutadas mediante un navegador *web* en las que los datos son procesados y almacenados en un servidor. Dentro de estas aplicaciones se pueden encontrar sistemas de correo electrónico (*Gmail*¹ o *Outlook*²), tiendas *online* (*Amazon*³), distribución de contenidos (*Netflix*⁴ o *Spotify*⁵) o *wikis*(*Wikipedia*⁶).

¹<https://mail.google.com/mail/>

²<https://outlook.live.com>

³[https://www.amazon.es/](https://www.amazon.es)

⁴[https://www.netflix.com/](https://www.netflix.com)

⁵[https://www.spotify.es/](https://www.spotify.es)

⁶[https://www.wikipedia.org/](https://www.wikipedia.org)

1.1.1. HTTP

HiperText Transfer Protocol (HTTP) es el protocolo de nivel de aplicación utilizado para transferir recursos hipermedia entre ordenadores y sigue el esquema petición-respuesta entre cliente y servidor (Figura 1.1). En esta comunicación el cliente abre una conexión *TCP* con el servidor y envía un mensaje de petición *HTTP* y, por parte del servidor, responde al cliente con un mensaje *HTTP* y cierra la conexión *TCP*. El protocolo *HTTP* no mantiene estado. Es decir, el servidor trata cada petición de manera aislada y no almacena información sobre peticiones realizadas por un mismo cliente. Las peticiones están definidas por el protocolo y tienen métodos concretos:

- GET: Solicita un recurso al servidor especificando su *URL*.
- HEAD: Método similar a GET con la diferencia de que únicamente solicita las cabeceras y no descarga el recurso completo.
- POST: Envía datos al servidor, normalmente un recurso específico que provoca un cambio de estado.
- PUT: Actualiza información sobre un recurso del servidor.
- DELETE: Elimina en el servidor un recurso.

Aunque estos son los principales métodos, el protocolo tiene flexibilidad para ir añadiendo nuevos e incorporar funcionalidad. El número de métodos ha ido aumentando con las nuevas versiones.

La versión actual del protocolo es del 2 de abril de 2019 (*HTTP/2.4.38*) pero la primera versión es del 1991, que es cuando aparece un estándar para la publicación de páginas web mediante un lenguaje de marcas de hipertexto: *HTML*.

1.1.2. Tecnologías en cliente

Es la parte encargada de dar forma a la interfaz de usuario y de establecer la comunicación con el servidor. Una pieza importante del cliente es el navegador, ya que es el encargado de

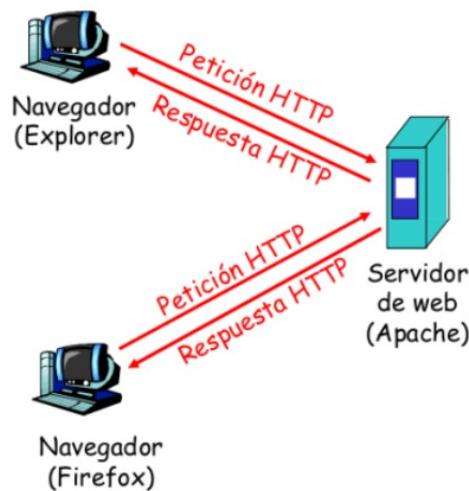


Figura 1.1: Comunicación cliente-servidor en HTTP

leer e interpretar la información recibida. Entre los navegadores más empleados se encuentran *Firefox*⁷, *Google Chrome*⁸ u *Opera*⁹. Las tecnologías más utilizadas son:

- **HTML:** es el estándar más utilizado para el desarrollo de páginas web. Actualmente los navegadores usan la versión *HTML5*, que incluye muchas mejoras respecto a su predecesor: *canvas*, *websockets*, *WebRTC*, vídeo, audio, etc. Este lenguaje indica la estructura de una página web, para editar el estilo y presentación visual hay que hacer uso de otros elementos como *CSS*.
- **Cascading Style Sheets (CSS):** es un lenguaje de diseño gráfico para definir y crear la presentación de un documento escrito en un lenguaje de marcado. De esta forma, se puede separar información y datos (en los documentos *HTML*) y todo lo relativo al diseño y presentación (en documentos *CSS*). Actualmente los navegadores usan la versión *CSS3*.
- **JavaScript:** para programar la lógica, dinamismo e interacción con el usuario es el lenguaje más común y extendido en el desarrollo de páginas. Debido a su importancia en el proyecto, se explicará más a fondo en próximos capítulos.

⁷<https://www.mozilla.org/>

⁸<https://www.google.com/intl/es/chrome/>

⁹<https://www.opera.com/>

1.1.3. Tecnologías en servidor

Son las encargadas de dar forma al servidor web de manera que permiten el acceso a herramientas como base de datos, conexiones de red o recursos compartidos. O, dicho de otra forma, se ocupan de realizar las tareas necesarias para hacer posible crear una aplicación que visualizará el cliente. Las más utilizadas son:

- *Node.js*: es una forma de ejecutar *JavaScript* en el servidor. Proporciona un entorno de ejecución del lado del servidor que compila y ejecuta a gran velocidad. Esto es debido a que compila en código máquina nativo en lugar de interpretarlo o ejecutarlo.
- *Django*: entorno web de alto nivel programado en *python* diseñado para realizar aplicaciones de cualquier complejidad. Es seguro, rápido y escalable. Además, incluye una interfaz para acceder a bases de datos lo que facilita las consultas al no tener que manejar *SQL* y realizarlas con filtros de *Python*.
- *Spring*: herramienta basada en *Java* cuya finalidad es simplificar el desarrollo de aplicaciones ya que facilita la configuración de la aplicación y el despliegue en servidor. De esta manera, *Spring* está pensado para aplicaciones web, servicios *REST*, análisis de datos e integración de sistemas.

1.2. Robótica

La robótica es una rama tecnológica encargada del diseño y construcción de aparatos que realizan operaciones y trabajos en sustitución de la mano de obra humana.

Un robot es un sistema autónomo programable capaz de realizar tareas de ayuda al ser humano y con aplicaciones en campos diversos como la medicina, el hogar, las fábricas, etc.

Los robots se componen de sensores, controladores y actuadores.

- **Sensores**: son los encargados de recoger información del entorno. En este grupo se encuentran láseres, cámaras, ultrasonidos u odómetros. Estos dispositivos equivalen a los sentidos humanos.
- **Controladores**: analizan los datos recogidos por los sensores y elaboran una respuesta que va a ser enviada a los actuadores. En los seres humanos equivale al cerebro.

- **Actuadores:** se encargan de transformar energía eléctrica, hidráulica o neumática en mecánica. Son los que interactúan con el entorno y equivalen a los músculos humanos.

1.2.1. Aplicaciones robóticas

La robótica es ya una realidad. Los robots no son utilizados únicamente por empresas para labores industriales, si no que se pueden encontrar en hogares y en la vida cotidiana de las personas. Esto es debido al aumento de eficiencia que generan, reducción de costes o el control de errores que aportan. Algunas ejemplos de *robots* empleados en la actualidad, tanto en ámbito doméstico como en industrial, son los siguientes:

- Aspiradora robótica *Roomba*, robot autónomo que, en los modelos más avanzados, reconocen el entorno, trazan un mapa de la casa y vacían su depósito automáticamente.

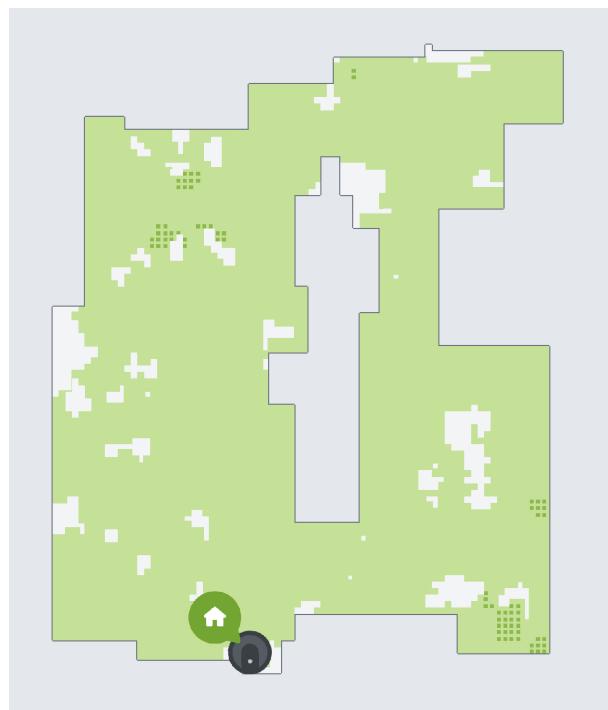


Figura 1.2: Mapa generado por un robot *Roomba*

- Robot médico *Da Vinci*, que permite al cirujano operar a través de una consola mejorando así su precisión y reduciendo riesgos en operaciones quirúrgicas.



Figura 1.3: Robot médico *Da Vinci*

- Vehículos autónomos *Tesla*, que mediante cámaras y sensores de ultra-sonidos analizan el entorno para garantizar una conducción autónoma segura.

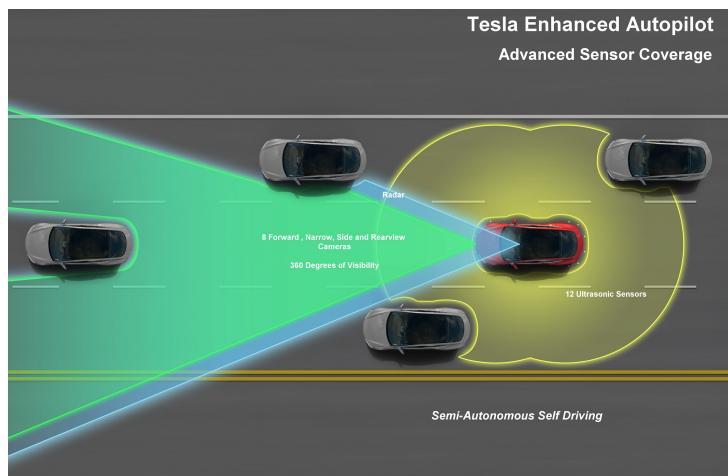


Figura 1.4: Sensores en conducción semi-autónoma de un coche *Tesla*

- Robots de logística de *Amazon*, que se encargan, de manera autónoma, de localizar la estantería donde se encuentra un paquete solicitado y la desplaza por todo el centro logístico al lugar de destino.

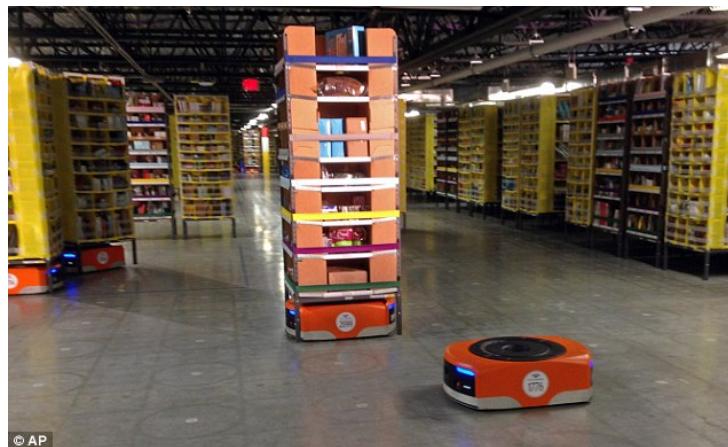


Figura 1.5: Robots de logística de *Amazon*

- Robot *Curiosity* de la *NASA*, empleado para la exploración de la superficie del planeta Marte.



Figura 1.6: Robot *Curiosity* de la *NASA*

1.2.2. Software en robótica

Para dotar de inteligencia autónoma a los robots se requiere desarrollar sistemas complejos, aplicaciones e infraestructuras. Por ejemplo, hace años, el desarrollo de *software* se realizaba adoptando soluciones “*ad-hoc*” dotando a cada robot de un diseño específico y con sensores y actuadores concretos. Esto implica que había que implementar todo el *software* para un nuevo robot debido a que no se podía aplicar el desarrollado anteriormente. En la actualidad, existen plataformas que permiten el desarrollo de aplicaciones robóticas de forma eficiente y genérica

permitiendo así reutilizar aplicaciones creadas en otros robots.

Como se ha dicho, se debe desarrollar un *software* para dotar de inteligencia a un *robot*, que habitualmente se programa con ayuda de herramientas, como los *middleware* robóticos. El uso de estas herramientas permiten introducir una capa de abstracción con los *drivers* y el *hardware* del robot, reduciendo la complejidad y los conocimientos necesarios para realizar desarrollos. Algunas de estos *middleware* son:

- **Internet Communication Engine (ICE)**¹⁰. Plataforma de *software* libre orientada a objetos que proporciona las herramientas para simplificar las comunicaciones entre componentes. Usado para la comunicación entre nodos, proporciona una capa que se encarga de abrir y cerrar conexiones entre ellos sin necesidad de usar el protocolo *HTTP*.
- **Robot Operating System (ROS)**¹¹. Plataforma de *software* libre para el desarrollo de *software* de robots. Provee servicios estándar de un sistema operativo como la abstracción de *hardware*, control de dispositivos de bajo nivel, mecanismos de intercambio de mensajes entre procesos y una serie de herramientas utilizadas en robótica.
- **Orca**¹². Plataforma de *software* libre dedicada al desarrollo de aplicaciones robóticas. Dedicada principalmente al desarrollo de componentes, que se pueden ejecutar de manera independiente o se pueden unir para formar sistemas robóticos de distintas complejidades. Permite reutilizar código y emplear componentes robóticos ya creados.
- **Orocos**¹³. Proyecto de *software* libre dedicado al control de robots y máquinas. Está orientado a componentes y permite añadir funcionalidad de manera sencilla y sin recopilar todo el código. Incluye paquetes complementarios como filtros de *Bayes*, librerías de control dinámico y cinemático o visión.

¹⁰<https://zeroc.com/products/ice>

¹¹<https://www.ros.org/>

¹²<http://orca-robotics.sourceforge.net/>

¹³<https://orocos.org/>

1.3. Robótica educativa

La robótica con fines educativos está empezando a adquirir importancia en la enseñanza porque su aprendizaje está disponible para estudiantes de cualquier nivel. Este método de enseñanza intenta despertar el interés de los alumnos porque, gracias a la innovación que posibilita la tecnología, añade un componente atractivo e integrador a las asignaturas tradicionales. En 2015 la comunidad de Madrid introdujo la asignatura de robótica en los planes docentes de Enseñanza Secundaria y en el curso 2020-2021 se empezará a implantar en Educación Primaria la asignatura “Tecnología, programación y robótica”.¹⁴.

El método más empleado para este tipo de enseñanza es la educación *STEM* (*Science, Technology, Engineering and Mathematics*). Este tipo de educación promueve una cultura de pensamiento científico, así como la adquisición de conocimientos tecnológicos aplicables a situaciones reales y permite desarrollar competencias para la resolución de problemas y un pensamiento creativo y crítico.

La enseñanza en centros escolares se realiza en gran parte mediante plataformas como la creada por LEGO o placas Arduino que simplifican el aprendizaje y resulta motivadora para el alumno porque obtiene resultados vistosos y se le puede dar una aplicación real.

Resulta importante acercar la robótica a alumnos en edades muy tempranas, partiendo de nociones básicas, dada la importancia que está adquiriendo la disciplina y la fuerte presencia que tiene en la mayor parte de los sectores laborales. Además, la capacidad de programar es una parte importante en la sociedad actual ya que, con esta habilidad, se aprenden estrategias para resolver problemas, diseñar proyectos y comunicar ideas.

Las nociones básicas de las que se sirve este acercamiento son los lenguajes de programación visual. Se trata de lenguajes potentes y muy intuitivos que abstraen al alumno de la complejidad que implica la sintaxis y aportan un entorno visual haciendo que el software sea bien aceptado por estudiantes de corta edad. Además permiten a los usuarios programar mani-

¹⁴<https://n9.cl/7lqc>

pulando elementos gráficos en lugar de hacerlo textualmente. Los lenguajes de programación visual más importantes son:

- *Scratch*¹⁵: proyecto liderado por el *MIT*, es utilizado por estudiantes y docentes para programar animaciones, juegos e interacciones fácilmente gracias a su interfaz visual. Se trata de un lenguaje donde los programas se construyen ensamblando bloques gráficos y cada bloque es el equivalente a una función o método de cualquier lenguaje de programación. Actualmente está en su versión *Scratch 3.0* y en la figura 1.7 se puede ver un ejemplo de su interfaz gráfica.

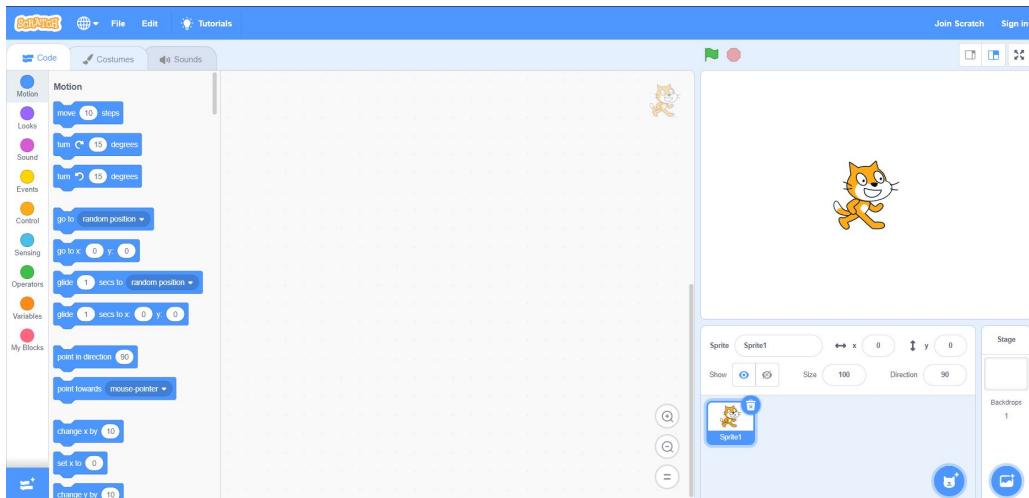
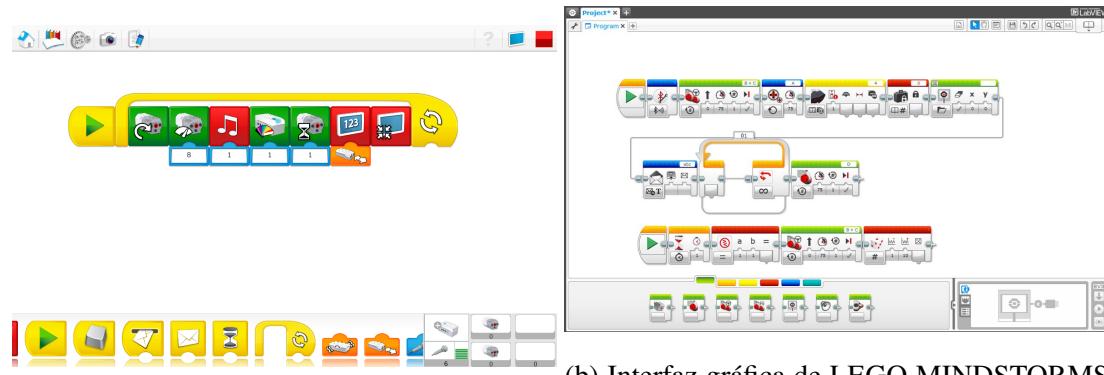


Figura 1.7: Interfaz gráfica de Scratch

- *LEGO*: dispone de una amplia gama de robots programables y cada uno de ellos tiene un sistema de programación bajo interfaz gráfica. En las figuras 1.8a y 1.8b se pueden ver dos ejemplos de distintos software para distintos robots.

¹⁵<https://scratch.mit.edu/>



(a) Interfaz gráfica de LEGO WeDo 2.0

(b) Interfaz gráfica de LEGO MINDSTORMS EV3

- *Kodu*¹⁶: lenguaje de programación visual creado por Microsoft para desarrollar videojuegos. Diseñado para ser muy accesible y agradable para cualquier usuario siendo un lenguaje basado en reglas, condiciones y acciones prescindiendo de muchas convenciones de programación como bucles, subrutinas o variables simbólicas. Permite a los más jóvenes a ser creadores de sus propios videojuegos.



Figura 1.9: Interfaz gráfica de Kodu

- *Snap!*¹⁷: basado en *Scratch*, sigue su facilidad para aprender a programar pero su uso se

¹⁶<https://www.kodugamelab.com>¹⁷<https://snap.berkeley.edu>

concentra en edades algo más avanzadas. Accesible desde cualquier navegador al estar programado en *JavaScript*.

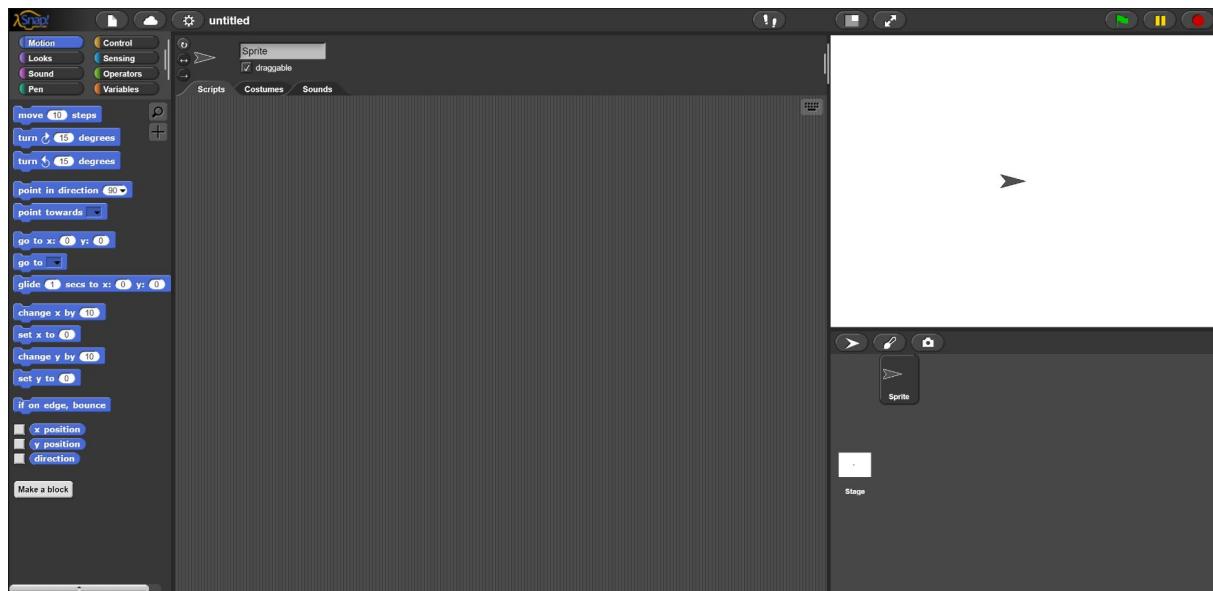


Figura 1.10: Interfaz gráfica de Snap!

- *AppInventor*¹⁸: Entorno de desarrollo de software creado por *Google* en colaboración con el *MIT* destinado a la elaboración de aplicaciones *Android*. El usuario puede desarrollar de forma visual y con ayuda de bloques una aplicación que cubre un gran número de necesidades básicas en un dispositivo móvil.

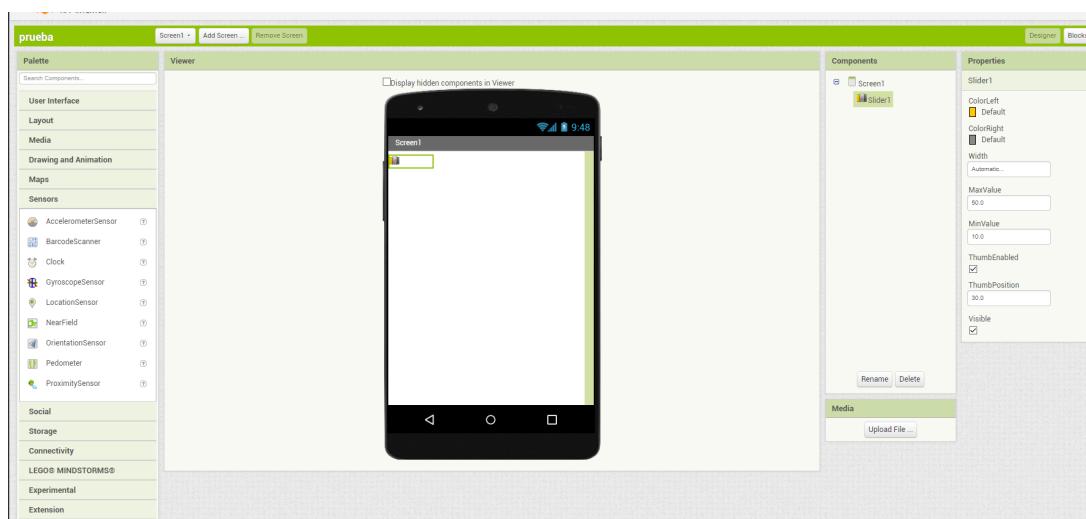


Figura 1.11: Interfaz gráfica de AppInventor

¹⁸<http://ai2.appinventor.mit.edu/>

También son muy habituales el uso de plataformas hardware, que incorporan los elementos básicos para la construcción de un robot. Las más empleadas son:

- *LEGO mindstorms*: plataforma dedicada a construir *robots* con piezas de LEGO. También incluye un microprocesador para ser programado, sensores (infrarrojos, táctiles y de color) y motores.



Figura 1.12: Ejemplo de *robot* montado con *Mindstorms*

- *Makeblock*¹⁹: plataforma de construcción robótica muy accesible que permite la creación de *robots* con *kits* dirigidos a todo tipo de público. El *robot* que se construye con estos *kits* es el *mBot*:



Figura 1.13: mBot de la plataforma *Makeblock*

- *Arduino*²⁰: Plataforma de creación electrónica. Permite crear microordenadores a los que dar multitud de usos con una sola placa, que dispone de entradas para periféricos y un microcontrolador.

¹⁹<https://makeblock.es/>

²⁰<https://www.arduino.cc/>

Capítulo 2

Objetivos y metodología

Una vez expuestas las motivaciones y contexto del proyecto, en este capítulo se detallarán objetivos y metodología empleada.

2.1. Objetivos

El propósito de este proyecto es la extensión y mejora de la herramienta docente basada en el simulador *WebSim* que está orientada a facilitar el aprendizaje de programación y robótica. Para cumplir ese propósito se han fijado varios subobjetivos:

- Añadir soporte para *drone* en la plataforma, incluyendo tanto su simulación realista en su apariencia visual y comportamiento físico como la infraestructura para que sea programable desde fuera del simulador.
- Añadir teleoperadores para que los robots se puedan controlar sin necesidad de programar. De esta manera se facilita la labor de los desarrolladores al poder probar el entorno y los sensores del robot de manera sencilla.
- Incluir más ejercicios sobre *WebSim*. Siendo necesario elaborar archivos de configuración para poder cambiar entre los distintos ejercicios y robots soportados. Esto incluye añadir más modelos de robots y nuevos escenarios a la plataforma.
- Incluir *ejercicios competitivos* de tal manera que dos usuarios puedan programar sobre el mismo escenario. Este objetivo también incluye crear un evaluador automático para puntuar la eficacia del comportamiento de cada robot.

2.2. Metodología

Para el desarrollo del proyecto se han hecho reuniones semanales con el tutor del TFG en las que se elaboraba un plan de trabajo para la semana y se revisaban las tareas concluidas. Cuando era necesario tener un desarrollo terminado, se aumentaba la frecuencia de las reuniones. Este tipo de trabajo se asemeja mucho a la metodología *extreme programming*.

Esta forma de trabajar es una metodología *agile* con el objetivo de conseguir un código de calidad y flexible y mejorar la productividad. Es un método muy útil para proyectos con requisitos cambiantes porque hace énfasis en la adaptabilidad. Para su cumplimiento son fundamentales la comunicación y realimentación con el resto de integrantes del equipo. Para ello se ha utilizado la herramienta *Slack*¹ en la los desarrolladores están en contacto en todo momento.

2.2.1. Repositorios GitHub

Como en la mayor parte de proyectos en la que se desarrolla *software*, se ha utilizado *GitHub*; un sistema de control de versiones que permite llevar un registro de los cambios efectuados del código del proyecto y administrarlo. Facilita trabajar en colaboración con otras personas, planificar proyectos y realizar un seguimiento el trabajo.

Los archivos de cada proyecto se almacenan en repositorios, que pueden estar en local o ubicado en el almacenamiento de *GitHub*. Para el desarrollo de este proyecto se han utilizado dos repositorios: en el que se desarrolla el software principal del proyecto² y en el que participa un equipo de seis personas y otro personal³ en el que se realizaba un registro semanal de los avances en el *README.md* del repositorio.

En el primer repositorio, la metodología de trabajo durante el proyecto ha consistido en crear incidencias (*issues*) de alguna tarea en específico (con el fin de solucionar problemas o añadir funcionalidad) y para cerrarla se creaba una rama (*branch*) realizando después un parche (*pull request*) para fusionarlo con la rama principal y así arreglar la incidencia. Se sigue esta meto-

¹<https://slack.com/>

²<https://github.com/jderobot-hub/kibotics-websim>

³<https://github.com/RoboticsLabURJC/2019-tfg-ruben-alvarez>

dología para registrar de forma limpia los cambios realizados por los diferentes desarrolladores que trabajan sobre el repositorio y, en caso de ser una modificación importante, solicitar la supervisión de otro desarrollador para integrarla. En la figura 2.1 se puede observar gráficamente una cronología de las ramas del repositorio durante medio mes de trabajo.

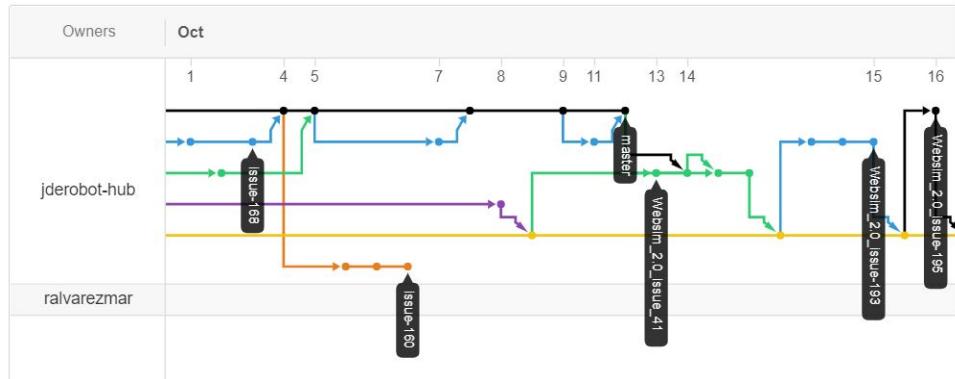


Figura 2.1: Representación gráfica de las ramas del proyecto

En el segundo repositorio se realizó una copia del repositorio (*fork*) para trabajar directamente sobre la cuenta personal de *GitHub*⁴. De esta manera, en primera instancia se suben los cambios al repositorio personal y, cuando hay progresos importantes, se suben todos los cambios al repositorio original. Para automatizar esta tarea se ha realizado un *script* de *shell* en el que el primer argumento es el mensaje del *commit* y, si se escribe '*-t*' después de este, se realiza la subida al repositorio copiado y al original.

```

1 #!/bin/sh
2 if [ $# -gt 2 ]
3 then
4 echo "usage: $1" 1>&2
5 exit 1
6 fi
7 git add .
8 git commit -m "$1"
9 git push
10 if [ "$2" = "-t" ]
11 then
12 git push upstream
13 fi

```

Listing 2.1: *Script* para subir código a *GitHub*

⁴<https://github.com/ralvarezmar/2019-tfg-ruben-alvarez>

2.3. Plan de trabajo

Se ha establecido un plan de trabajo dividido en fases para afrontar los objetivos previstos:

- Fase 1: Estudio de *A-Frame* y *Websim*. Periodo de aprendizaje y familiarización con el entorno en el que se va a trabajar.
- Fase 2: Desarrollo de soporte a *drones* y nuevos modelos. Incluye la ampliación de los drivers y creación de bloques.
- Fase 3: Desarrollo de teleoperadores. Incorpora el diseño de un frontal para seleccionar teleoperador entre los modelos disponibles de *WebSim*.
- Fase 4: Ejercicios individuales. Se lleva a cabo desarrollo, creación y prueba de nuevos escenarios.
- Fase 5: Ejercicios competitivos. Comprende la creación de escenarios y nuevos modelos.
- Fase 6: Evaluadores automáticos. En la que se realiza el diseño de cada evaluador y la lógica para su correcto funcionamiento.

Capítulo 3

Herramientas

En este capítulo se van a detallar las herramientas y tecnologías utilizadas en el desarrollo de este proyecto principalmente en el ámbito web. Algunas se han elegido por facilidad de uso y otras por necesidad del entorno desarrollado.

3.1. JavaScript

JavaScript es un lenguaje interpretado de alto nivel que se encuentra bajo el estándar *ECMAScript*¹ y está basado en otros lenguajes de programación como Java o C.

En su principio fue concebido como lenguaje de aplicaciones web para el lado cliente, interpretado en un navegador web, permitiendo mejorar la interfaz de usuario y realizar páginas web dinámicas.

En la actualidad, *JavaScript* se ha ido extendiendo hacia el lado servidor con *Node.js* y es por ello que es el lenguaje más utilizado para desarrollo web y todos los navegadores interpretan el código integrado en las páginas web.

Para este proyecto se ha usado *ECMAScript 6* o *ECMAScript 2015* y, como el proyecto se orienta a la creación de una aplicación con todo el peso en el lado del cliente, *JavaScript* es el lenguaje que mejor se adapta a los requerimientos del desarrollo. De esta manera, se ha programado toda la inteligencia de la aplicación web, que corre en el navegador.

Las siguientes características son las principales de *ECMAScript*:

¹Especificación de lenguaje de programación el cual define tipos dinámicos y soporta características de programación orientada a objetos.

- Es un lenguaje estructurado, tiene gran similitud con *C* y comparte gran parte de su estructura (bucles, condicionales, sentencias...) a excepción del alcance de sus variables. En *C* su ámbito es el bloque en el que fue definida y, en su origen, *JavaScript* tenía un alcance global en las variables definidas. Es en ECMAScript 2015 cuando se añade la palabra clave *let*, que incorpora compatibilidad con *block scoping* (alcance de la variable en el bloque en la que es definida).
- Tipado débil, por el cual el tipo de datos está asociado al valor, no a la variable. Esto significa que una variable puede ser *number* o *string* en distintos momentos de ejecución.
- Formado en su totalidad por objetos, en los cuales los nombres de sus propiedades son claves de tipo cadena siendo *objeto.a = 1* y *objeto['a'] = 1* equivalentes.
- Lenguaje interpretado, es por esto que no requiere un compilador ni crear un fichero binario del código; cada navegador tiene su intérprete que se encarga de ejecutarlo.
- Evaluación en tiempo de ejecución gracias a la función *eval*, la cual evalúa un código en *JavaScript* representado como una cadena de caracteres.

3.2. A-Frame

A-Frame es un entorno de código abierto destinado a crear experiencias de realidad virtual a partir de *HTML* de forma que sea sencillo de leer y comprender. De esta manera es accesible para crear una gran comunidad. *A-Frame* está en constante desarrollo, pero la versión utilizada en este proyecto es la última versión estable (*v0.9.2*) y se ha utilizado este framework por la facilidad de crear escenarios tridimensionales a través de un documento *HTML* y de añadir modelos sofisticados en distintos formatos.

Además tiene compatibilidad con *Vive*, *Rift*, *Windows Mixed Reality*, *Daydream*, *GearVR* y *CardBoard* así como soporte para todos los controladores respectivos. También ofrece soporte para ordenadores de escritorio y para la mayoría de teléfonos inteligentes.

3.2.1. HTML y primitivas

A-Frame se basa en *HTML* y el *DOM* usando un *polyfill*² para elementos personalizados. *HTML* es un componente básico para Web y como tal, tiene una gran accesibilidad como lenguaje. Para crear una escena de realidad virtual con *A-Frame* no se requiere ninguna instalación y simplemente con la creación del *HTML* se puede abrir en el navegador. La mayoría de herramientas existentes (como *React*, *Vue.js*, *d3.js* y *jQuery*) funcionan en este entorno.

HTML y *DOM* son solo la capa más externa del framework, debajo se encuentra el componente *three.js* en el que está basado *A-Frame* gracias al cual un componente puede ser utilizado en distintas entidades. De esta manera hace posible seguir el principio de programación *Don't Repeat Yourself* ya que, una vez registrada una primitiva, se puede hacer referencia al elemento todas las veces que sea necesario.

A-Frame proporciona elementos como *a-box* o *a-sky* llamados primitivas. Podemos crear un escenario a través de estas primitivas como el mostrado en la figura 3.1 con el siguiente código:

```

1  <!DOCTYPE html>
2  <html>
3      <head>
4          <meta charset="utf-8">
5          <title>Escenario primitivas</title>
6          <script src="https://aframe.io/releases/0.9.2/aframe.min.js"></script>
7      </head>
8      <body>
9          <a-scene background="color: #FAFAFA">
10             <a-box position="-1 0.5 -3" rotation="0 45 0" color="#4CC3D9" shadow></a-box>
11             <a-sphere position="0 1.25 -5" radius="1.25" color="#ff0000" shadow></a-sphere>
12             <a-cylinder position="1 0.75 -3" radius="0.5" height="1.5" color="#FFC65D" shadow></a-
13                 cylinder>
14             <a-plane position="0 0 -4" rotation="-90 0 0" width="4" height="4" color="#1cde83"
15                 shadow></a-plane>
16             <a-sky color="#e7e1e0"></a-sky>
17         </a-scene>
18     </body>
19 </html>
```

Listing 3.1: Código con primitivas que representa un escenario

²Fragmento de código en *JavaScript* utilizado para proporcionar una funcionalidad moderna en navegadores antiguos.

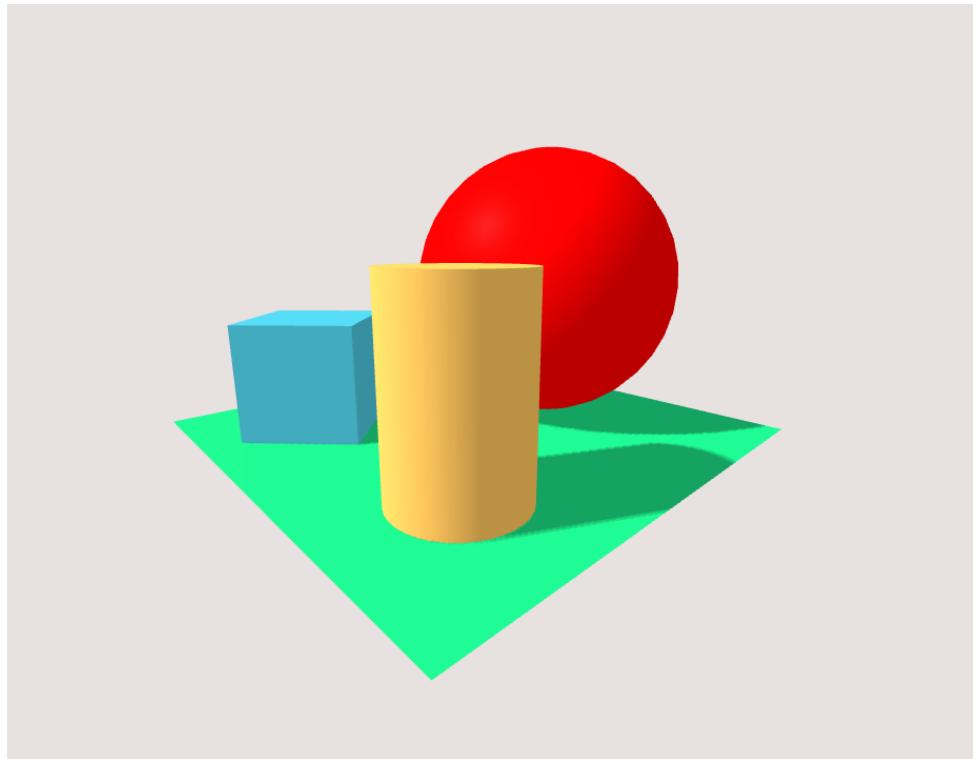


Figura 3.1: Escenario a-frame

A-Frame, además de disponer primitivas como las mostradas, hace posible la creación de primitivas para poder elaborar escenas lo más completas posible. También se pueden incluir entidades más complejas a partir de modelos 3D en formatos como *gltf*, *obj* o *collada* de los cuales se hablará en siguientes apartados.

3.2.2. Entidad, Componente y Sistema

Como ya se ha comentado, A-Frame es un entorno *three.js* con una arquitectura de entidad-componente-sistema (*ECS*). Es un patrón común en 3D y desarrollo de juegos que siguen la composición sobre el principio de herencia y jerarquía. Algunos beneficios que *ECS* aporta son mayor flexibilidad al definir objetos, gran escalabilidad o eliminación de problemas de largas cadenas de herencia. Existe un *API* que representa cada pieza de *ECS*:

- Una entidad se representa con la etiqueta *a-entity*.

```
1 <a-entity geometry="primitive: box" material="color: red">
```

En este ejemplo se hace uso de *a-entity* para crear una caja de color rojo.

- Un componente se representa como un atributo de HTML. Cada componente es un objeto que tiene un esquema, manejadores y métodos. Para registrar componentes se utiliza el método de A-Frame *registerComponent*.

```

1   AFRAME.registerComponent('example', {
2     init: function () {
3       var el = this.el;
4       el.setObject3D('mesh', new THREE.Mesh());
5       el.getObject3D('mesh'); // Returns THREE.Mesh that was just created.
6     }
7   });

```

Listing 3.2: Código para registrar un componente

- Un sistema es el representado por atributos HTML mediante la etiqueta *a-scene*. Se registran de manera similar a un componente; gracias al método *registerSystem*.

3.3. Blockly

Blockly es una librería que añade un editor de código visual a aplicaciones web y móviles. Utiliza bloques gráficos para representar conceptos complejos de código de manera más sencilla. De esta manera permite a los usuarios aplicar principios de programación sin tener que preocuparse por la sintaxis y ayuda a iniciarse y a aprender a programar a estudiantes de temprana edad. Esta librería es un proyecto de *Google* y está diseñada por las personas que están detrás de *Scratch* del *MIT* y construido sobre su base de código. Actualmente se encuentra en su versión 1.20190215 y se ha utilizado para crear bloques que den funcionalidad a los robots empleados en la plataforma.

3.3.1. Traductor de código

Blockly es para desarrolladores y las *aplicaciones Blockly* son pensadas para estudiantes. Desde la perspectiva de usuario, *Blockly* es una forma visual e intuitiva de crear código y, desde la de desarrollador, es una interfaz de usuario preparada para crear un lenguaje visual que emite código y se puede exportar a otros lenguajes de programación como *JavaScript*, *Python*, *PHP*, *Lua* o *Dart*.

Estos generadores de código aportan las herramientas para crear funciones, condicionales, bucles, etc. El principal problema de esto es que en ocasiones se requiere del uso de APIs de otras dependencias. *Blockly* aporta una solución; un generador de bloques personalizados que traduce al código que deseemos. Esto aporta mucha flexibilidad y da la funcionalidad deseada para el desarrollo de este proyecto.

3.3.2. Bloques personalizados

Como se ha explicado, *Blockly* dispone de una gran cantidad de bloques predefinidos; desde funciones matemáticas hasta estructuras en bucle. Sin embargo, para interactuar con una aplicación externa, se deben crear bloques personalizados para formar una API. Según la documentación ofrecida por *Google*³, la mejor forma de crear un bloque es buscar un bloque existente con una funcionalidad similar y modificarlo según se necesite. De otra manera, para generar un bloque personalizado, hay varios aspectos a tener en cuenta:

- Primero se define el bloque para determinar su aspecto gráfico y su comportamiento. Esto incluye el texto, color, forma o cómo conectarlos con otros bloques. La configuración de estos parámetros se puede realizar mediante *JSON* o *JavaScript*. El bloque mostrado en la figura 3.2 se puede configurar con los dos métodos de la siguiente manera:

```

1  {
2      "type": "block_test",
3      "message0": "%1 %2",
4      "args0": [
5          {
6              "type": "field_label_serializable",
7              "name": "NAME",
8              "text": "custom_block"
9          },
10         {
11             "type": "input_value",
12             "name": "NAME",
13             "check": "Number"
14         }
15     ],
16     "inputsInline": false,
17     "previousStatement": null,

```

³<https://developers.google.com/blockly/guides/create-custom-blocks/overview>

```

18     "nextStatement": null,
19     "colour": 120,
20     "tooltip": "This is a custom block",
21     "helpUrl": ""
22 }
```

Listing 3.3: Código en JSON para configurar un bloque personalizado

```

1 Blockly.Blocks['block_test'] = {
2   init: function() {
3     this.appendValueInput("NAME")
4       .setCheck("Number")
5       .appendField(new Blockly.FieldLabelSerializable("custom_block"), "NAME");
6     this.setInputsInline(false);
7     this.setPreviousStatement(true, null);
8     this.setNextStatement(true, null);
9     this.setColour(120);
10    this.setTooltip("This is a custom block");
11    this.setHelpUrl("");
12  }
13};
```

Listing 3.4: Código en *JavaScript* para configurar un bloque personalizado

Figura 3.2: Bloque personalizado

- Configurar la traducción del bloque a la instrucción deseada en los distintos lenguajes necesarios.
- Inicializar el bloque para que sea visible en el editor visual de código.

Aunque los parámetros del *JSON* son auto-descriptivos, es complicado generar un bloque desde cero. Es por ello que *Google* facilita una herramienta de creación de bloques, la cual se muestra en la figura 3.3. En ella se puede personalizar todos los aspectos del bloque: texto, color, variables de entrada, formas de conexión con otros bloques, etc. Además, da la opción de exportar la configuración del bloque en formato *JSON* o *JavaScript* y muestra una función con la configuración básica para obtener el código en *JavaScript* (y permite seleccionar el lenguaje deseado entre los admitidos por *Blockly*).

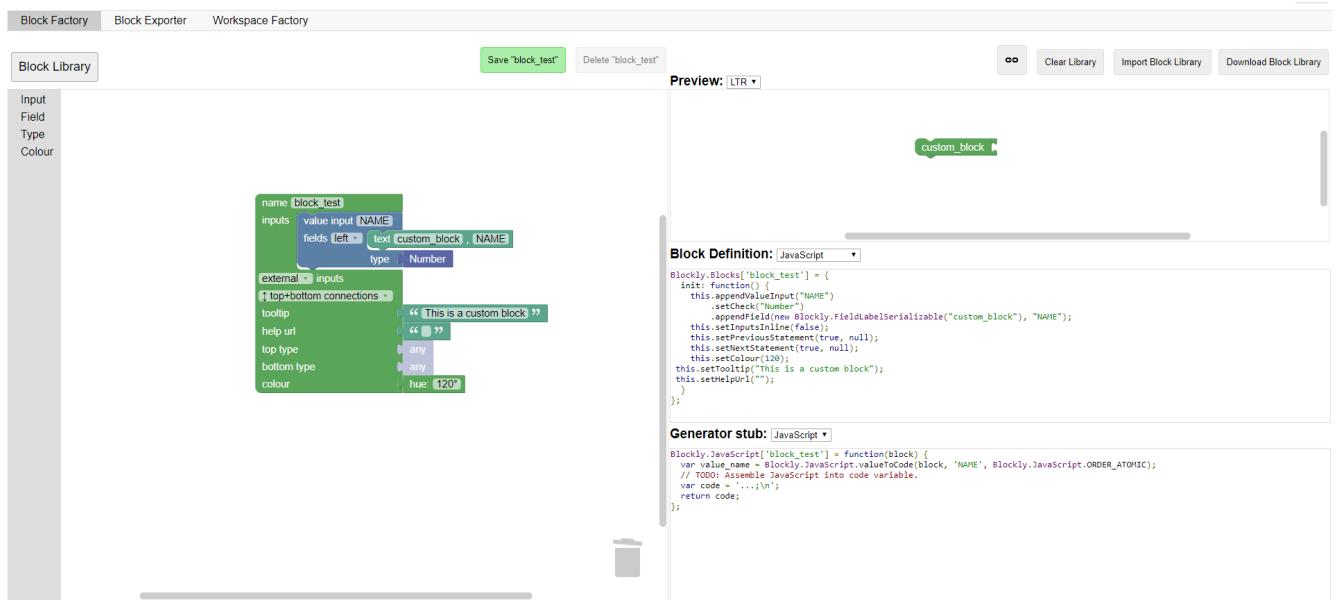


Figura 3.3: Herramienta para creación de bloques personalizados

3.4. Gestores de paquetes

Un gestor de paquetes es una herramienta para automatizar el proceso de instalación, actualización, configuración y eliminación de software. En este proyecto se han utilizado *NPM* y *Webpack* en sus versiones 3.5.2 y 4.41.2 respectivamente. Se han empleado para descargar los paquetes y dependencias para la correcta ejecución de la aplicación y para generar los *bundles* correspondientes.

3.4.1. NPM

*Node Package Manager*⁴ (*NPM*) es el sistema de gestión de dependencias por defecto para *Node.js*, que es un entorno de ejecución para *JavaScript* y permite, con la configuración de un fichero, descargar dependencias y paquetes necesarios para el correcto funcionamiento de la aplicación. Todo ello ha de estar definido en un fichero llamado *package.json* que debe ser escrito en *JSON*. Para usar *NPM* como instalador de paquetes únicamente hay que ejecutar “*npm install*” en el directorio en el que se encuentre el fichero *package.json*. Además, gracias a este gestor, podemos configurar pequeños *scripts* para ejecutar algún proceso como lanzar una

⁴<https://www.npmjs.com/>

aplicación después de la instalación o empaquetar la aplicación en *bundles* combinándolo con *Webpack*.

3.4.2. Webpack

*Webpack*⁵ es un sistema de *bundling* usado para empaquetar una aplicación web en su fase de producción. Se puede considerar una evolución de *Grunt*⁶ y *Gulp*⁷ porque permite automatizar procesos principales tales como transpilar y preprocesar código.

3.5. Blender

Blender es un programa libre dedicado al diseño y animación 3D. Mediante una interfaz gráfica permite diseñar objetos, personajes y escenas en tres dimensiones con muy diversas técnicas. Cada uno de los elementos creados pueden ser animados mediante *keyframing* o animación por fotogramas clave. En su origen *Blender* fue distribuido como una herramienta privada explotada por un estudio de animación, pero actualmente se encuentra bajo licencia *GPL*⁸.

La versión utilizada en este proyecto es la 2.79 y se ha empleado para la creación de modelos y escenarios para ser incluidos en el simulador *WebSim*. En la imagen 3.4 se puede ver la interfaz gráfica de *Blender*.

⁵<https://webpack.js.org/>

⁶<https://gruntjs.com/>

⁷<https://gulpjs.com/>

⁸<https://www.gnu.org/licenses/gpl-3.0.en.html>

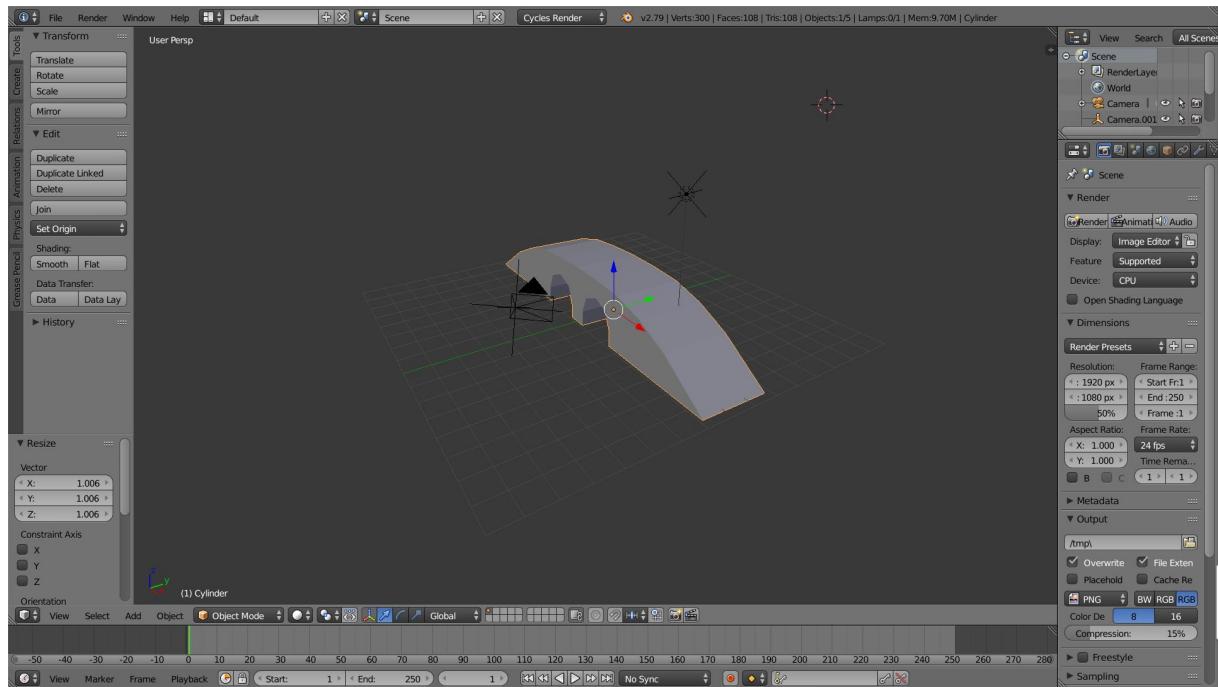


Figura 3.4: Interfaz gráfica de *Blender*

Entre las muchas características que ofrece *Blender*, las más útiles para el desarrollo del proyecto son:

- **Modelados:** es el proceso mediante el cual se crean objetos en un espacio 3D de manera digital. Están compuestos de líneas, puntos, cuadrados, triángulos, etc.
- **Iluminación:** existen varios tipos de luz que se pueden adaptar a la escena. Es posible aplicar a cada uno de los objetos para especificar la luz recibida por cada uno de ellos además de variar aspectos como la intensidad, el color o la posición en el escenario.
- **Tracking:** permite especificar el comportamiento y características de un determinado objeto en el escenario tridimensional.
- **Animaciones:** cada objeto se puede animar de manera independiente. Se realiza mediante una secuencia de fotogramas en la cual cada instante de tiempo es un fotograma en el cual se especifica cualquier tipo de característica (rotación, movimiento, cambio de color, etc).
- **Texturizado:** son un medio para agregar o eliminar detalles a la superficie de un determinado objeto. Proyectando imágenes sobre la superficie del mallado se pueden aplicar patrones para personalizar el aspecto de la textura.

Además permite exportar los modelos a los formatos más usados para *A-Frame*: *glTF* y *COLLADA*.

3.5.1. COLLADA

COLLaborative Design Activity (COLLADA) es un formato de archivo de intercambio para aplicaciones 3D interactivas. Se define como un esquema *XML* para intercambiar activos digitales entre varias aplicaciones de software de gráficos. Son definidos con la extensión *.dae* y pueden hacer referencia a archivos de imagen adicionales que actúan como texturas del objeto 3D.

3.5.2. glTF

GL Transmission Format (*glTF*) es una especificación basada en el estándar *JSON* para transmisión y carga eficiente de escenas y modelos 3D para aplicaciones. Este formato minimiza el tamaño de los ficheros y su tiempo de ejecución necesario para desempaquetar y usarlos. Permite animación de los objetos, que puede ser activada por *A-Frame* vía *HTML* o dinámicamente con *JavaScript*. En la imagen 3.5 se puede ver la figura añadida a un escenario de *A-Frame* vía *HTML*:

```
1 <a-asset-item id="tree" src="assets/models/tree.dae"></a-asset-item>
2 <a-entity id="a-tree" collada-model="#tree" rotation="0 0 0" position="2.75 0.01 -2.27">
```

Listing 3.5: Código para añadir un modelo 3D personalizado a A-Frame

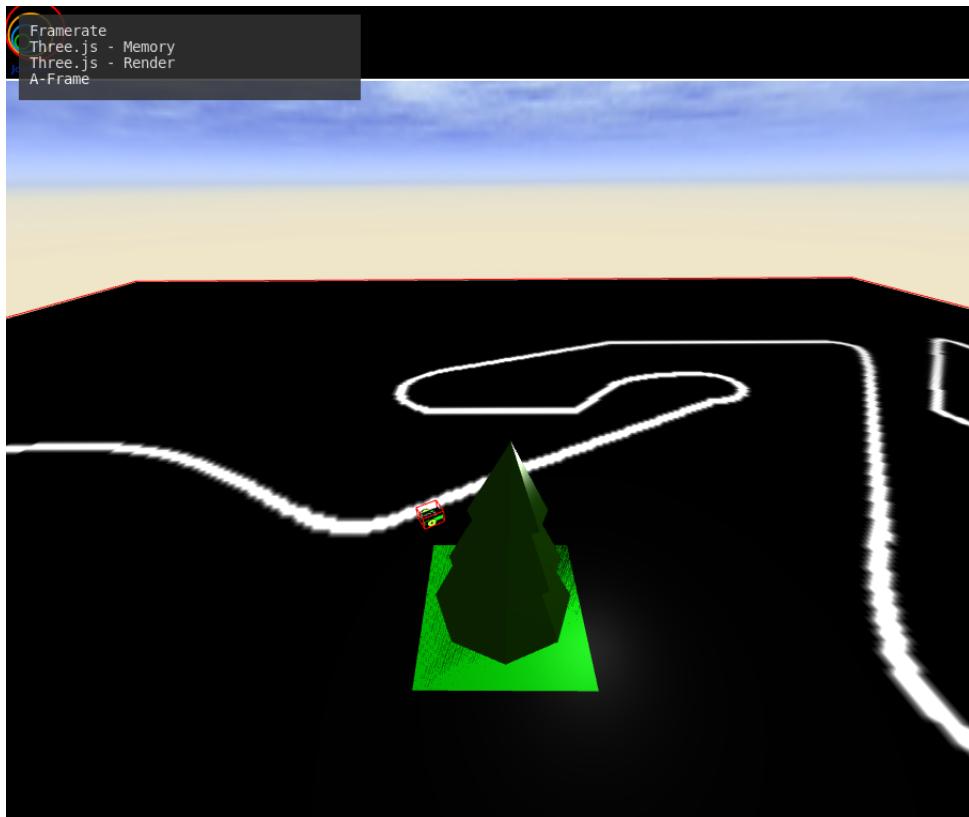


Figura 3.5: Modelo glTF añadido en un escenario de A-Frame

Además del *software* explicado en esta sección, existen multitud de herramientas para convertir a glTF desde formatos como *OBJ*, *COLLADA* o *PCD*⁹. De esta manera es posible buscar modelos 3D en librerías¹⁰ para editarlos e incluirlos en *Websim*.

3.6. Simulador WebSim

Websim es un simulador diseñado para enseñar conceptos básicos de tecnología e iniciar a niños en robótica y programación. La versión inicial de la que parte este proyecto está desarrollada por Álvaro Paniagua Tena¹¹ y dada su importancia en el desarrollo de este TFG, se describe con cierta profundidad.

⁹<https://github.com/KhronosGroup/glTF>

¹⁰<https://sketchfab.com>

¹¹https://github.com/RoboticsLabURJC/2018-tfg-alvaro_paniagua

3.6.1. Diseño

El simulador hace uso del entorno *A-Frame* y su diseño permite conectar un editor de texto o un editor de bloques para programar en *JavaScript* o *Blockly* y conectar este código con el robot simulado. También permite acoplar una aplicación externa al navegador a través de comunicaciones ICE. En la figura 3.6 se puede ver el diseño de *WebSim*.

Las principales funcionalidades del simulador son:

- Registrar los componentes principales para constituir un robot en *A-Frame*, los cuales son *followBody*, *spectatorComponent* e *intersectionHandler*. El primero se encarga de simular una cámara en el robot, el segundo maneja eventos de intersección de los láseres y el último permite anclar distintos elementos al robot simulado.
- Ofrece una interfaz en *JavaScript* para manejar el robot en el entorno simulado de *A-Frame* llamada *Hardware Abstraction Layer (HAL API)*. *Websim* se encarga de enviar instrucciones al robot de manera sencilla sin necesidad de comunicarse con el motor de *A-Frame*.
- No es necesario que el usuario instancie ningún tipo de variable de la clase que contiene al objeto robot porque el simulador lo ofrece de manera directa. De esta manera el usuario puede mandar directamente instrucciones al robot simulado con el objeto *myRobot* y los métodos existentes de la clase *RobotI*.
- Permite manejar la ejecución de la simulación del robot. Es decir, permite lanzar o pausar la ejecución del robot e incluso reiniciar su posición para no tener que recargar la página en caso de querer probar distintos códigos con la misma simulación. Además este control del entorno evita que la variable *myRobot* pierda el objeto instanciado porque el usuario cambie su valor.

Gracias a estas características, el simulador hace que los usuarios puedan programar de manera sencilla, ya que solo tienen que acceder a la información que ofrecen los sensores del robot y mandar órdenes sobre los actuadores del mismo. Es decir, solo se tienen que encargar de programar la lógica del robot para resolver los ejercicios propuestos que se explicarán en próximos capítulos.



Figura 3.6: Interfaz gráfica de *WebSim*

3.6.2. Robot en WebSim

Para tener una entidad en *A-Frame* que pueda ser programada por el usuario existe una clase en *JavaScript* llamada *RobotI*. El constructor de esta clase es un método obligatorio y es el primero que se llama cuando se instancia un objeto. Tiene un parámetro de entrada que es una cadena de caracteres con el identificador de la etiqueta *HTML* en la que se encuentra el robot simulado. Además de inicializar el robot, la clase contiene una serie de variables para su configuración:

- ***defaultDistanceDetection***: declara la distancia máxima en la que los sensores de ultrasónico detectan un objeto.
- ***defaultNumOfRays***: declara el número de rayos que se simulan para detectar objetos. Abarca un ángulo de 180 grados y, por lo tanto, cuanto mayor sea el número, más precisión habrá en la detección de objetos.
- ***robot***: crea la unión entre la clase y la entidad del robot simulado en *A-Frame*.
- ***initialPosition***: en la inicialización del robot, obtiene del *HTML* la posición del robot y la guarda en esta variable.

- ***initialRotation***: en la inicialización del robot, obtiene del *HTML* la rotación del robot y la guarda en esta variable.
- ***activeRays***: es una variable de tipo *boolean* y permite saber si los rayos del sensor de ultrasonidos están activos no.
- ***distanceArray***. Es un objeto en *JavaScript* con tres variables tipo *array* y almacena la distancia con los objetos detectada por los sensores de ultrasonido. Las variables están diferenciadas en derecha, centro e izquierda y permite conocer donde se encuentra la ubicación del objeto detectado.
- ***understoodColors***: variable que permite asociar un color como tipo *string* a sus valores de filtro en el espacio *RGB*. Esto permite hacer más simple la detección de los objetos para los usuarios ya que con el nombre del color se puede realizar el filtro sin necesidad de tener conocimientos sobre visión artificial.
- ***velocity***: variable que guarda la velocidad del robot en los distintos ejes (x,y,z). En el eje X se almacena la velocidad en el plano horizontal y en el eje Z la velocidad de giro. La velocidad del eje Y será útil en el desarrollo del proyecto ya que es la que otorga velocidad de elevación.

Además, el constructor llama a los métodos para inicializar los motores y sensores que tienen sus propios métodos para su manejo.

Para enviar al *robot* el código programado por el usuario se hace uso del módulo *brains* en *JavaScript*. En este se crea un “hilo” en el método *runBrain* que guarda el código escrito en el editor en un *array* y lo ejecuta periódicamente gracias a la función *timeOut* de *JavaScript*.

Este módulo también dispone de un método para parar el “cerebro” y reanudarlo después con un código distinto.

```

1 brains.runBrain = (robotID, code) =>
2   code = 'async function myAlgorithm() {\n' +code+'\n} \nmyAlgorithm();'
3   brains.threadsBrains.push({
4     "id": robotID,
5     "running": true,
6     "iteration": brains.createTimeoutBrain(code, Websim.robots.getHalAPI(robotID), robotID),
7     "codeRunning": code

```

```

8   });
9 }
```

Listing 3.6: Método *runBrain* para ejecutar el código del usuario

```

1 brains.resumeBrain = (robotID, code) =>{
2   code = 'async function myAlgorithm() {\n'+code+'\n}\nmyAlgorithm();';
3   var threadBrain = brains.threadsBrains.find((threadBrain)=> threadBrain.id == robotID);
4   threadBrain.iteration = brains.createTimeoutBrain(code, Websim.robots.getHalAPI(robotID),
5     robotID);
6   threadBrain.running = true;
7   threadBrain.codeRunning = code;
8 }
9 brains.stopBrain = (robotID) =>{
10   var threadBrain = brains.threadsBrains.find((threadBrain)=> threadBrain.id == robotID);
11   stopTimeoutRequested = true;
12   clearTimeout(threadBrain.iteration);
13   threadBrain.running = false;
14 }
```

Listing 3.7: Métodos *resumeBrain* y *stopBrain* para parar/reanudar el cerebro del robot

3.6.3. Drivers de sensores

El robot consta de sensores simulados con *A-Frame*. Los drivers permiten que el usuario, vía *JavaScript*, pueda acceder a estos y obtener su información. En este entorno disponemos de los siguientes sensores:

- **Ultrasonido:** permite detectar obstáculos en un radio de 180 grados en la parte frontal del robot. En *A-Frame* se simula gracias al atributo *raycaster* el cual permite conocer el punto de intersección entre un rayo emitido y un objeto. Para la simulación de este sensor se usan los componentes *followBody* (para anclar un *raycaster* al robot y que pueda seguir la posición del robot) e *intersectionHandler* (que crea un manejador para controlar el evento que lanza un *raycaster* cuando un objeto se interpone en el rayo que emite). Además, este componente es capaz de detectar el identificador del *raycaster* y la distancia a la que es detectado.
- **Cámara:** obtiene una imagen con lo que capta el robot de la escena. Para su simulación se ha creado el atributo *spectator* y permite mostrar la visión del robot en primera persona.

- **Infrarrojos:** dispositivo que está compuesto de un LED infrarrojo y un fototransistor siendo capaz de detectar si el objeto con el que choca la luz emitida por el LED es una superficie blanca (la luz rebota y llega hasta el fotoreceptor) o negra (la superficie absorbe toda la luz y no llega al fotoreceptor). Para simular este sensor se emplea una cámara y se recorta la imagen obtenida para quedarse con los píxeles de la parte inferior (con una dimensión de 5x150px). La función del *HAL API* que hace referencia a este sensor es *readIR(color)* siendo el parámetro pasado a la función un color en formato de *string* que toma como variable *undestandedColors* para obtener los valores del filtro correspondiente. De tal manera, después del filtrado se obtiene una imagen según la posición en la que se encuentre la línea a seguir y según donde esté el centro devuelve distintos valores: 0 si la línea se encuentra entre los píxeles 57 y 93 de la imagen, 1 entre los píxeles 0 y 57, 2 si la línea está entre 93 y 150 y 3 si no detecta la línea.
- **Odómetro:** obtiene la posición absoluta del robot durante la ejecución del código. Para la simulación de los sensores de odometría se ha hecho uso del sistema de coordenadas y rotación de *A-Frame* y se ha hecho el cálculo a través de su *API* en *JavaScript*.

En la tabla 3.1 se explican todas las funciones del *HAL API* que hacen referencia a los sensores.

Cuadro 3.1: Métodos (*HAL API*) de los sensores del robot.

Método	Descripción	Salida
.getDistance()	Devuelve la distancia entre el robot y la intersección del raycaster en el centro	number(metros)
.getDistances()	Devuelve la distancia entre el robot y la intersección con cada una de los raycaster	list number(metros)
.readIR(color)	Recorta la imagen, filtra y calcula el centro del objeto con el color pasado como argumento	number
.getRotation()	Retorna un objeto con la orientación del robot en los 3 ejes	{x:number, y:number, z:number}
.getPosition()	Obtiene la posición del robot en la escena	{x:number, y:number, z:number}
.getImage()	Devuelve la imagen de la cámara del robot	cv.Mat()
.getObjectColor(color)	Devuelve un objeto con la posición del elemento detectado por la cámara del color pasado por parámetro	{center:[x,y], area: int}
.getObjectColorRGB(valorBajo,valorAlto)	Devuelve un objeto con la posición del elemento detectado por la cámara con los valores pasados por parámetro	{center:[x,y], area: int}

3.6.4. Drivers de actuadores

La función de los actuadores es otorgar movimiento al cuerpo del robot simulado en *A-Frame*. Con los métodos creados, además, no es necesario mandarle órdenes constantemente, si no que es suficiente con mandar la instrucción una vez y el robot seguirá ejecutándola hasta que reciba una nueva.

Para que el robot siga las órdenes mandadas existen los siguientes métodos:

- **getV**: Método para conocer la velocidad lineal ordenada al robot.
- **getW**: Método que devuelve la velocidad angular comandada.
- **setV**: Método que permite ordenar velocidad lineal al robot.
- **setL**: Método para que el usuario comande la velocidad angular del robot.
- **move**: Método que sirve para ordenar tanto velocidad lineal como angular.

Después de llamar a cada uno de estos métodos, se guarda la velocidad pasada por parámetro en la variable interna llamada *velocity* y, para calcular la posición en el escenario simulado con *A-Frame* es necesaria la siguiente función:

```

1 updatePosition(rotation, velocity, robotPos) {
2     let x = velocity.x/10 * Math.cos(rotation.y * Math.PI/180);
3     let z = velocity.x/10 * Math.sin(-rotation.y * Math.PI/180);
4     robotPos.x += x;
5     robotPos.z += z;
6     return robotPos;
7 }
```

Listing 3.8: Función para actualizar la posición del robot en el escenario

En esta función se establece la nueva posición relativa para el objeto. En cada iteración se calcula y se establece nueva posición y rotación del robot. Además, gracias a la función de *JavaScript* llamada *setTimeout*, se establece un temporizador para actualizar la posición del robot llamando a la función *updatePosition* iterativamente cada cierto periodo de tiempo. Esto permite simplificar el código ya que hace que no es necesario mandar órdenes al robot constantemente.

En la tabla 3.2 se explican todas las funciones del *HAL API* que hacen referencia a los actuadores.

Cuadro 3.2: Métodos (*HAL API*) de los actuadores del robot.

Método	Descripción
.setV(integer)	Mueve hacia delante o atrás el robot.
.setW(integer)	Hace girar al robot.
.move(integer, integer)	Mueve el robot hacia delante/atrás y gira al mismo tiempo.
.getV()	Obtener la velocidad lineal configurada en el robot.
.getW()	Obtener la velocidad angular configurada en el robot.

Capítulo 4

Mejoras a WebSim

Una vez presentado el contexto, objetivos y herramientas empleadas, en este capítulo se detallan todas las mejoras realizadas del simulador *WebSim* y cómo se han llevado a cabo. Se va a explicar cómo se han integrado los *drones* ampliando los drivers existentes para incluir nueva funcionalidad, el modelo en 3D o los bloques de *Scratch* necesarios, los teleoperadores desarrollados y sus ficheros de configuración. También se detallan los nuevos ejercicios creados, tanto individuales como competitivos, que incorporan evaluadores automáticos para puntuar el funcionamiento de los *robots* programados.

4.1. Soporte a drones y nuevos robots

Uno de los objetivos principales del TFG era ampliar el soporte para otros robots y escenarios en *WebSim*. Se ha comenzado dando soporte a drones debido a las diferencias con el *robot PiBot* del que ya disponía soporte. Para ello hay que extender el *software* existente de la plataforma.

4.1.1. Driver del drone e interfaz de programación en *JavaScript*

Una de las principales diferencias entre el *drone* y el robot con ruedas *PiBot* es el movimiento vertical o en el eje Y (en la figura 4.1 se puede ver una representación del sistema de coordenadas de *A-Frame*) tanto para darle velocidad al *drone* como para actualizar la posición.

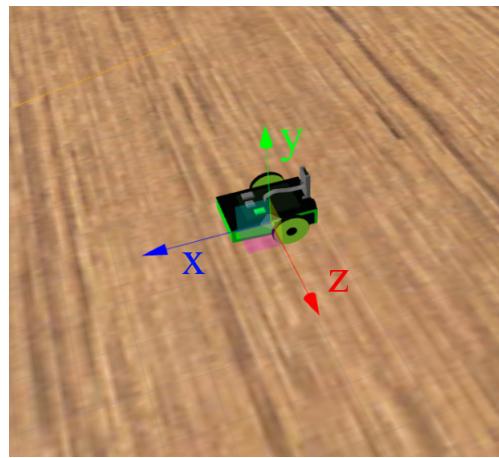


Figura 4.1: Sistema de ejes en A-Frame

Se han creado las siguientes funciones para ello:

- ***setL()***: Método que permite ordenar velocidad vertical al *robot*.

```

1   setL(l) {
2       this.velocity.y = l;
3   }
```

- ***getL()***: Método que devuelve la velocidad vertical del *robot*.

```

1 getL() {
2     return this.velocity.y;
3 }
```

- ***despegar()***: Método que imprime velocidad vertical al *robot* hasta alcanzar cierta altura.

```

1 despegar() {
2     this.velocity.y=3;
3     await sleep(0.5);
4     this.velocity.y=0;
5 }
```

- ***aterrizar()***: Método que imprime velocidad vertical negativa al *robot* hasta que alcance el suelo.

```

1 aterrizar() {
2     this.velocity.y=-3;
3     await sleep(0.4);
4     this.velocity.y=0;
5 }
```

Además, se han editado funciones y variables que ya existían para ampliar su funcionalidad:

- ***move()***: ahora acepta 3 parámetros y se incluye velocidad vertical como nuevo:

```

1      move(v, w, h) {
2          this.setV(v);
3          this.setW(w);
4          this.setL(h);
5      }

```

- ***updatePosition()***: se ha extendido para poder actualizar el eje Y para representarlo en la escena de *A-Frame*:

```

1 updatePosition(rotation, velocity, robotPos) {
2     let x = velocity.x/10 * Math.cos(rotation.y * Math.PI/180);
3     let z = velocity.x/10 * Math.sin(-rotation.y * Math.PI/180);
4     let y = (velocity.y/10);
5     robotPos.x += x;
6     robotPos.z += z;
7     robotPos.y += y;
8     return robotPos;
9 }

```

- ***this.velocity***: se ha ampliado la variable de la clase *robot* que guarda la velocidad:

```

1 this.velocity = {x:0, y:0, z:0, ax:0, ay:0, az:0};

```

En la tabla 4.1 se recopilan todas las funciones del *HAL API* que extienden la plataforma para dar soporte a *drones*.

Cuadro 4.1: Métodos (HAL API) de los actuadores implementados para el drone.

Método	Descripción
.setL(integer)	Mueve a cierta velocidad hacia arriba o hacia abajo el robot.
.getL()	Devuelve la velocidad vertical del robot.
.move(integer, integer, integer)	Mueve el robot a ciertas velocidades hacia delante/atrás, arriba/abajo y gira al mismo tiempo.
.despegar()	Comanda velocidad vertical al robot hasta que alcanza una determinada altura.
.aterrizar()	Comanda velocidad vertical negativa al robot hasta que alcanza el suelo.

Uno de los principales problemas encontrados es que el motor de físicas de *A-Frame* no simula correctamente la posición de robot al otorgarle velocidad vertical y hace que el robot “rebote” sobre el escenario. Esto es debido a que el escenario tiene un atributo llamado “gravedad” que se aplica cada pocos milisegundos y entra en conflicto con la función *updatePosition*. Se ha solucionado cambiando su valor haciendo que el escenario carezca de gravedad cuando se simula el *drone*.

4.1.2. Modelo 3D, apariencia

Para simular el robot en el entorno de *A-Frame* es necesario realizar un modelo tridimensional. Para ello se ha buscado un modelo disponible en un repositorio¹ y se ha retocado en *Blender* (figura 4.2) para que se ajuste a los requisitos del entorno. Las modificaciones que se han realizado al modelo son:

- Reducción de polinomios (modelo *low-poly*) para disminuir el peso del modelo y aliviar los tiempos de carga del escenario.
- Rotación del modelo para que encaje con la orientación que disponía el anterior robot. Es decir, que el robot tenga su parte frontal mirando hacia el eje X positivo para que al

¹<https://sketchfab.com/>

comandarle velocidad lineal se desplace hacia delante.

- El modelo 3D dispone de focos de luz, que son independientes de la luz que proporciona *A-Frame*. Se ha suprimido o desplazado para adaptarla al escenario.
- Elaborar una animación a las hélices para darle un aspecto más realista usando animaciones de *A-Frame*, que se activa vía *software* cuando el drone despegue del suelo².

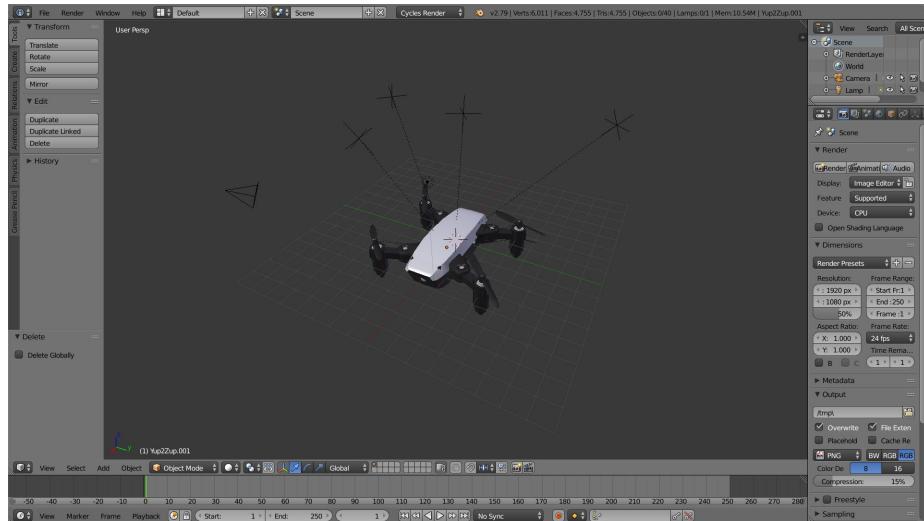


Figura 4.2: Drone en Blender

Blender genera un fichero *gltf* y da la posibilidad de contener en él un binario que incluye las animaciones o exportarlo en dos distintos. En este caso, se ha hecho como un único fichero y tiene un aspecto similar al formato *JSON*³.

El drone implementado en el entorno de *WebSim* se puede ver en la figura 4.3.

²<https://www.youtube.com/watch?v=XjQNhNCkOJA>

³https://github.com/RoboticsLabURJC/2019-tfg-ruben-alvarez/blob/master/upgrades/drones/drone_animation.gltf

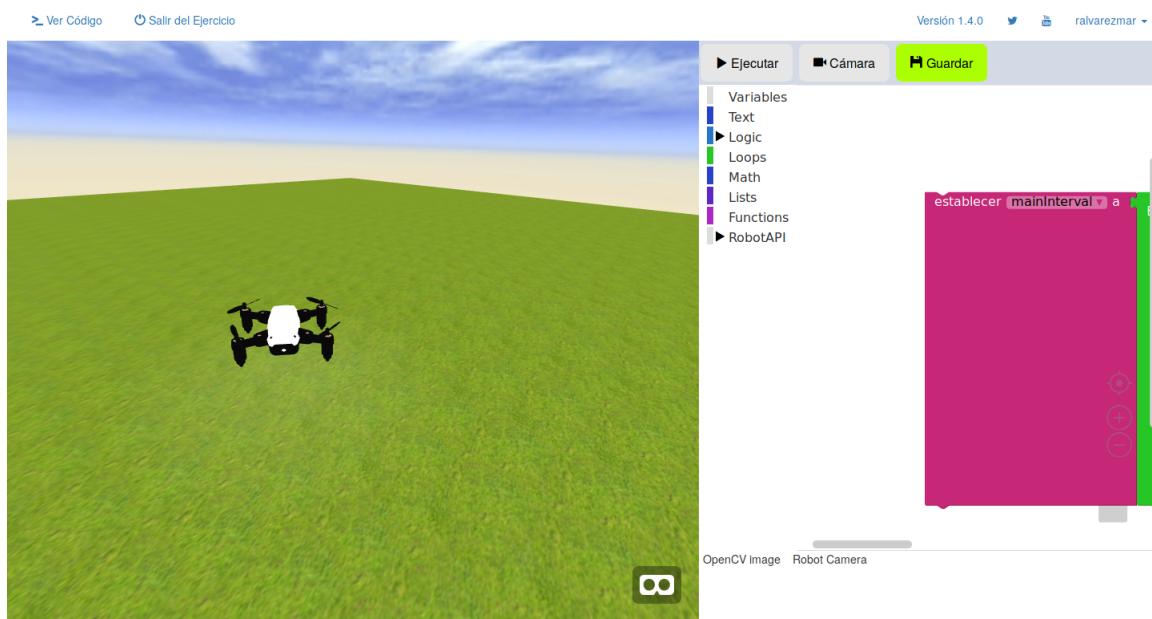


Figura 4.3: Escenario de WebSim con drone integrado

También se han creado modelos de *drones* de distintos colores para su disposición en ejercicios que requieran más de un *robot* en el escenario:

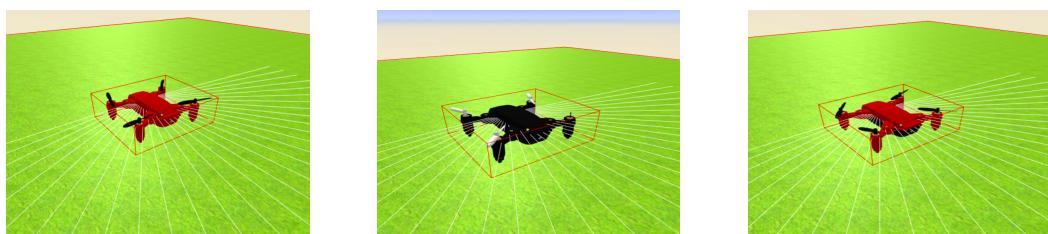


Figura 4.4: Modelos de *drone* de distintos colores

4.1.3. Bloques Scratch para programación gráfica del drone

Una vez implementado el código *JavaScript* para el soporte del drone, es necesario crear los bloques con *Blockly* para añadir sus funcionalidad en el editor de *Scratch*.

Para ello, se han creado 4 bloques con las funciones anteriormente explicadas:

- Velocidad ascenso: Comanda la velocidad ascendente del bloque que se le adjunte.



Figura 4.5: Bloque de velocidad de ascenso

- Velocidad descenso: Comanda al robot la velocidad descendente del bloque que se le adjunte.

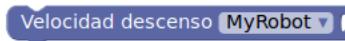


Figura 4.6: Bloque de velocidad de descenso

- Aterrizar: Comanda velocidad vertical negativa al *drone* hasta que alcance el suelo. Mantendrá esa posición hasta recibir una nueva orden.



Figura 4.7: Bloque de aterrizaje

- Despegar: Comanda velocidad vertical positiva al *drone* hasta que alcance cierta altitud.



Figura 4.8: Bloque de despegue

Como ejemplo ilustrativo se muestra en detalle la traducción del bloque despegar, que va incluida (junto a los otros bloques creados) en el directorio de bloques personalizados.

```

1 export default function initLandBlock() {
2   var landBlock = {
3     "type": "land",
4     "message0": "Despegar %1",
5     "args0": [
6       {
7         "type": "field_variable",
8         "name": "NAME",
9         "variable": "MyRobot"
10      }
11    ],
12    "previousStatement": null,
13    "nextStatement": "String",
14    "colour": "#%{BKY_MATH_HUE}",
15    "tooltip": "Aterriza el drone",
16    "helpUrl": "Aterriza el drone"
17  }
18  Blockly.Blocks['land'] = {

```

```
19     init: function() {
20         this.jsonInit(landBlock);
21     }
22 };
23 Blockly.JavaScript['land'] = function(block) {
24     var variable_name = Blockly.JavaScript.variableDB_.getName(block.getFieldValue('NAME'),
25         Blockly.Variables.NAME_TYPE);
26     var value_robotvar = Blockly.JavaScript.valueToCode(block, 'ROBOTVAR', Blockly.JavaScript.
27         ORDER_ATOMIC);
28     var code = variable_name + '.aterrizar(); \n';
29     return code;
30 };
31 Blockly.Python['land'] = function(block) {
32     var variable_name = Blockly.Python.variableDB_.getName(block.getFieldValue('NAME'),
33         Blockly.Variables.NAME_TYPE);
34     var value_robotvar = Blockly.Python.valueToCode(block, 'ROBOTVAR', Blockly.Python.
35         ORDER_ATOMIC);
36     var code = variable_name + '.aterrizar()\r\n';
37     return code;
38 };
39 }
```

Para que los bloques aparezcan en el editor de *Scratch* es necesario importarlos e inicializarlos en el fichero que lo configura. En la siguiente imagen se muestra el espacio de trabajo de *Scratch* con los nuevos bloques incorporados:

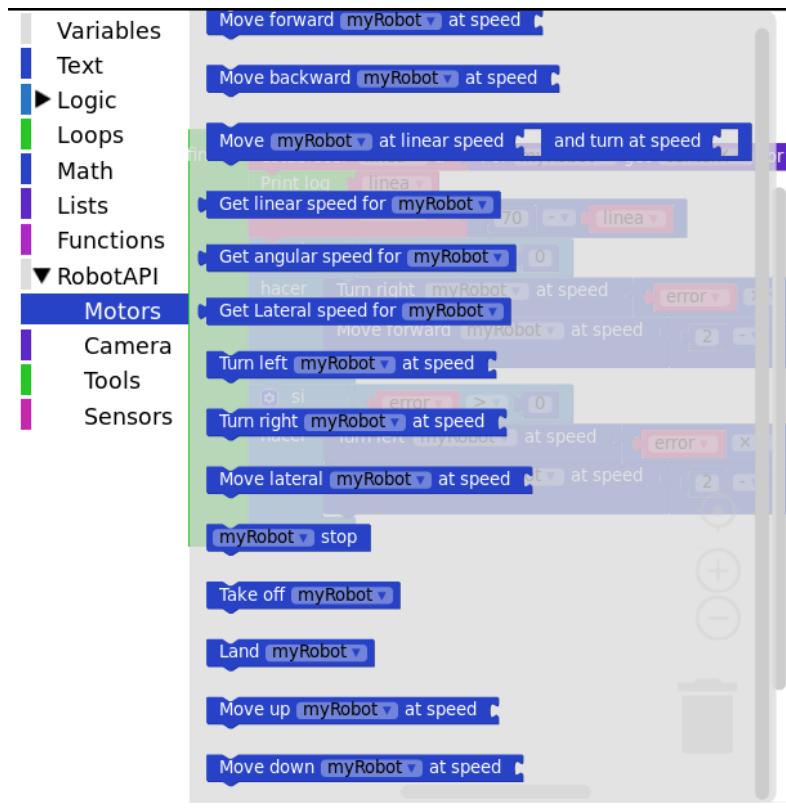


Figura 4.9: Espacio de trabajo de *Scratch* con los bloques del dron incorporados

4.1.4. Nuevos robots

Además del soporte a *drones*, se han incluido nuevos *robots* a *WebSim*:

- **Fórmula 1:** se han creado dos modelos distintos para incorporarlos a ejercicios que necesiten dos robots en la misma escena y que haya diferencias entre ambos para poder distinguirlos.

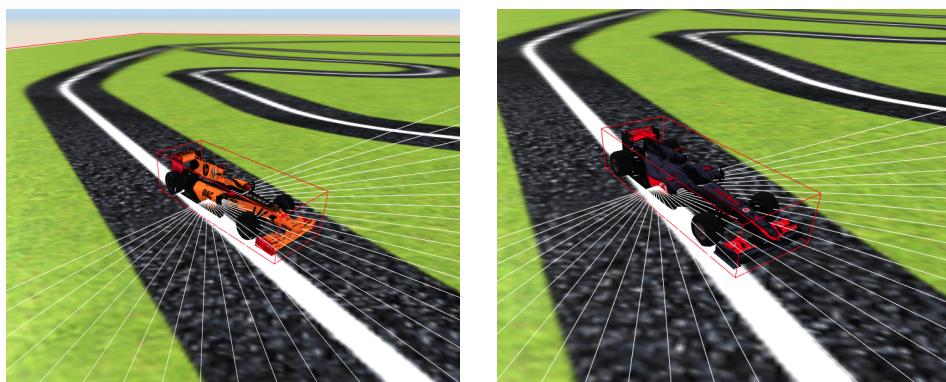


Figura 4.10: Modelos de coches Fórmula 1

- **mBot:** creado por la disponibilidad del *robot* real y su facilidad para añadir elementos al *robot* en la simulación.

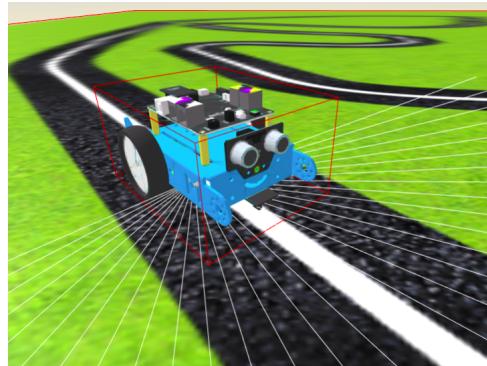


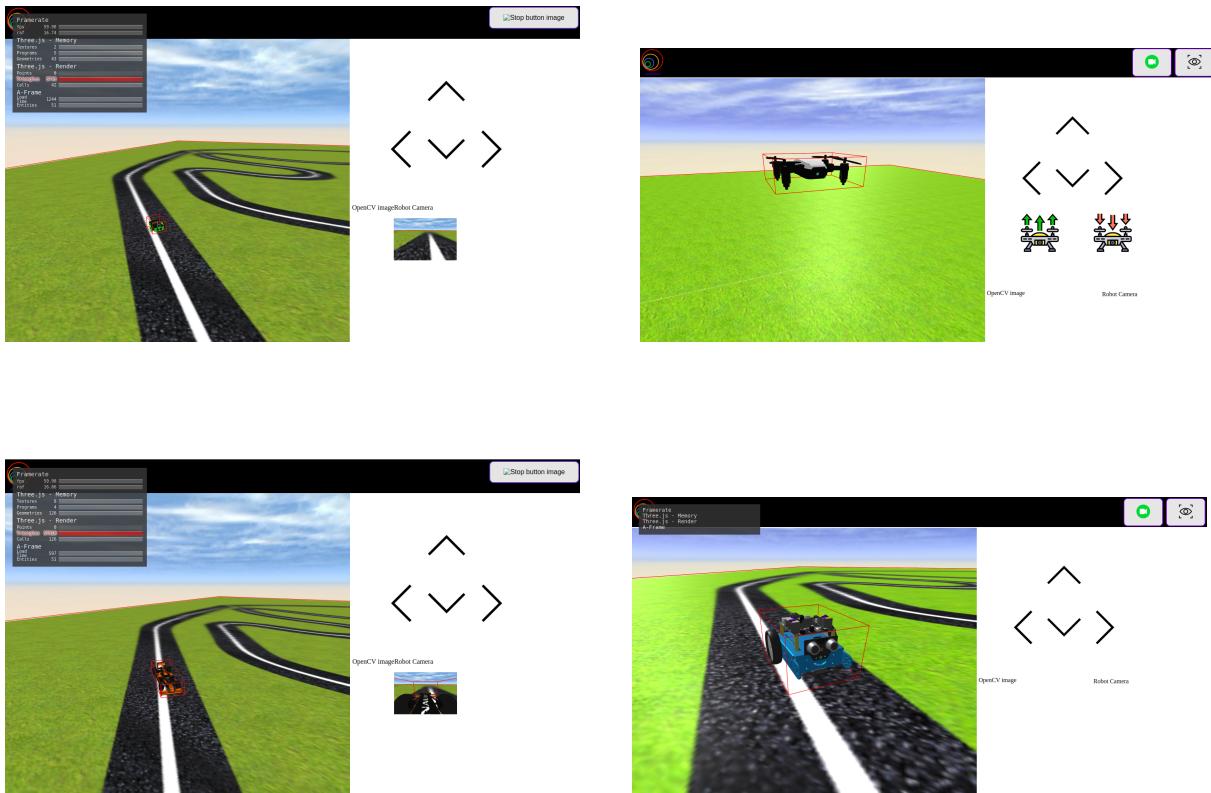
Figura 4.11: Modelo mBot

4.2. Teleoperadores en WebSim

Se han incorporado teleoperadores en *WebSim* para poder controlar los robots sin necesidad de programarlos. De esta manera es posible saber el estado y valor de sus sensores de manera sencilla ayudando a los desarrolladores a buscar fallos en drivers o incorporar nuevos escenarios. Su principal utilidad es ayudar a depurar la implementación de los nuevos *robots* desarrollados, el correcto funcionamiento de sus actuadores, sus sensores y sus interfaces de programación.

4.2.1. Interfaz gráfica

Se han creado teleoperadores para los modelos del *piBot*, *mBot*, *Formula 1* y del *drone*. Los 3 primeros tienen la misma interfaz gráfica y el último incorpora dos botones para controlar la velocidad vertical.



Siendo el código fuente *HTML* empleado para el teleoperador del drone el mostrado a continuación:

```

1 <div id="right-container">
2   <div class="buttons">
3     <div id="upArrow">
4       <button onclick class="buttonArrow" id="invisible"><img class="buttonArrow" src=
5         ".../assets/resources/speed.svg" /></button>
6       <button onclick class="buttonArrow" id="speed"></button>
8     </div>
9     <div id="bottomArrows">
10      <button onclick class="buttonArrow" id="left" ></button>
12      <button onclick class="buttonArrow" id="brake"></button>
14      <button onclick class="buttonArrow" id="right"></button><br>
16    </div>
17    <div id="takeoff">
18      <button onclick id="up"></button>
20      <button onclick id="down"></button>
22    </div>
23  </div>
24</div>
```

```

15     .svg" /></button></button>
16 </div>
</div>
```

Listing 4.1: Código HTML del teleoperador del drone

También se ha creado una página *web* única para acceder a todos ellos:

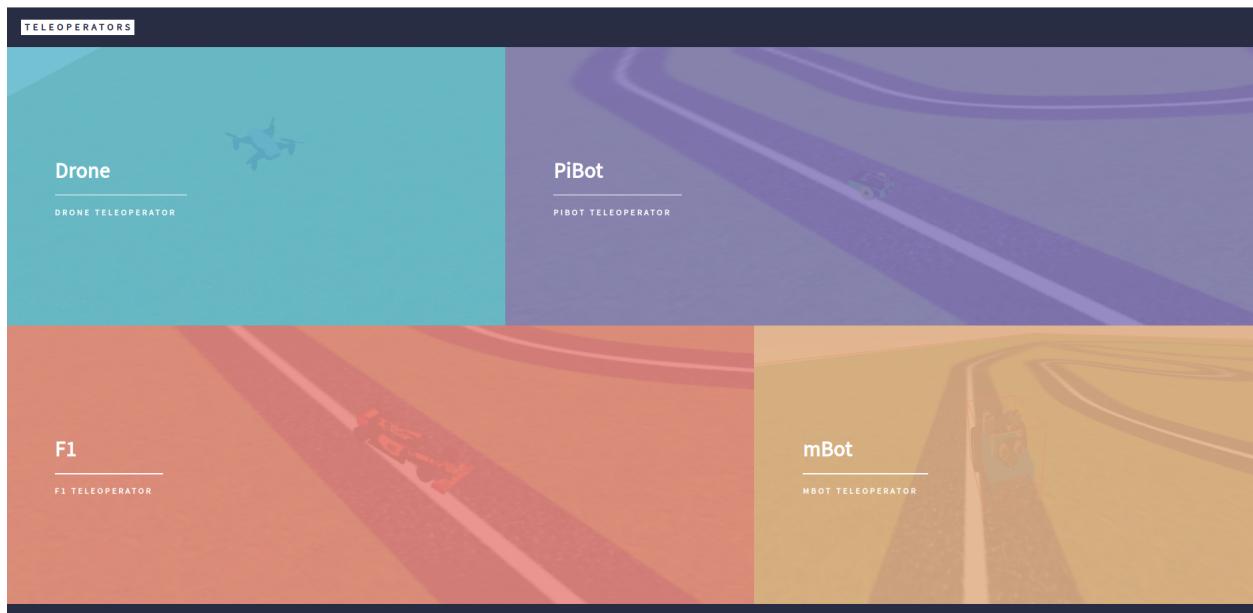


Figura 4.13: Interfaz que permite acceder a cada uno de los teleoperadores

```

1 <body>
2   <div id="wrapper">
3     <header id="header" class="alt">
4       <p class="logo"><strong>Teleoperators</strong></a>
5     </header>
6     <div id="main">
7       <section id="one" class="tiles">
8         <article>
9           <span class="image">
10             
11           </span>
12           <header class="major">
13             <h3><a href="drone.html" class="link">Drone</a></h3>
14             <p>Drone teleoperator </p>
15           </header>
16         </article>
17         <article>
18           <span class="image">
19             
20           </span>
```

```

21      <header class="major">
22          <h3><a href="pibot.html" class="link">PiBot</a></h3>
23          <p>PiBot teleoperator</p>
24      </header>
25  </article>
26  <article>
27      <span class="image">
28          
29      </span>
30      <header class="major">
31          <h3><a href="f1.html" class="link">F1</a></h3>
32          <p>F1 teleoperator</p>
33      </header>
34  </article>
35  <article>
36      <span class="image">
37          
38      </span>
39      <header class="major">
40          <h3><a href="mBot.html" class="link">mBot</a></h3>
41          <p>mBot teleoperator</p>
42      </header>
43  </article>
44  </section>
45      </div>
46  </div>
47 </body>
```

Listing 4.2: Código HTML de la interfaz para acceder a los teleoperadores

4.2.2. Arquitectura

Estos teleoperadores tienen una arquitectura *software* similar a las aplicaciones creadas de *Scratch* o *JavaScript*. En la figura 4.14 se muestra un esquema con el diseño de esta aplicación:

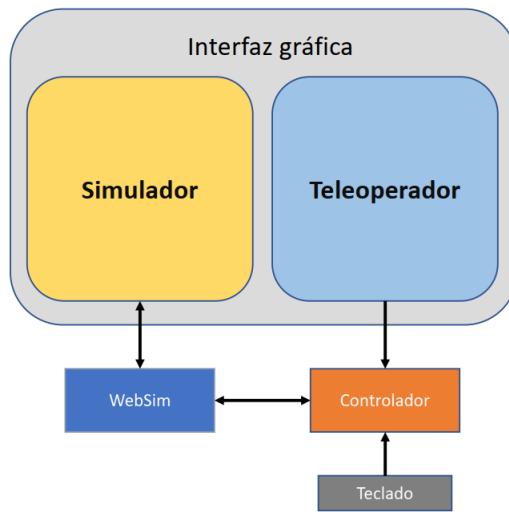


Figura 4.14: Arquitectura de la aplicación teleoperadores

Siendo el bloque teleoperador la interfaz creada en *HTML* (*listing 4.1*). Se envía un evento cuando se pulsa uno de los botones del teleoperador o del teclado, que mediante un controlador se comunica con *WebSim* tanto como para obtener las velocidades del robot como para enviarle las nuevas originadas por el evento, siempre usando los interfaces de programación para sensores y actuadores ofrecidos por el simulador.

Como ejemplo ilustrativo, a continuación se detalla el código del evento que inicializa el robot, los eventos que genera pulsar el teclado y los botones de la interfaz gráfica.

```

1  document.addEventListener('robot-loaded', (evt)=>{
2      localRobot = evt.detail;
3      console.log(localRobot);
4      document.addEventListener("keydown", keypressHandler, false);
5      document.addEventListener("keyup", keyupHandler, false);
6      $("#speed").click(()=>{
7          speed();
8      });
9      $("#brake").click(()=>{
10         brake();
11     });
12     $("#left").click(()=>{
13         left();
14     });
15     $("#right").click(()=>{
16         right();
17     });

```

```

18  $("">#up").click(()=>{
19      up();
20  });
21  $(">#down").click(()=>{
22      down();
23  });
24  $(">#takeOff").click(()=>{
25      takeOff();
26  });
27  $(">#land").click(()=>{
28      land();
29  });
30  });

```

Las funciones llamadas cuando se pulsa el teclado comandan una velocidad fija mientras se esté pulsando:

```

1 function keypressHandler(evt) {
2     if (evt.key == "i") {
3         localRobot.setV(0.9);
4     } else if (evt.key == "l") {
5         localRobot.setW(-0.2);
6     } else if (evt.key == "j") {
7         localRobot.setW(0.2);
8     } else if (evt.key == "k") {
9         localRobot.setV(-0.9);
10    } else if (evt.key == "u") {
11        localRobot.setL(0.9);
12    } else if (evt.key == "h") {
13        localRobot.setL(-0.9);
14    }
15 }

```

Cuando se pulsa alguno de los botones de la interfaz gráfica el comportamiento es diferente que pulsando el teclado:

- Flecha hacia arriba: se obtiene la velocidad lineal del robot y se incrementa.

```

1 function speed() {
2     let velocity = localRobot.getV()
3     localRobot.setV(velocity + 0.2);
4 }

```

- Flecha hacia abajo: se obtiene la velocidad lineal del robot y se reduce.

```

1 function brake() {
2     let velocity = localRobot.getV()

```

```

3     localRobot.setV(velocity - 0.2);
4 }
```

- Flechas laterales: se obtiene la velocidad angular del robot y, si es distinta de 0, se le comanda una velocidad en el sentido pulsado.

```

1 function right() {
2     let rotation = localRobot.getW()
3     if(rotation>0){
4         localRobot.setW(0);
5     }else{
6         localRobot.setW(-0.2);
7     }
8 }
9 function left() {
10    let rotation = localRobot.getW()
11    if(rotation<0){
12        localRobot.setW(0);
13    }else{
14        localRobot.setW(0.2);
15    }
16 }
```

- Botón de ascenso/descenso: se obtiene la velocidad vertical y se incrementa/disminuye.

```

1 function up() {
2     let velocity = localRobot.getL()
3     localRobot.setL(velocity + 0.2);
4 }
5 function down() {
6     let velocity = localRobot.getL()
7     localRobot.setL(velocity - 0.2);
8 }
```

4.2.3. WebSim y sus ficheros de configuración

Se ha mejorado el código de *WebSim* para que acepte sus ficheros de configuración en los que se especifica el escenario simulado, sus elementos, el robot elegido, distintos parámetros, etc. Esto da mucha flexibilidad al uso del simulador, qued dejando de tener estos elementos directamente en el código fuente.

Estos archivos se han creado en *JSON* y se ha programado un *script* (*listing 4.3*) para cargar cada fichero de configuración. Para ello se crea una variable en el *index.html* del editor (*listing*

4.4) con la ruta en la que esté ubicado y el *script* abre el fichero y recorre el *JSON* para dar al escenario los valores establecidos. Esta funcionalidad se ha probado y validado satisfactoriamente con los teleoperadores creados. De modo que además de mejorar el simulador para que los acepte, se han creado varios ficheros de configuración concretos como ejemplo con los diferentes *robots* soportados y varios escenarios distintos.

En los ficheros creados se pueden configurar aspectos como el modelo del robot cargado, su posición y rotación, la posición de la cámara a bordo del robot, la textura de cielo y de suelo que debe cargar o los elementos que queramos añadir en el escenario.

```

1  loadJSON(function(response) {
2      var config = JSON.parse(response);
3      var sceneEl = document.querySelector('a-scene');
4      var robot = sceneEl.querySelector('#a-pibot');
5      robot.setAttribute('gltf-model', config.robot.model);
6      robot.setAttribute('scale', config.robot.scale);
7      robot.setAttribute('position', config.robot.position);
8      robot.setAttribute('rotation', config.robot.rotation);
9      sceneEl.systems.physics.driver.world.gravity.y = config.gravity;
10     sceneEl.querySelector('#ground').setAttribute('src', config.ground);
11     sceneEl.querySelector('#sky').setAttribute('src', config.sky);
12     sceneEl.querySelector('#ground').setAttribute('src', config.ground);
13     sceneEl.querySelector('#secondaryCamera').setAttribute('position', config.secondaryCamera);
14     sceneEl.querySelector('#cameraRobot').setAttribute('position', config.cameraRobot);
15     if(config.objects.length>0){
16         setObjects(config.objects,sceneEl);
17     }
18 });
19 function setObjects(object,scene){
20     for (let i in object){
21         let keys = Object.keys(object[i]);
22         var element = document.createElement(object[i][keys[0]]);
23         for (let j = 1; j < keys.length; j++) {
24             let attribute = object[i][keys[j]];
25             element.setAttribute(keys[j],attribute);
26         }
27         scene.appendChild(element);
28     }
29 }
```

Listing 4.3: *script* que carga los ficheros de configuración

```
<script>var config_file = '../assets/config/config_follow_line.json';</script>
```

Listing 4.4: variable en *HTML* para indicar la ruta del fichero de configuración

```

1  {
2      "robot": {
3          "model": "../assets/models/drone.gltf",
4          "scale": "0.5 0.5 0.5",
5          "position": "12 0 25",
6          "rotation": "0 320 0"
7      },
8      "gravity": 0,
9      "ground": "../assets/textures/escenarioLiso.png",
10     "sky": "../assets/textures/sky.png",
11     "secondaryCamera": "0 0 0",
12     "cameraRobot": "0 0.03 -0.01",
13     "objects": [
14         {
15             "type": "a-sphere",
16             "position": "12 1 15",
17             "color": "#FF0000"
18         }
19     ]
20 }
```

En este ejemplo se configura el escenario para que cargue el modelo del *drone*, con el tamaño indicado en *size*, la posición y rotación que aparece en *position* y *rotation*. En el valor *gravity* se indica que el escenario no tenga gravedad, se carga la textura que posee el campo *ground* como suelo del mundo y, por último, la posición de las cámaras es la ubicada en *secondaryCamera* y *cameraRobot*. Además, en *objects* se pueden añadir todos los objetos deseados a la escena. En este ejemplo se añade al escenario una pelota de color rojo en la posición indicada.

4.3. Nuevos ejercicios individuales

Se han incorporado nuevos escenarios a *WebSim* que dan la posibilidad de realizar nuevos ejercicios y mejorar los ya disponibles. En esta sección se explicarán los nuevos ejercicios desarrollados con un solo *robot* en escena y sus soluciones en *Scratch*.

4.3.1. Sigue-líneas visión

Este ejercicio consiste en seguir una línea blanca sobre fondo negro haciendo uso de la cámara del *robot*, que recoge las imágenes y las filtra para poder seguirla.

Se ha mejorado el escenario cambiando la textura del suelo a una creada con la trazada del circuito de Interlagos de Fórmula 1. Se ha realizado con un programa de diseño gráfico

(*Photoshop*) y, debido a su peso computacional, se ha reducido posteriormente su tamaño para aliviar los tiempos de carga.

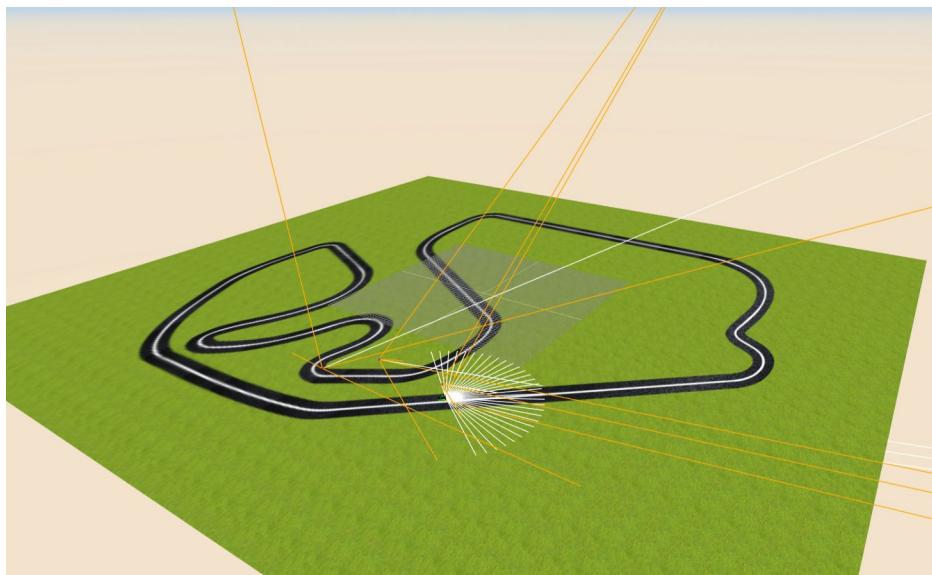


Figura 4.15: Escenario para el ejercicio *piBot sigue-líneas con cámara*

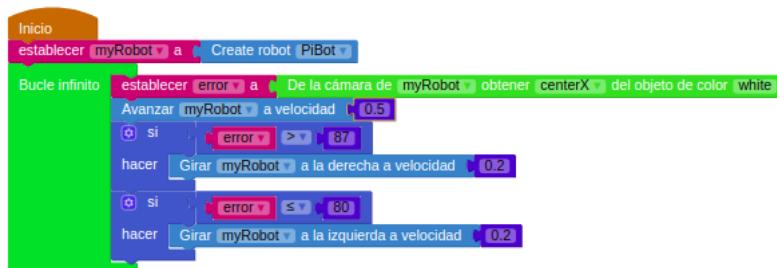


Figura 4.16: Solución en *Scratch* para el ejercicio *sigue-líneas visión*

En esta solución se comanda velocidad lineal al robot y utiliza el bloque que obtiene el centro del eje X de los objetos blancos. Según el valor obtenido, se gira el *robot* en un sentido u otro.

4.3.2. Sigue-líneas infrarrojos

Este ejercicio consiste en seguir una línea negra sobre fondo blanco haciendo uso de los sensores infrarrojos del *robot*.

Tiene un recorrido similar a sigue-líneas visión, pero con fondo blanco y recorrido negro para facilitar la implementación de código en el robot real y que no haya que realizar modificaciones. Para que funcione correctamente ha sido necesario añadir el color blanco a *undestandedColors* para realizar el filtro y poder pasar “white” como atributo a la función *getColorColor()*.

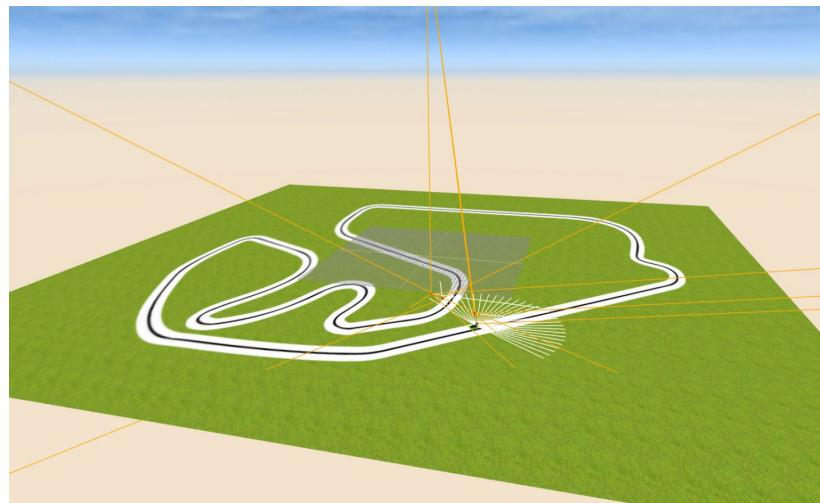


Figura 4.17: Escenario para el ejercicio para el *robot piBot* sigue-líneas infrarrojo

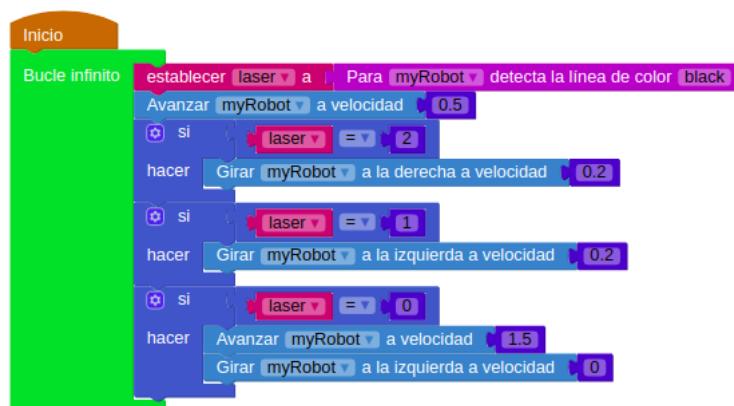


Figura 4.18: Solución en *Scratch* para el ejercicio sigue-líneas infrarrojos

En esta solución se comanda una velocidad lineal y se obtienen los valores de los sensores infrarrojos del *robot* y se comanda una velocidad angular en función de donde detecte la línea.

4.3.3. Choca-gira

En este ejercicio hay programar al *robot* para que avance recto mientras no haya obstáculos haciendo uso del sensor de ultra-sonidos. Si encuentra un obstáculo, tiene que detenerse, retroceder un poco, girar un ángulo aleatorio y reemprender la marcha.

Escenario creado en *Blender* con un aspecto similar a su análogo en el simulador *Gazebo*. Para ello se han adaptado la mayor parte de las estructuras que dispone el escenario original para su integración en *WebSim*.

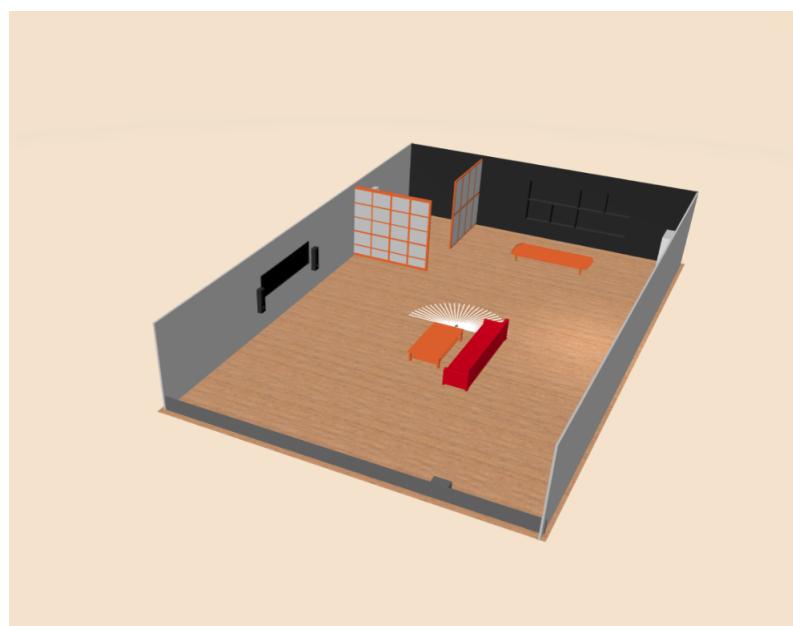


Figura 4.19: Escenario para el ejercicio choca-gira



Figura 4.20: Solución en *Scratch* para el ejercicio choca-gira

En esta solución se obtienen los valores del sensor de ultra-sonidos y se comanda una velocidad lineal. Si encuentra un obstáculo se gira a la derecha durante 0.5 segundos.

4.3.4. Sigue-pelota

Este ejercicio consiste en seguir una pelota en movimiento. Hay que emplear las imágenes obtenidas por el *robot* y programar la lógica de control que permita seguir la pelota.

Se ha realizado dos escenarios distintos, uno para *PiBot* y otro para *drone*. Ambos disponen de una pelota de color rojo, que debe ser seguida usando la cámara del *robot*, a la que se le ha dado movimiento a través de primitivas de *A-Frame*. En la figura 4.21 se puede ver una secuencia con la animación de la pelota y en el siguiente código se muestra el archivo de configuración de este ejercicio, que incluye esa animación de *A-Frame*:

```

1  {
2      "robot": {
3          "model": "../assets/models/drone_animation.gltf",
4          "scale": "0.5 0.5 0.5",
5          "position": "12 1 25",
6          "rotation": "0 50 0"
7      },
8      "gravity": 0,
9      "ground": "../assets/textures/escenarioLiso.png",
10     "sky": "../assets/textures/sky.png",
11     "secondaryCamera": "4 20 30",
12     "cameraRobot": "0 0.03 -0.01",
13     "objects": [
14         {
15             "type": "a-sphere",
16             "id": "redBall",
17             "position": "4 15 20",
18             "color": "#FF0000",
19             "radius": "1.5",
20             "animation": "property: position; from: 4 15 20 ;to: 0 15 -20; dir: alternate; dur: 10000
21                 ; loop: true",
22             "animation__2": "property: position; from: 0 15 -20 ;to: 0 2 -20 ; delay: 10000; dir:
23                 alternate; dur: 10000; loop: true",
24             "animation__3": "property: position; from: 0 2 -20 ;to: 4 2 20 ; delay: 20000; dir:
25                 alternate; dur: 10000; loop: true",
26             "animation__4": "property: position; from: 4 2 20 ;to: 4 15 20; delay: 30000; dir:
27                 alternate; dur: 10000; loop: true",
28             "animation__5": "property: position; from: 4 15 20 ;to: -10 15 10; delay: 40000; dir:
29                 alternate; dur: 10000; loop: true",
30             "animation__6": "property: position; from: -10 15 10 ;to: 20 8 -30; delay: 50000; dir:
31                 alternate; dur: 10000; loop: true"
32         }
33     }
34 }
```

Haciendo especial mención al campo *objects*, en el que se crea una pelota roja con la ani-

mación indicada en todos los campos *animation* y genera el siguiente elemento en *HTML*:

```

1 <a-sphere id="redBall" position="12 1 25" color="#FF0000" radius="1.5"
2 animation="property: position; from: 4 15 20 ;to: 0 15 -20; dir: alternate; dur: 10000; loop:
   true"
3 animation__2="property: position; from: 0 15 -20 ;to: 0 2 -20 ; delay: 10000; dir: alternate;
   dur: 10000; loop: true"
4 animation__3="property: position; from: 0 2 -20 ;to: 4 2 20 ; delay: 20000; dir: alternate;
   dur: 10000; loop: true"
5 animation__4="property: position; from: 4 2 20 ;to: 4 15 20; delay: 30000; dir: alternate; dur
   : 10000; loop: true"
6 animation__5= "property: position; from: 4 15 20 ;to: -10 15 10; delay: 40000; dir: alternate;
   dur: 10000; loop: true"
7 animation__6="property: position; from: -10 15 10 ;to: 20 8 -30; delay: 50000; dir: alternate;
   dur: 10000; loop: true">
8 </a-sphere>
```

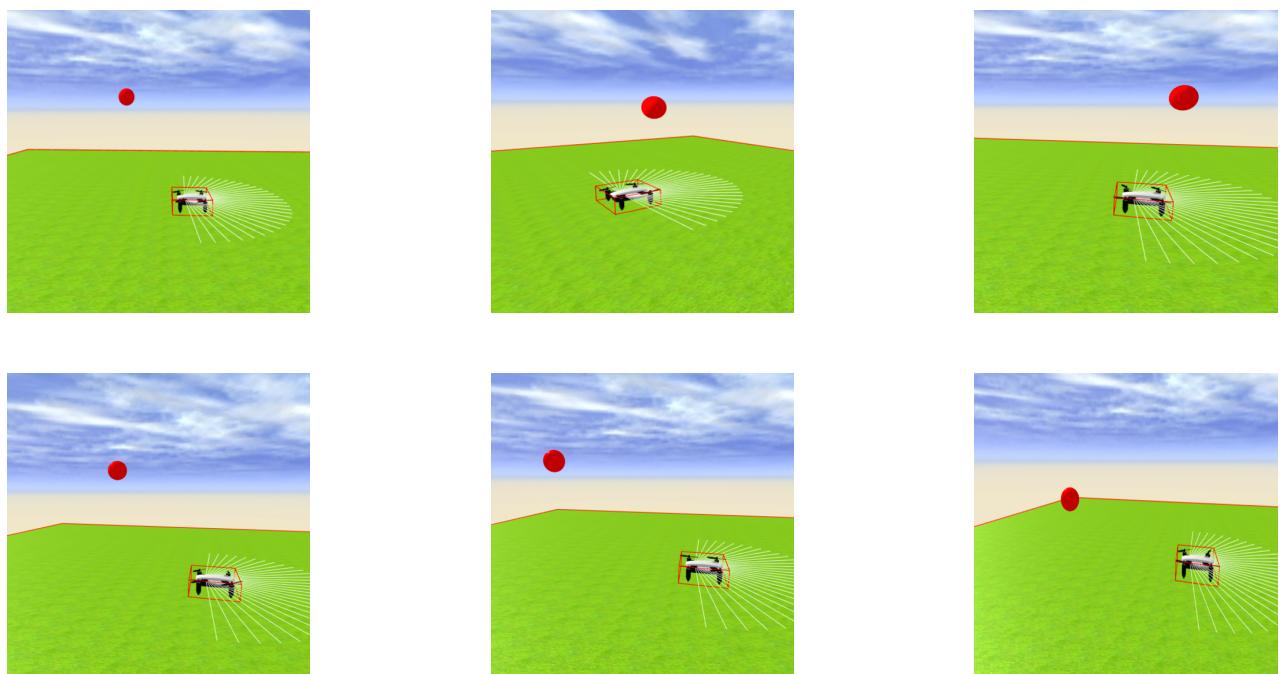


Figura 4.21: Secuencia de la animación de una pelota para el ejercicio *drone sigue-pelota*



Figura 4.22: Solución en *Scratch* para el ejercicio sigue pelota drone

La solución de este ejercicio se ha realizado obteniendo toda la información que aporta la cámara, tanto la posición del objeto en el eje X e Y como su área. Según los valores obtenidos se comandan distintas velocidades angulares y lineales.

4.3.5. Atraviesa-bosque

Ejercicio basado en atravesar un pasillo con diversos objetos que hay que esquivar. El sensor necesario es el infrarrojos para detectar en qué posición se encuentra cada uno de los obstáculos. El escenario y los obstáculos se han creado con primitivas de *A-Frame*.

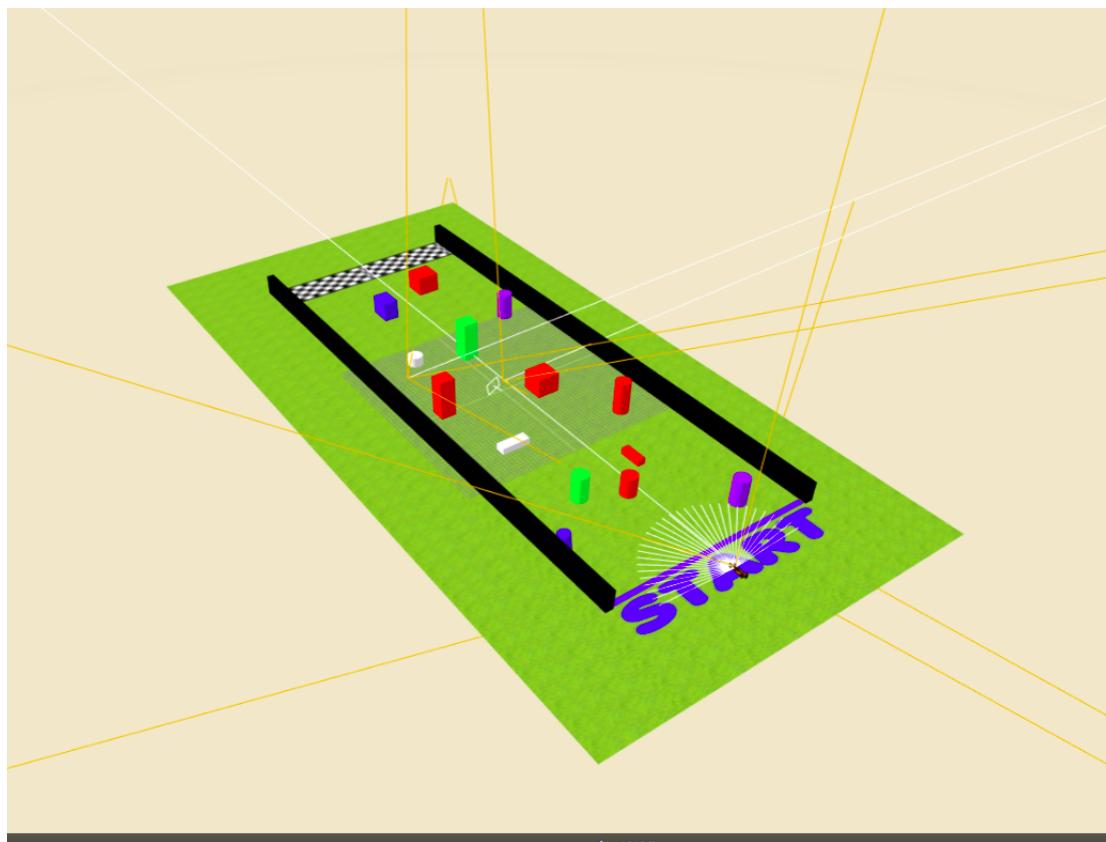


Figura 4.23: Escenario para el ejercicio atravesía bosque

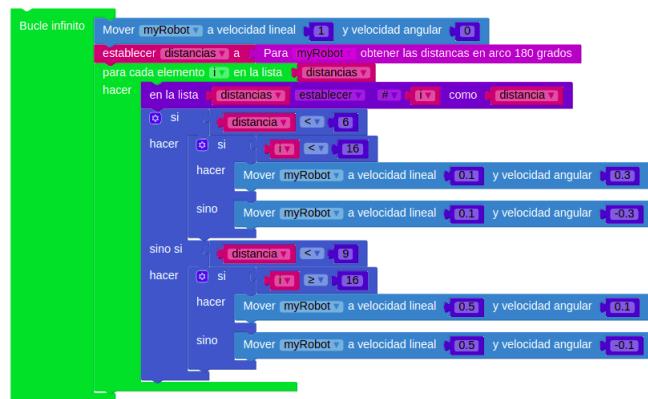


Figura 4.24: Solución en Scratch para el ejercicio atravesía bosque

En esta solución se obtienen todos los valores que devuelve el sensor de ultra-sonidos y, según donde detecte el obstáculo, se gira en un sentido u otro.

4.3.6. Cuadrado con drone

Este ejercicio consiste en comandar velocidades al *drone* para dibujar un cuadrado con el movimiento del *robot*.

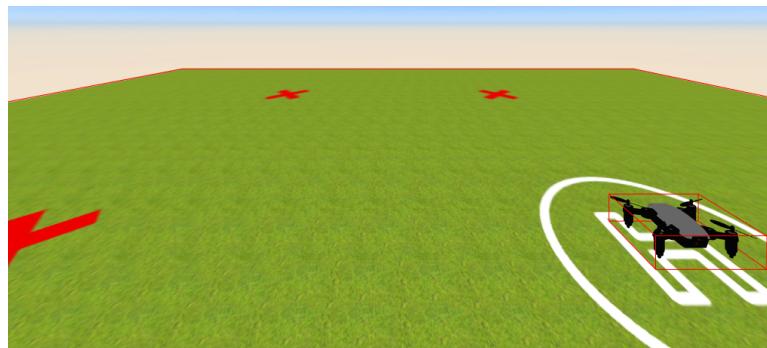


Figura 4.25: Escenario de WebSim para el ejercicio drone cuadrado



Figura 4.26: Solución en *Scratch* para el ejercicio cuadrado drone

Su solución consiste en avanzar los metros necesarios para llegar a la cruz y, cuando se ha completado, girar 90 grados aproximadamente para así “dibujar” un cuadrado repitiendo este proceso.

4.4. Ejercicios competitivos

Uno de los objetivos de este proyecto era añadir ejercicios competitivos a *Kibotics* sobre el simulador *Websim*. Se hace especial mención a ellos debido a que son completamente diferentes al resto de los ya creados. Este tipo de ejercicios aumenta el valor de la plataforma ya que da la posibilidad de programar dos robots y ponerlos a funcionar en el mismo escenario

simultáneamente, pudiendo entender la programación como un juego en el que se premia al que aporte la mejor solución.

4.4.1. Arquitectura de cómputo

En este tipo de ejercicios hay dos robots en una misma escena y cada uno de ellos se puede programar con un código distinto. Para su implementación e integración en *Kibotics* se ha extendido el módulo *brains* y se han incorporado dos aplicaciones más a *WebSim*: ejercicios competitivos en *Scratch* y ejercicios competitivos en *JavaScript*.

Se ha comenzado creando la aplicación llamada *competitive-JavaScript* debido a la facilidad para probar código y hacer pruebas en el entorno. Para ello se ha cambiado la interfaz del editor de código fuente, añadiendo botones para cada uno de los robots (figura 4.27) y añadido funcionalidad a cada botón para guardar el código de cada robot o mostrar el código en caso de tener uno guardado.

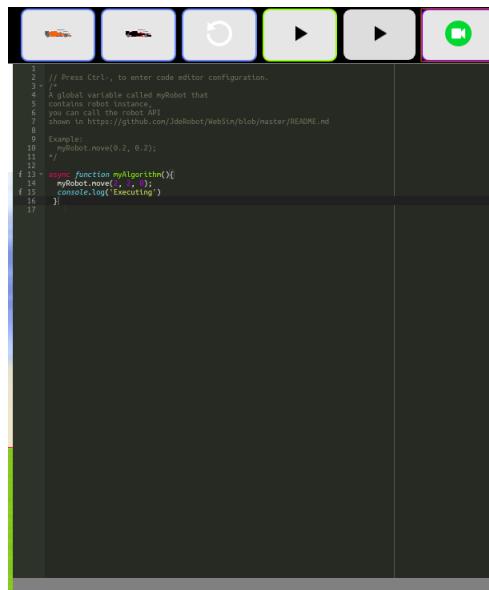


Figura 4.27: Editor de *JavaScript* para ejercicios competitivos

Cada uno de los botones tiene la funcionalidad de guardar el código escrito en el editor y, si se pulsa el botón del robot que no se está editando, se guarda el código y se carga el del otro robot en caso de que ya haya uno guardado. Si no hay ninguno se carga un editor vacío.

```

1 var editFirst = true;
2 var editSecond = false;

```

```

3 var codeFirst = null;
4 var codeSecond = null;
5 $('#firstRobot').click(()=>{
6     if(editFirst){
7         codeFirst = editor.getCode();
8     }
9     if(editSecond) {
10        codeSecond = editor.getCode();
11        editSecond=false;
12        if(codeFirst==null) {
13            editor.insertCode("",editor);
14        }else{
15            editor.insertCode(codeFirst,editor);
16        }
17    }
18    editFirst= true;
19 });

```

Cuando se pulsa el botón de ejecutar código, se ejecuta el método *runBrain* del módulo *brains* obteniendo previamente el código del robot que se esté editando.

```

1 $("#runbtn").click(()=>{
2     if (editFirst) {
3         codeFirst = editor.getCode();
4     } else {
5         codeSecond = editor.getCode();
6     }
7     if (brains.threadExists(editorRobot1)){
8         if (brains.isThreadRunning(editorRobot1)){
9             brains.stopBrain(editorRobot1);
10            brains.stopBrain(editorRobot2);
11        }else{
12            brains.resumeBrain(editorRobot1,codeFirst);
13            brains.resumeBrain(editorRobot2,codeSecond);
14        }
15    }else{
16        brains.runBrain(editorRobot1,codeFirst);
17        brains.runBrain(editorRobot2,codeSecond);
18    }
19 });

```

La aplicación *web competitive-Scratch* se ha realizado de manera similar, con la diferencia de que en este caso es necesario guardar el código de los bloques en *XML* y su traducción en *JavaScript*. Para realizarlo de forma limpia se ha creado un objeto que contiene un *boolean* y dos cadenas de texto (*listing 4.5*). En el primero indica el código de qué *robot* se está editando,

en una cadena de texto se guarda el código *XML* y en la otra su traducción en *JavaScript*. Se puede ver la interfaz de este editor en la figura 4.28.

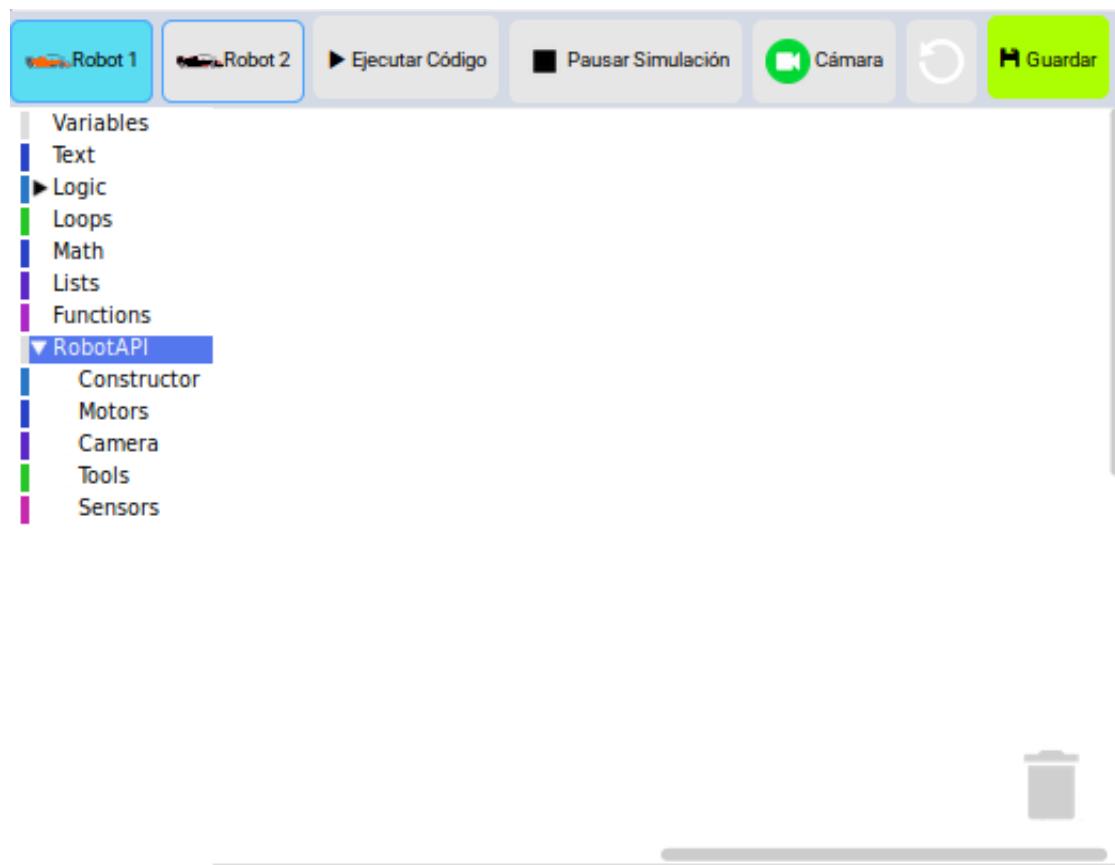


Figura 4.28: Editor de *Scratch* para ejercicios competitivos

```

1 var codeFirst = {
2   js:"",
3   xml:null,
4   edit:true
5 };
6 var codeSecond = {
7   js:"",
8   xml: null,
9   edit: false
10};
11 $('#firstRobot').click(()=>{
12   if(codeFirst.edit){
13     codeFirst.xml = editor.storeCode(editor.ui);
14     editor.ui = editor.injectCode(editor.ui, codeFirst.xml);
15   }
16   if(codeSecond.edit){
17     codeSecond.xml = editor.storeCode(editor.ui);

```

```

18 codeSecond.edit = false;
19 if(codeFirst.xml == null){
20     editor.ui = editor.injectCode(editor.ui, '<xml></xml>');
21 } else{
22     editor.ui = editor.injectCode(editor.ui, codeFirst.xml);
23 }
24 }
25 codeFirst.edit = true;

```

Listing 4.5: código *JavaScript* para guardar código de un robot

```

1 $( "#runbtn" ).click(()=>{
2     if (codeFirst.edit) {
3         codeFirst.xml = editor.storeCode(editor.ui);
4         editor.ui = editor.injectCode(editor.ui,codeSecond.xml);
5         codeSecond.js = editor.getCode();
6         editor.ui = editor.injectCode(editor.ui,codeFirst.xml);
7         codeFirst.js = editor.getCode();
8     } else {
9         codeSecond.xml = editor.storeCode(editor.ui);
10        editor.ui = editor.injectCode(editor.ui,codeFirst.xml);
11        codeFirst.js = editor.getCode();
12        editor.ui = editor.injectCode(editor.ui,codeSecond.xml);
13        codeSecond.js = editor.getCode();
14    }
15    if (brains.threadExists(editorRobot1)){
16        if (brains.isThreadRunning(editorRobot1)){
17            brains.stopBrain(editorRobot1);
18            brains.stopBrain(editorRobot2);
19        }else{
20            brains.resumeBrain(editorRobot1,codeFirst.js);
21            brains.resumeBrain(editorRobot2,codeSecond.js);
22        }
23    }else{
24        brains.runBrain(editorRobot1,codeFirst.js);
25        brains.runBrain(editorRobot2,codeSecond.js);
26    }
27 });

```

Listing 4.6: código *JavaScript* para ejecutar código de los robots y guardar el que se está editando

Para puntuar el comportamiento de los robots de manera justa se han incluido en estos ejercicios *evaluadores automáticos*. Van a tener diferentes comportamientos en cada ejercicio, por lo que se han desarrollado de tal forma que se pueda cargar cargar un evaluador distinto para cada uno o, incluso, no cargar ninguno.

Para su implementación se ha creado el módulo *evaluators*, que es similar a *brains*. Tiene un método *runEvaluator*, que acepta como parámetro un *array* con los identificadores de los *robots* a los que el código del evaluador debe conectarse para poder medir la calidad y el archivo del evaluador deseado. Este fichero se recoge como variable en el *index.html* (listing 4.4.1) del editor correspondiente de forma similar a los archivos de configuración:

```
1 <script>var config_evaluator = "evaluator_follow_line.js";</script>
```

Para llamar a *runEvaluator* se comprueba que se haya pasado un fichero en el *index.html* en el código que inicializa el editor correspondiente:

```
1 if(typeof config_evaluator!="undefined") {
2   evaluators.runEvaluator([editorRobot1,editorRobot2],config_evaluator);
3 }
```

En el método *runEvaluator* se realiza un *require* (que es la forma de importar módulos en *JavaScript* de manera dinámica) de ese fichero, se crea la interfaz gráfica en el método *evaluator.createInterface()* y se crea un objeto en el array de *brains* que se ejecuta cada 400 milisegundos por medio del método *evaluators.createTimeoutEvaluator*, que se apoya en la función *setTimeout* de *JavaScript*.

```
1 evaluators.runEvaluator = (arrayRobots,config_file)=>{
2   evaluator = require("../assets/evaluators/"+config_file);
3   evaluator.createInterface();
4   brains.threadsBrains.push({
5     "id": "evaluator",
6     "running": true,
7     "iteration": evaluators.createTimeoutEvaluator(arrayRobots,"evaluator"),
8     "codeRunning": ""
9   });
10 }
11 evaluators.createTimeoutEvaluator = (arrayRobots,id)=>{
12   stopTimeoutRequested = false;
13   let brainIteration = setTimeout(async function iteration(){
14     evaluator.setEvaluator(arrayRobots);
15     if (!stopTimeoutRequested) {
16       var t = setTimeout(iteration, 400);
17       var threadBrain = brains.threadsBrains.find((threadBrain)=> threadBrain.id == id);
18       threadBrain.iteration = t;
19     }
20   }, 400);
21   return brainIteration;
```

Listing 4.7: Funciones que crean el objeto para ejecutar el evaluador periódicamente

4.4.2. Atraviesa-bosque competitivo

Este ejercicio es similar al escenario con un solo robot, pero en este caso se han creado dos pasillos en lugar de uno⁴. Se han añadido distintos objetos y elementos de *A-Frame* en la misma ubicación para los dos *robots* para que el recorrido sea justo.

Para su evaluador se crea una barra de progreso para cada robot y un cronómetro. Cuando se empiezan a mover los robots la barra de progreso empieza a completarse y el cronómetro se inicia, para comprobar el porcentaje completado se obtiene la posición del robot y la compara con el punto de llegada.

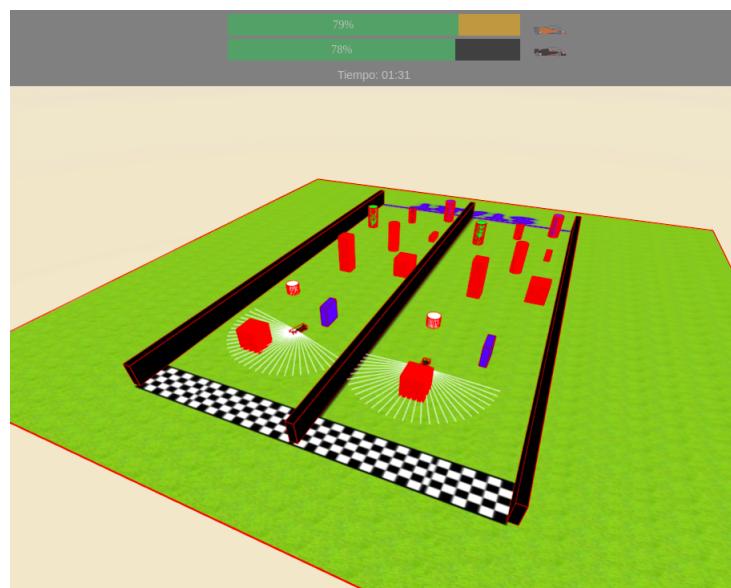


Figura 4.29: Escenario y evaluador para el ejercicio atravesia-bosque

Las funciones necesarias para el evaluador se definen a continuación:

- Una dedicada a crear la interfaz que establece los elementos necesarios para añadir las barras de progreso, iconos, tiempo y sus atributos.

```
1 evaluator.createInterface= ()=>{
2   var node = document.createElement("div");
```

⁴https://www.youtube.com/watch?v=v_aMvNPtnCG

```

3   node.setAttribute("class","evaluator");
4   var img1 = document.createElement("img");
5   img1.setAttribute("class","carMarker");
6   img1.setAttribute("src","../assets/resources/car1.svg")
7   node.appendChild(img1);
8   var node2 = document.createElement("div");
9   node2.setAttribute("id","car1Progress");
10  var node3 = document.createElement("div");
11  node3.setAttribute("id","a-car1bar");
12  node3.innerHTML = "0 %";
13  node2.appendChild(node3);
14  node.appendChild(node2);
15  var img2 = document.createElement("img");
16  img2.setAttribute("class","carMarker");
17  img2.setAttribute("src","../assets/resources/car2.svg")
18  node.appendChild(img2);
19  var node4 = document.createElement("div");
20  node4.setAttribute("id","car2Progress");
21  var node5 = document.createElement("div");
22  node5.setAttribute("id","a-car2bar");
23  node5.innerHTML = "0 %";
24  node4.appendChild(node5);
25  node.appendChild(node4);
26  var time = document.createElement("div");
27  time.setAttribute("id","time");
28  time.innerHTML="Tiempo: 00:00";
29  time.style.marginTop="-87px";
30  time.style.color="white";
31  node.appendChild(time);
32  var myiframe= document.getElementById("myIFrame");
33  myiframe.insertBefore(node,myiframe.childNodes[0]);
34 }

```

- La función que se ejecuta periódicamente comprueba la velocidad de los robots y, si es mayor que 0, se inician los evaluadores, llama a una función que realiza la lógica para actualizar las barras de progreso y añadir un cronómetro al *DOM*. La variable *timeInit* se actualiza constantemente hasta que el usuario ejecuta su código y el *robot* comienza a moverse, que se calcula el tiempo transcurrido y lo muestra en pantalla.

```

1 evaluator.setEvaluator = (arrayRobots) =>
2   let robot=Websim.robots.getHalAPI(arrayRobots[0]);
3   if(!clock){
4     timeInit = new Date();
5   }
6   if(robot.velocity.x>0){

```

```

7   clock=true;
8   var time= document.getElementById("time");
9   progressBar(arrayRobots);
10  var realTime = new Date(new Date() - timeInit);
11  var formatTime = timeFormatter(realTime);
12  time.innerHTML = "Tiempo: " + formatTime;
13 }
14 }
```

- Función que calcula el porcentaje recorrido por cada uno de los *robots*, modifica las barras de progreso y lo añade al *DOM* para que aparezca en texto.

```

1 function progressBar(arrayRobots) {
2   arrayRobots.forEach(function(robotID) {
3     let robot = Websim.robots.getHalAPI(robotID);
4     var left=38.24 + robot.getPosition().x;
5     var completed=(left*100)/78.48;
6     var element = document.getElementById(robot.myRobotID+"bar");
7     if((100-completed)>100){
8       element.style.width = 100 + '%';
9       element.innerHTML = 100 + '%';
10    }else{
11      element.style.width = Math.round(100-completed) + '%';
12      element.innerHTML = Math.round(100-completed) + '%';
13    }
14  });
15 }
```

- Función que da formato al cronómetro.

```

1 function timeFormatter(time){
2   var formatTime;
3   if (time.getMinutes()<10){
4     formatTime="0"+time.getMinutes();
5   }else{
6     formatTime=time.getMinutes();
7   }
8   formatTime+=":";
9   if (time.getSeconds()<10){
10     formatTime+="0"+time.getSeconds();
11   }else{
12     formatTime+=time.getSeconds();
13   }
14   return formatTime;
15 }
```

4.4.3. Sigue-líneas competitivo

En este ejercicio hay dos robots en el que tienen que seguir una linea de color blanco sobre fondo negro atravesando un puente en medio del circuito para que, de esta manera, ambos recorran la misma distancia⁵.

La principal novedad del escenario es el puente creado, que permite ser cruzado mientras siguen la línea. La solución definitiva ha sido creando primitivas de *A-Frame (a-plane)* y añadiendo una textura diseñada con *Photoshop* con el mismo aspecto que el resto del circuito.

El evaluador de este ejercicio es similar al de atravesia-bosque, con la diferencia de que es necesario guardar en todo momento la posición y distancia recorrida por cada robot para calcular el porcentaje del circuito completado.

```

1 var car={
2   pos:{
3     x:robot.getPosition().x,
4     z:robot.getPosition().z
5   },
6   dist: 0
7 }
```

Listing 4.8: Objeto creado para guardar posición y distancia recorrida de un robot

```

1 evaluator.setEvaluator = (arrayRobots) => {
2   let robot1=Websim.robots.getHalAPI(arrayRobots[0]);
3   let robot2=Websim.robots.getHalAPI(arrayRobots[1]);
4   if(!clock){
5     timeInit = new Date();
6     car1 = {
7       pos:{
8         x:robot1.getPosition().x,
9         z:robot1.getPosition().z
10      },
11      dist: 0
12    };
13    car2 = {
14      pos:{
15        x:robot2.getPosition().x,
16        z:robot2.getPosition().z
17      },
18      dist: 0
19    }
20  }
```

⁵https://www.youtube.com/watch?v=OaA7_wsXhk8

```

21 if(robot1.velocity.x>0) {
22   clock=true;
23   var time= document.getElementById("time");
24   progressBar(arrayRobots,[car1,car2]);
25   var realTime = new Date(new Date() - timeInit);
26   var formatTime = timeFormatter(realTime);
27   time.innerHTML = "Tiempo: " + formatTime;
28 }
29 }
```

Listing 4.9: Función que realiza la funcionalidad para llenar la barra de progreso

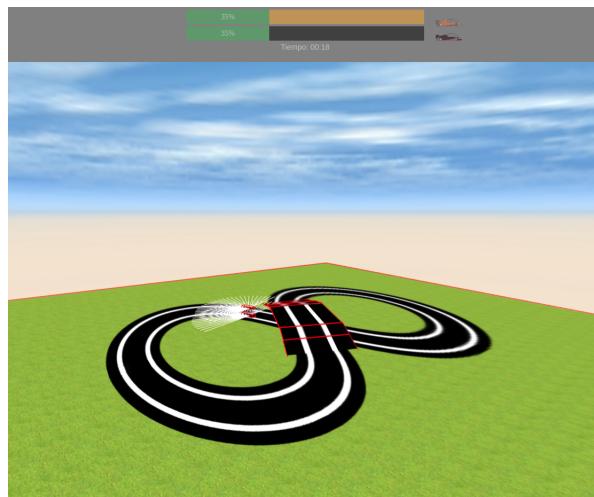


Figura 4.30: Ejercicio y evaluador sigue-líneas competitivo

4.4.4. Gato-ratón

No es estrictamente hablando un ejercicio competitivo directamente, con dos códigos de estudiantes sobre sendos robots en el mismo escenario simultáneamente. En este ejercicio hay dos robots, uno (el *drone gato*) programado por el estudiante que hace el ejercicio y otro (el *drone ratón*) programado por desarrolladores de la plataforma. En este ejercicio el usuario tiene que desarrollar su solución para que el *robot* no se aleje del objetivo, el *drone ratón*, que está en constante movimiento⁶.

Para este ejercicio se ha creado un *script* llamado *agents-methods.js*, basado en *brains*, para ejecutar el código del *drone ratón* sin necesidad de escribir código. Es muy similar al método *runBrain* con la diferencia de que el código viene de un fichero en lugar del editor. En este

⁶https://www.youtube.com/watch?v=xA9Emhdk_HQ

módulo se guarda el código en la variable `agents.code` con el siguiente código:

```

1 agents.getCode = (file) => {
2   var request = new XMLHttpRequest();
3   request.open("GET", file);
4   request.onreadystatechange = function () {
5     if(request.status === 200 || request.status == 0) {
6       agents.code = request.responseText;
7     }
8   }
9   request.send();
10 }

```

Siendo `file` una variable que se inicializa en el `index.html` de manera similar a los ficheros de configuración y evaluadores automáticos. Una vez obtenido el código se llama al método `runAgent` que recoge el código e incorpora la lógica programada en el `array` de `robots` del módulo `brains`.

```

1 agents.runAgent = (robotID, code) =>{
2   code = 'async function myAlgorithm() {\n' +code+'\n} \nmyAlgorithm();';
3   brains.threadsBrains.push({
4     "id": robotID,
5     "running": true,
6     "iteration": brains.createTimeoutBrain(code, Websim.robots.getHalAPI(robotID), robotID),
7     "codeRunning": code
8   });
9 }

```

A la hora de ejecutar el código, se elige si el código que se ejecuta es el que hay en el agente llamando al método `runAgent` del módulo `agents` o el que hay en el editor con el método `runBrain` del módulo `brains`.

En el siguiente código se ejecuta en un `robot` el código escrito en el editor y en otro el escrito en el agente:

```

1 $("#runbtn").click(()=>{
2   if (editFirst) {
3     codeFirst = editor.getCode();
4   } else {
5     codeSecond = editor.getCode();
6   }
7   if (brains.threadExists(editorRobot1)){
8     if (brains.isThreadRunning(editorRobot1)){
9       brains.stopBrain(editorRobot1);
10      brains.stopBrain(editorRobot2);
11    }else{
12      brains.resumeBrain(editorRobot1,codeFirst);
13    }
14  }
15 }

```

```

13     agents.resumeAgent(editorRobot2,agents.code);
14 }
15 }else{
16     brains.runBrain(editorRobot1,codeFirst);
17     agents.runAgent(editorRobot2,agents.code);
18 }
19 });

```

Para el evaluador de este ejercicio se crea un gráfico con ayuda de una librería externa de *JavaScript (JavaScript Graphics Library)*⁷) que muestra la distancia entre *drones* y el tiempo que lleva de ejecución. Se puede ver este evaluador en la figura 4.31. En este caso, el método *createInterface* realiza añade todo lo necesario al *DOM* para que el gráfico sea completo y llama a la función *setAxis()* para añadir los ejes y las etiquetas.

```

1 evaluator.createInterface= ()=>{
2     var node = document.createElement("div");
3     node.setAttribute("id","panel");
4     node.style.height="130px";
5     node.style.backgroundColor="white";
6     var time = document.createElement("div");
7     time.setAttribute("id","time");
8     time.marginLeft="50px";
9     time.innerHTML="Tiempo: 00:00";
10    time.style.color="black";
11    time.style.textAlign="center";
12    node.appendChild(time);
13    var myiframe= document.getElementById("myIFrame");
14    myiframe.insertBefore(node,myiframe.childNodes[0]);
15    myPanel = new jsgl.Panel(document.getElementById("panel"));
16    setAxis(myPanel);
17    line = myPanel.createPolyline();
18    line.getStroke().setColor('blue');
19    line.getStroke().setWeight(2);
20 }

```

Listing 4.10: Función que establece los ejes y etiquetas de la gráfica

```

1 function setAxis(myPanel) {
2     var axisX = myPanel.createLine();
3     axisX.setStartPointXY(20,10);
4     axisX.setEndPointXY(20,100);
5     myPanel.addElement(axisX);
6     var axisY = myPanel.createLine();
7     axisY.setStartPointXY(20,100);

```

⁷<http://www.jsgl.org/>

```

8 axisY.setEndPointXY(500,100);
9 myPanel.addElement(axisY);
10 var myLabel = myPanel.createLabel();
11 myLabel.setLocation(new jsGL.Vector2D(75,100));
12 myLabel.setText("00:30");
13 myPanel.addElement(myLabel);
14 var myLabel = myPanel.createLabel();
15 myLabel.setLocation(new jsGL.Vector2D(0,20));
16 myLabel.setText("10");
17 myPanel.addElement(myLabel);
18 }
```

Listing 4.11: Método *createInterface*

```

1 evaluator.setEvaluator = (arrayRobots) => {
2     var robot1 = Websim.robots.getHalAPI(arrayRobots[0]);
3     var robot2 = Websim.robots.getHalAPI(arrayRobots[1]);
4     if(!clock){
5         timeInit = new Date();
6     }
7     if(robot1.velocity.x >0 || robot2.velocity.x>0){
8         clock = true;
9         var time= document.getElementById("time");
10        var realTime = new Date(new Date() - timeInit);
11        var formatTime = timeFormatter(realTime);
12        time.innerHTML = "Tiempo: " + formatTime;
13        var pos1 = robot1.getPosition();
14        var pos2 = robot2.getPosition();
15        var dist = Math.sqrt(Math.pow(pos2.x-pos1.x,2)+Math.pow(pos2.y-pos1.y,2)+Math.pow(pos2.z-
16            pos1.z,2));
17        line.addPointXY(x,dist+10);
18        x=x+0.5;
19        myPanel.addElement(line);
20    }
}
```

Listing 4.12: Código JavaScript que calcula la distancia entre *drones*, la representa e incorpora un cronómetro al *DOM*



Figura 4.31: Evaluador y escenario con dos robots para ejercicio gato-ratón

Capítulo 5

Conclusiones

Tras detallar las mejoras aportadas a *WebSim*, en este capítulo se recopilan los objetivos alcanzados, se valoran los conocimientos adquiridos y se exponen las posibles líneas de mejora y extensión de la plataforma.

5.1. Conclusiones

El objetivo general de mejorar el entorno docente basado en *WebSim* se ha conseguido ampliamente y con éxito en todos los frentes. Los dos más importantes son el soporte para *drones* y la programación de la infraestructura para ejercicios competitivos en la plataforma.

El primer subobjetivo consistía en añadir soporte a *drone* en *WebSim*. En la sección 4.1 se explica como se ha llevado a cabo aportando el modelo en 3D, *drivers* y bloques para las nuevas funciones. Gracias a ello *WebSim* soporta ahora la programación realista de *drones*, que además de controlables en velocidad de giro y avance, como los robots en tierra, lo son también en velocidad de ascenso y descenso. Igualmente se han incluido órdenes de despegue y aterrizaje. Las físicas se han extendido para materializar movimiento en 3D incluyendo el efecto de la gravedad. También se han creado nuevos modelos de robots como fórmula 1 o *mBot*, que se detallan en la subsección 4.1.4.

El segundo subobjetivo consistía en añadir teleoperadores, que se explica en la sección 4.2 y, además, se han creado archivos de configuración para poder cambiar de escenario y facilitar así su integración en servidor. Estos teleoperadores son útiles principalmente para probar y depurar

el soporte a los nuevos robots en *WebSim*.

El tercer subobjetivo era incluir más ejercicios a *WebSim* y mejorar los existentes. Se han explicado en la sección 4.3 y otorga a la plataforma el poder realizar nuevos ejercicios como choca-gira (subsección 4.3.3), sigue-pelota (subsección 4.3.4) y atraviesa-bosque (subsección 4.3.5). Para todos ellos se ha creado un mundo 3D con sus objetos, obstáculos, físicas activadas, etc.

El último subobjetivo, ejercicios competitivos, se ha descrito en la sección 4.4. Para estos ejercicios se ha llevado a cabo una refactorización que se explica en la subsección 4.4.1. En esta nueva arquitectura se separan los hilos de la simulación, robot y editor dando la posibilidad de crear más de un robot en el mismo escenario, parar la simulación y reanudarla después de haber cambiado el código y crear un hilo que tenga acceso a los sensores de los robots para evaluar su comportamiento. Para su correcto funcionamiento se han refactorizado también todos editores disponibles y se han creado nuevos para los ejercicios competitivos, que se diferencian de los que había disponibles en la interfaz gráfica y que dan la posibilidad de elegir qué *robot* programar.

Para estos ejercicios se ha hecho uso de los *robots* creados en la subsección 4.1.4 y se ha incorporado un evaluador automático por cada ejercicio. Pueden acceder a todos los sensores de los *robots*, pero se han desarrollado para que accedan a su posición y muestren el porcentaje recorrido del circuito (ejercicios sigue-líneas y atraviesa-bosque) o la distancia con otro *robot* (ejercicio gato-ratón).

5.2. Mejoras futuras

El desarrollo de este trabajo ha supuesto un progreso para *WebSim*, pero aún hay muchas posibles vías de desarrollo para su mejora:

- Añadir nuevos modelos de *robots* como la aspiradora robótica *Roomba* o un *robot* con pinzas.

- Añadir más ejercicios a la plataforma, por ejemplo aparcamiento automático o uno basado en coger objetos del entorno. Para ello habrá que construir nuevos escenarios y evaluadores automáticos.
- Explorar el uso de *WebWorkers* en los cerebros para optimizar el rendimiento de *WebSim*. Los *WebWorkers* se pueden ejecutar en diferentes *cores* de la *CPU* del ordenador.
- Explorar la gravedad que simula *A-Frame* para que la simulación sea más realista cuando hay un *drone* en el escenario.
- Establecer un control en posición modificando la arquitectura de cómputo. Actualmente el control de posición es bloqueante (órdenes como “*avanza 1 metro*” lleva un tiempo completarse) frente al control en velocidad que no es bloqueante, que es el que hay implementado. Este carácter bloqueante altera la arquitectura de cómputo de *JavaScript* que tiene *WebSim* actualmente.

Bibliografía

- [1] *HTML5*: <https://developer.mozilla.org/es/docs/HTML/HTML5>
- [2] *Eduación STEAM*: <https://www.aulaplaneta.com/2018/01/15/recursos-tic/educacion-steam-la-integracion-clave-del-exito/>
- [3] *Entornos de Programación Visual para Programación Orientada a Objetos: Aceptación y Efectos en la Motivación de los Estudiantes*. Felipe I. Anfarrutia, Ainhoa Álvarez, Mikel Larrañaga, Juan-Miguel López-Gil. 2017.
- [4] *Extreme programming*: <https://profile.es/blog/programador-extremo-extreme-programming/>
- [5] *Documentación JavaScript*: <https://developer.mozilla.org/es/docs/Web/JavaScript>
- [6] *Introducción JavaScript*: <https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Introducción>
- [7] *Documentación A-Frame*: <https://aframe.io/>
- [8] *Físicas en A-Frame*: <https://github.com/donmccurdy/aframe-physics-system>
- [9] *Animaciones en A-Frame*: <https://blog.prototypypr.io/learning-a-frame-how-to-do-animations-2aac1ae461da>
- [10] *A-Frame extras*: <https://github.com/donmccurdy/aframe-extras/tree/master/src/loaders>
- [11] *Documentación Blockly*: <https://developers.google.com/blockly>

- [12] *Desarrollo web*: <https://www.w3schools.com/>
- [13] *Documentación Blender*: <https://docs.blender.org/manual>
- [14] *Información sobre Blender*: <https://www.desarrollolibre.net/blog/blender/que-es-blender>
- [15] *Manual de modelado y animación con Blender*. Pablo Suau. Universidad de Alicante. 2011.
- [16] *Información sobre glTF*: <https://www.khronos.org/gltf/>
- [17] *WebSim*: <https://www.kibotics.org/>