



GRADO EN INGENIERÍA TELEMÁTICA

Curso Académico 2020/2021

Trabajo Fin de Grado

Integración del Robot Lego Ev3 a la plataforma de Kibotics

Autor : Daniel Pulido Millanes

Tutor : Dr. José María Cañas Plaza

Índice general

Lista de figuras	5
Lista de tablas	7
1. Introducción	1
1.1. Robótica	1
1.1.1. Aplicaciones robóticas	3
1.1.2. Software en robótica	6
1.2. Robótica educativa	7
2. Objetivos	11
2.1. Objetivos del TFG	11
2.2. Metodología	12
2.3. Requisitos	13
2.4. Plan de trabajo	14
3. Soporte Simulado	15
3.1. Características del Lego Ev3	15
3.2. Soporte de sensores	18
3.2.1. Sensor de color	18
3.2.2. Sensor de ultrasonido	21
3.2.3. Modelo 3D, apariencia	24
3.2.4. Bloques Scratch para programación gráfica del drone	26
3.2.5. Nuevos robots	28
3.3. Teleoperadores en WebSim	29

3.3.1. Interfaz gráfica	29
3.3.2. Arquitectura	32
3.3.3. WebSim y sus ficheros de configuración	35
3.4. Nuevos ejercicios individuales	37
3.4.1. Sigue-líneas visión	37
3.4.2. Sigue-líneas infrarrojos	39
3.4.3. Choca-gira	40
3.4.4. Sigue-pelota	41
3.4.5. Atraviesa-bosque	44
3.4.6. Cuadrado con drone	45
3.5. Ejercicios competitivos	46
3.5.1. Arquitectura de cómputo	46
3.5.2. Atraviesa-bosque competitivo	51
3.5.3. Sigue-líneas competitivo	54
3.5.4. Gato-ratón	55

Índice de figuras

1.1.	Imagen clásica de un robot	2
1.2.	Aspiradora robótica <i>Roomba</i>	4
1.3.	Robot médico <i>Da Vinci</i>	4
1.4.	Robot militar <i>Big Dog</i> creado por <i>Boston Dynamics</i>	5
1.5.	Vehículo <i>Waymo</i> de <i>Google</i>	6
1.6.	Robot <i>Perseverance</i> de la <i>NASA</i>	6
1.7.	Interfaz gráfica de Scratch	8
1.8.	Interfaz de LEGO WeDo	9
1.9.	Interfaz gráfica de LEGO Ev3	9
1.10.	Kit de piezas y sensores de LEGO Ev3	10
2.1.	Metodología de modelo iterativo	13
3.1.	Conjunto Ev3	16
3.2.	Robot con sensor de color	17
3.3.	Robot con sensor tactil	17
3.4.	Robot con sensor de ultrasonidos	17
3.5.	Sensor color	18
3.6.	Sensor de ultrasonidos	21
3.7.	Drone en Blender	25
3.8.	Escenario de WebSim con drone integrado	25
3.9.	Modelos de <i>drone</i> de distintos colores	26
3.10.	Bloque de velocidad de ascenso	26
3.11.	Bloque que obtiene la velocidad vertical del robot	26

3.12. Bloque de aterrizaje	26
3.13. Bloque de despegue	27
3.14. Espacio de trabajo de <i>Scratch</i> con los bloques del drone incorporados	28
3.15. Modelos de coches Fórmula 1	28
3.16. Modelo mBot	29
3.18. Interfaz que permite acceder a cada uno de los teleoperadores	31
3.19. Arquitectura de la aplicación teleoperadores	33
3.20. Escenario para el ejercicio <i>piBot</i> sigue-líneas con cámara	38
3.21. Solución en <i>Scratch</i> para el ejercicio sigue-líneas visión	38
3.22. Escenario para el ejercicio para el robot <i>piBot</i> sigue-líneas infrarrojos	39
3.23. Solución en <i>Scratch</i> para el ejercicio sigue-líneas infrarrojos	39
3.24. Escenario para el ejercicio choca-gira	40
3.25. Solución en <i>Scratch</i> para el ejercicio choca-gira	41
3.26. Secuencia del ejercicio <i>drone</i> sigue-pelota	43
3.27. Solución en <i>Scratch</i> para el ejercicio sigue pelota drone	43
3.28. Escenario para el ejercicio atraviesa bosque	44
3.29. Solución en <i>Scratch</i> para el ejercicio atraviesa bosque	44
3.30. Escenario de WebSim para el ejercicio drone cuadrado	45
3.31. Solución en <i>Scratch</i> para el ejercicio cuadrado drone	45
3.32. Editor de <i>JavaScript</i> para ejercicios competitivos	46
3.33. Editor de <i>Scratch</i> para ejercicios competitivos	48
3.34. Escenario y evaluador para el ejercicio atraviesa-bosque	51
3.35. Ejercicio y evaluador sigue-líneas competitivo	55
3.36. Evaluador y escenario con dos robots para ejercicio gato-ratón	56

Índice de cuadros

3.1. Métodos (HAL API) de los actuadores implementados para el drone. 23

Capítulo 1

Introducción

En este capítulo se introducen los conceptos básicos en robótica, de como esta nos ayuda en nuestro día a día, y cual es su estado actual. Y como puede ser un gran recurso en la educación. En lo que se basa este proyecto

1.1. Robótica

La robótica es una rama de las ingenierías y de las ciencias de la computación que se encarga del diseño, construcción, operación, estructura, manufactura y aplicación de los robots. El término *robot* se popularizó con el éxito de la obra R.U.R. (*Robots Universales Rossum*), escrita por Karel Čapek en 1920. En la traducción al inglés de dicha obra la palabra checa *roboťa*, que significa trabajos forzados o trabajador, fue traducida al inglés como robot. Un robot es una entidad virtual o mecánica artificial. Están diseñados con un propósito propio. La independencia creada en sus movimientos hace que sus acciones sean la razón de un estudio razonable y profundo en el área de la ciencia y tecnología. La palabra robot puede referirse tanto a mecanismos físicos como a sistemas virtuales de software, aunque suele aludirse a los segundos con el término de bots.



Figura 1.1: Imagen clásica de un robot

No hay un consenso sobre qué máquinas pueden ser consideradas robots, dentro de este proyecto tomaremos como definición que un robot es un sistema autónomo programable capaz de realizar tareas complejas. Además, todos los robots se componen de tres partes esenciales se componen de sensores, controladores y actuadores.

- **Sensores:** Son los sentidos del robot, con ellos ve, escucha y sabe lo que hay en el entorno. Recogen la información necesaria para que el robot realice la tarea En este grupo se encuentran láseres, cámaras, ultrasonidos u odómetros..

- **Controladores:** El equivalente al cerebro humano, utiliza los datos recogidos por los sensores para elaborar una respuesta para que la lleve a cabo los actuadores.
- **Actuadores:** Equivalen a los músculos humanos, son los que se encargan de interactuar con el entorno para llevar a cabo su tarea. Son brazos mecánicos, motores, etcétera...

1.1.1. Aplicaciones robóticas

Ahora que tenemos las bases de lo que es un robot asentadas podemos hablar de cuales son los principales propósitos de los robots hoy en día, aunque la mayor parte de ellos son utilizados por empresas en labores industriales. Aunque hay otros que podemos encontrar en nuestra vida cotidiana, en casas, hospitales, almacenes de tiendas... Esto es debido a la precisión de algunos trabajos, la eficiencia en el trabajo, la reducción de costes que supone o que pueden realizar acciones de alto riesgo para las personas. Los ejemplos más famosos en estos campos son los siguientes:

- Robots Domésticos: Creados para realizar las tareas del hogar. Los más famosos y destacados en el mercado son los Robots *Roomba*, aspiradores autónomos, y también el primer robot que se ha comercializado para todos los públicos y de manera global. Un gran paso para la robótica



Figura 1.2: Aspiradora robótica *Roomba*

- Robots médicos: Son robots diseñados para el uso en medicina para realizar tareas que requieren mucha precisión como en el caso de una cirugía, con el robot *Da Vinci* o robots diminutos que son capaces de navegar por las venas hasta llegar al corazón y allí realizar la cirugía necesaria.

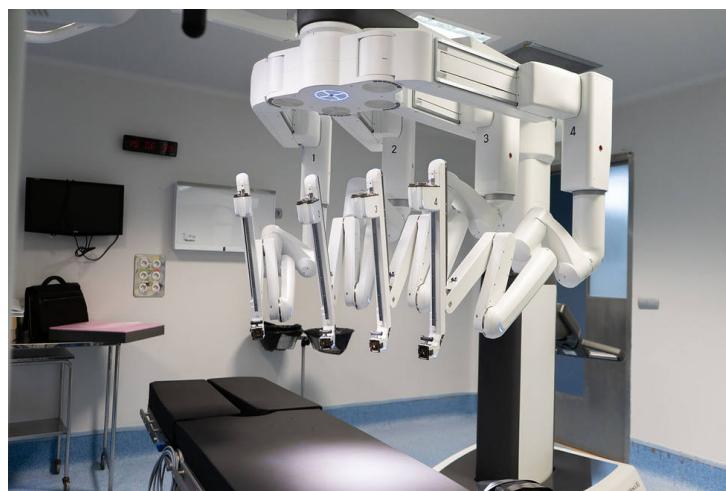


Figura 1.3: Robot médico *Da Vinci*

- Robots militares: Son robots orientados a tareas militares, como reconocimientos de zo-

nas conflictivas o rescate de personas, desactivación de bombas. En los últimos años también se han desarrollado mucho los drones en combate.



Figura 1.4: Robot militar *Big Dog* creado por *Boston Dynamics*

- Vehículos autónomos: Es el campo de la robótica que más en auge esta ahora mismo. El objetivo de estos robots es usar la información que proporcionan sus sensores internos, como cámaras, sensores infrarrojos *Lidar*, y sensores externos como el GPS para llevar de un punto a otro un vehículo.



Figura 1.5: Vehículo Waymo de *Google*

- Robots Espaciales: Los famosos *Rover* de la *NASA* son robots diseñados para entornos donde el ser humano no puede llegar. Se centran en reconocimiento del terreno y análisis de las muestras que recogen.

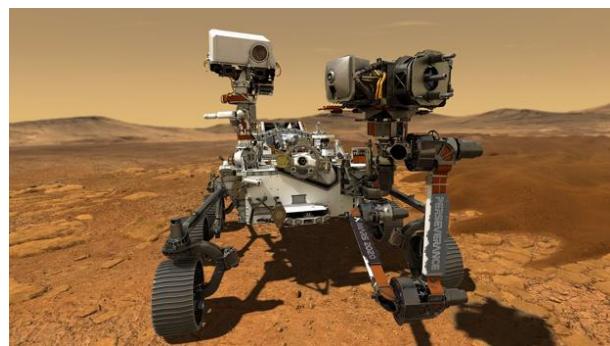


Figura 1.6: Robot *Perseverance* de la *NASA*

1.1.2. Software en robótica

Para dotar de esta inteligencia a los robots se necesitan herramientas que transformen los datos recibidos de los sensores en algo que puedan aplicar en los actuadores. Hace años, cada máquina tenía un software específico con sensores y actuadores únicos para ese robot y esa tarea a desarrollar. Esto hacía, que aunque hubieras implementado el software para otros robots anteriormente, tuvieras que repetir el proceso con cada nuevo robot. Con los años se desarrollaron plataformas de software que permiten desarrollar de manera genérica para todos los robots, y actuando de mediador entre el robot y el software del creador, estos son los llamados *middleware*

que hacen que te puedas abstraer de los *drivers* característicos de cada robot. Los middleware mas importantes a día de hoy son:

- **Robot Operating System (ROS)**[?]. 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 mas herramientas vitales para el desarrollo del robot. Es el mas utilizado a día de hoy porque fue especialmente desarrollado para *UNIX* y luego se implemento para el resto de sistemas operativos
- **ORCA**[?]. Plataforma de *software* libre diseñado para crear aplicaciones mas complejas, ya que esta orientado a las componentes por separado
- **OROCOS**[?] Proyecto de *software* libre también orientado a componentes y basado en C++
- **JdeRobot**[?] Plataforma de desarrollo robótico, en la que se basa este proyecto. Tiene varios nodos programados con varios lenguajes de programación, con compatibilidad con otros *middleware*.

1.2. Robótica educativa

La robotica educativa ha ido tomando mas importancia con los años, ya que cada vez es mas importante que estudiantes de cualquier nivel estén familiarizados con la tecnología, tiene valores positivos como la implementación de pensamiento lógico, resolución de problemas y trabajo en equipo en las actividades académicas, que son ramas del conocimiento que se desarrollan poco en edades tempranas , con una componente en conocimiento matemático y físicos y ademas añade un atractivo que no tienen las asignaturas convencionales. Muchos estudios han demostrado que el uso de kits de robótica en la educación favorece a la capacidad de reflexión de los estudiantes. Cada año se crean mas cursos de robótica, y en 2015 la comunidad de Madrid introdujo la asignatura de robótica en los planes docentes de Enseñanza Secundaria con la asignatura “Tecnología, Programación y Robótica”[?] y en el curso 2020-2021 se empezará a implantar en Educación Primaria la asignatura “Programación y Robótica”[?].

Una de las mayores partes de la robótica tiene que ver con la programación, que además de ser una habilidad muy importante para la sociedad actual, es algo complejo. Por lo que se utilizan lenguajes de programación visual, estos se tratan de lenguajes que abstraen en bloques las funciones o métodos de cualquier lenguaje de programación. Dentro de este tipo de lenguajes, los más destacables son :

- **Scratch[?]:** proyecto liderado por el Grupo *Lifelong Kindergarten* del *MIT*, es utilizado por estudiantes para programar animaciones, juegos e interacciones. Su atractivo reside en lo fácil que es de entender el pensamiento computacional debido a su sencilla interfaz gráfica y la implementación de sus bloques.



Figura 1.7: Interfaz gráfica de Scratch

- **LEGO[?]:** Es el robot base de este proyecto, dispone de una amplia gama de robots programables y cada uno de ellos tiene un sistema gráfico, que es similar entre ellos pero también ligado a la edad el estudiante para el que está diseñado el software.

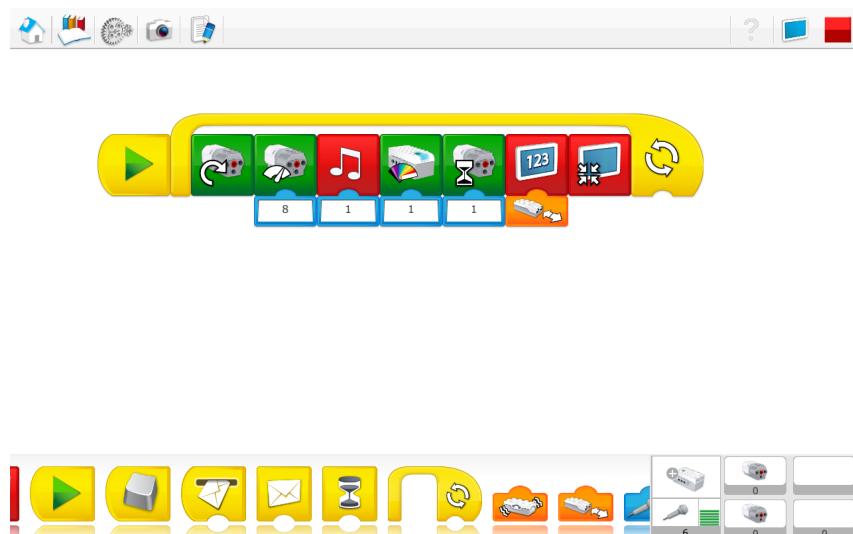


Figura 1.8: Interfaz de LEGO WeDo

Por ejemplo en la figura 1.8 se puede ver que la interfaz en este caso, es con colores vivos, los cuales representan distintas funcionalidades dentro del robot, es decir, el amarillo representa las acciones propias de programación, como: inicio de programa, fin de programa, bucles, esperar, etcétera. El color rojo representa los sensores del robot, todo lo que recoja datos. Y el color verde representa los motores que equivalen a los actuadores en este robot. Como se puede observar es una abstracción muy simple para estudiantes de mas corta edad.

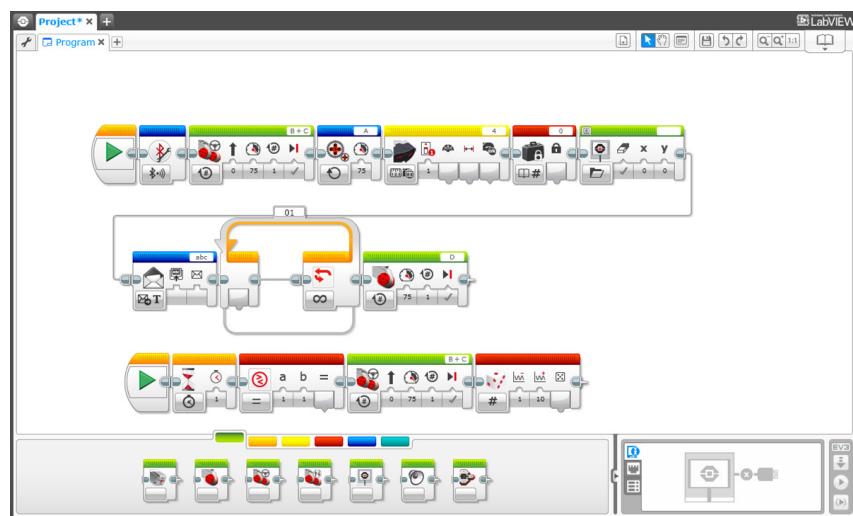


Figura 1.9: Interfaz gráfica de LEGO Ev3

En el caso del software para el **LEGO Ev3**, añade un grado de complejidad, incluyendo

apartados para realizar operaciones matemáticas, envío de archivos entre robots, y añade más actuadores, como la pantalla que integra el robot, o los altavoces.

En el caso de LEGO y en otros kits incorporan los elementos básicos para la construcción de un robot. En este en particular viene con lo indispensable para construir con piezas de LEGO. También incluye un microprocesador para ser programado, con Linux instalado, sensores (infrarrojos, táctiles y de color) y motores.



Figura 1.10: Kit de piezas y sensores de LEGO Ev3

En el siguiente capítulo, profundizare mas en lo que se puede hacer con el robot **LEGO Ev3** y explicare cuales van a ser los objetivos y porque elegir este robot.

Capítulo 2

Objetivos

Una vez explicado en el ámbito en el que se realiza este proyecto, en este capítulo explicaremos los objetivos que se han tratado de alcanzar y el método de trabajo que se ha seguido para lograrlo

2.1. Objetivos del TFG

El objetivo de este trabajo es dar soporte en la plataforma de *Kibotics*¹ a un robot bastante popular para la robótica educativa como es el *LEGO Ev3*, esto significa integrarlo de forma que se pueda programar dentro de la plataforma, y que también funcione para el robot real.

Soporte Simulado

- Añadir una simulación 3D realista sabiendo que hay modelos de robots prediseñados por *LEGO* de varios modelos de robots, al menos uno por cada tipo de sensor que pueda llevar. Y que tenga sentido físico dentro de la simulación.
- Añadir la infraestructura necesaria como *drivers*, y funciones al *Robot API* para que el robot se programable en cualquier lenguaje soportado por la plataforma, y funcione en el robot real
- Crear un conjunto de ejercicios para que haya un temario fácil de seguir por el estudiante, y con una curva de dificultad moderada

¹<https://kibotics.org/>

Soporte Real

- Instalar una imagen de un sistema operativo en el *Lego ev3* en este caso, una distribución basada en *Debian Linux*.
- Instalar un servidor en el robot capaz de recibir mensajes con el código, y lo transforme en un archivo y lo ejecute dentro de la máquina.

2.2. Metodología

La metodología para completar el trabajo de fin de grado se puede dividir en diferentes fases que se iban repitiendo cada cierto tiempo, en cada una de ellas, semanalmente, tenía lugar una reunión con el tutor del trabajo para determinar los siguientes objetivos a cumplir y evaluar las tareas propuestas en anteriores sesiones. Esto ayuda mucho en proyectos como este, en constante desarrollo.

Este proyecto se lleva a cabo con un equipo de trabajo, que se ocupa de la plataforma de *Kibotics*, cada uno con sus labores y ocupaciones. Por lo tanto, es necesaria la comunicación y realimentación con el resto de integrantes. Para ello se utiliza la herramienta *Slack*² en la que los desarrolladores están en contacto en todo momento, no solo para comunicar avances, si no también para ayudar en todo momento si surge algún contratiempo en el desarrollo.

Para trabajar en local, con el repositorio original me hice *git clone* de los repositorios que necesitaba, e iba trabajando sobre ellos, guardando los cambios en un repositorio creado solo para el trabajo de fin de grado³ hasta que la funcionalidad estaba completa, que era cuando se subían al repositorio principal. Esta metodología, es el llamado modelo iterativo de desarrollo de software.

²<https://slack.com/>

³<https://github.com/RoboticsLabURJC/2020-tfg-daniel-pulido>

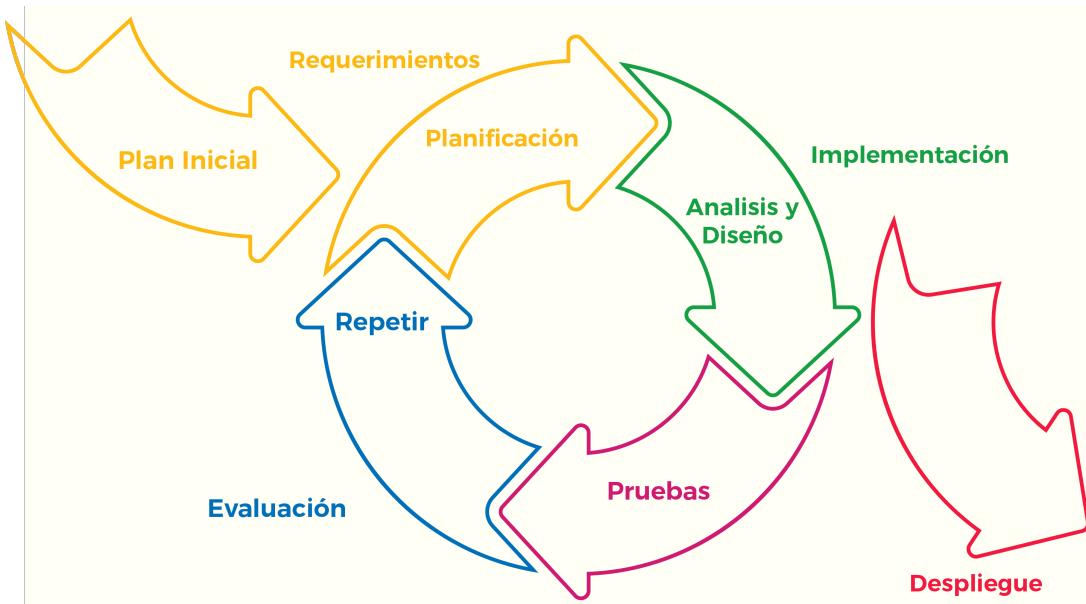


Figura 2.1: Metodología de modelo iterativo

Cuando los cambios eran revisados por el tutor, se seguía una dinámica de *Incidencia y parche* en el repositorio principal. Para ello se creaba una incidencia (*issues*) con el tema que se iba a solucionar o añadir en la funcionalidad de la plataforma, y una vez resuelto en local y para cerrarla, se creaba una rama (*branch*) creando parche (*pull request*) para que un desarrollador principal de *Kibotics*, lo aceptará y fusionará con la rama principal y así arreglar la incidencia. Esto se hace para registrar todos los cambios, y comprobarlos antes de que se trabaje directamente en el repositorio

2.3. Requisitos

Para completar la integración del robot **LEGO Ev3** se necesitan ciertos requisitos que cumplir:

- Dentro del **LEGO Ev3** debe correr una distribución de *Linux*, en este caso he optado por la imagen creada por *ev3dev*, que es una distribución basada en el sistema operativo *Debian Linux*, de la cual entrare en mas detalle en el Capítulo 5: *Soporte Lego ev3 físico*.
- El resultado final debe ser lo más sencillo posible, no debe requerir configuraciones adicionales. Este software tiene que estar diseñado para estudiantes.

2.4. Plan de trabajo

El plan de trabajo a seguir para conseguir el objetivo se puede dividir en los siguientes pasos:

- *Paso 1:* Aterrizaje en *Kibotics*. Lo primero que hay que hacer es familiarizarse con el entorno con el que se va a trabajar. *Kibotics* es una plataforma web en la que entraremos mas en detalle en el siguiente capitulo
- *Paso 2:* Comienzo de la creación del robot simulado. Comenzaremos por la creación de varios modelos 3D para usarlos en el entorno simulado, son varios diferentes ya que lo original que tiene *LEGO* es poder construir tu robot en base a la tarea que vaya a realizar, con las piezas que incluye el kit .
- *Paso 3:* Desarrollo de las funciones y ejercicios dentro de la aplicación de *Kibotics* para crear una dinámica de trabajo con el nuevo robot.
- *Paso 4:* Dar soporte al robot real para poder programarlo en *python* desde el exterior, y la instalación de un server para que pueda recibir código y ejecutarlo en local.
- *Paso 5:* Creación de los *drivers* que hagan de traductor entre lo que creas en la plataforma, y lo que entiende el robot.

Capítulo 3

Soporte Simulado

Ahora que tenemos definidos cuales van a ser nuestros objetivos, y la infraestructura que vamos a utilizar para llevarlos acabo, en este capítulo, se va a hablar del proceso de integración en la plataforma de la parte simulada del robot. Pero antes de eso se darán unas nociones básicas de las características de este robot, para poder entender, el proceso de desarrollo.

3.1. Características del Lego Ev3

Dentro de los robots de *LEGO* el que mas versatilidad, además de más potencia de procesamiento y posibilidades en las opciones de creatividad a la hora de crear robots es *LEGO MINDSTORMS Education*(Pack en el que viene integrado el *LEGO Ev3*) tambien trae una mayor gama de sensores, por no decir que es uno de los lideres en la educación STEM (siglas en inglés de Ciencias, Tecnología,Ingeniería y Matemática), En el centro de *LEGO MINDSTORMS Education* se encuentra el Bloque EV3, el bloque inteligente programable que controla motores y sensores y además proporciona comunicación inalámbrica como quiera que sea.

Descripción general

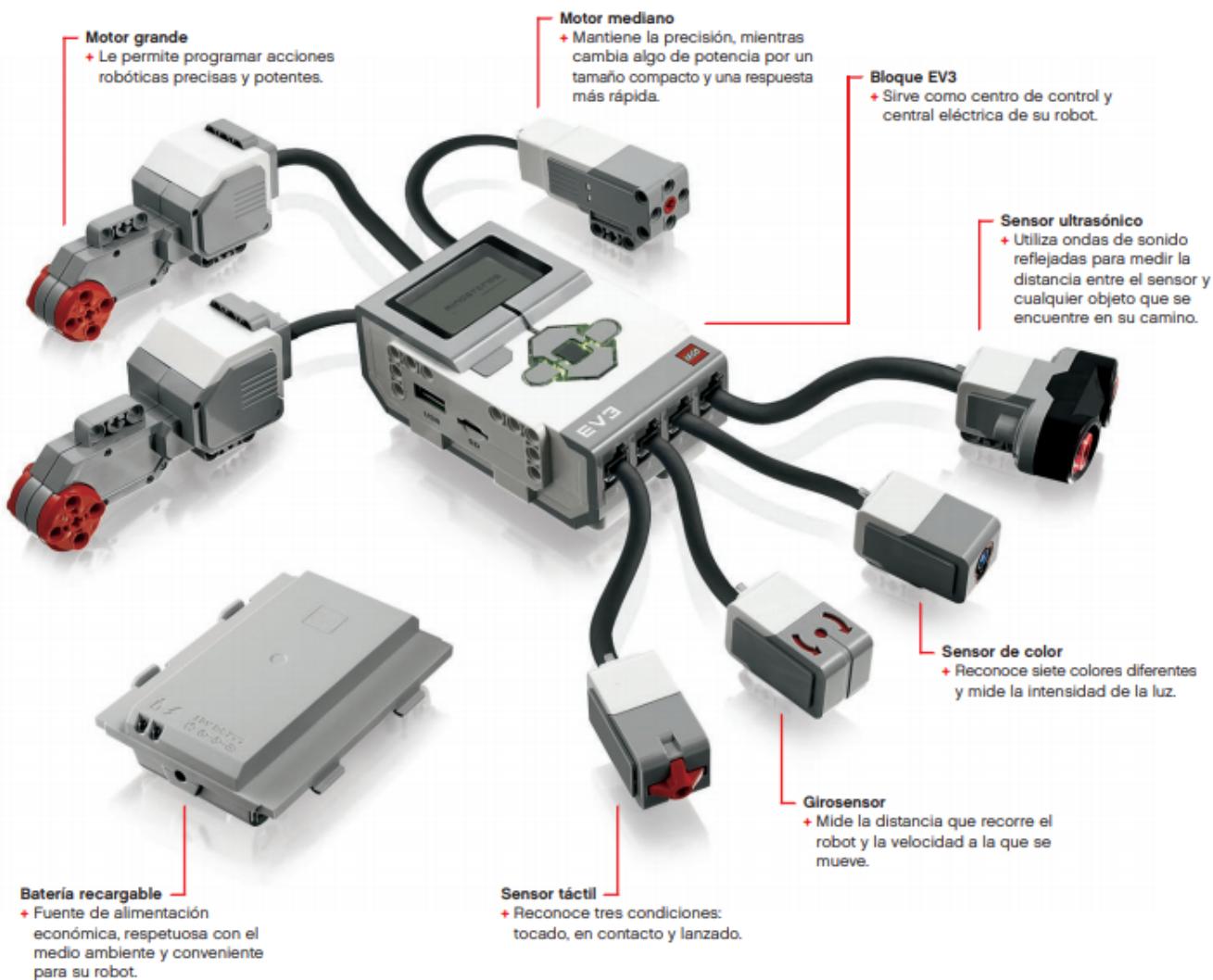


Figura 3.1: Conjunto Ev3

Esto quiere decir que las posibilidades a la hora de crear diferentes robots, con diferentes configuraciones de sensores, son prácticamente infinitas. Pero vamos a centrarnos en los casos que mas funcionalidad tienen. El robot que mas versatilidad presenta a la hora de superar ejercicios, es un robot triciclo, con dos ruedas delanteras y una pivotante trasera. Así que este será nuestro modelo para el robot. Y ademas como el *kit de LEGO MINDSTORMS* viene con tres sensores diferentes, crearé tres modelos para integrarlos por separado en la plataforma.

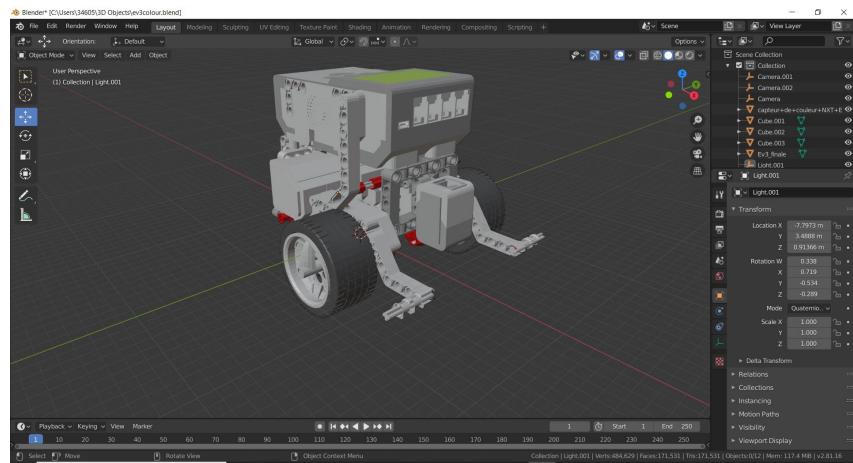


Figura 3.2: Robot con sensor de color



Figura 3.3: Robot con sensor tactil

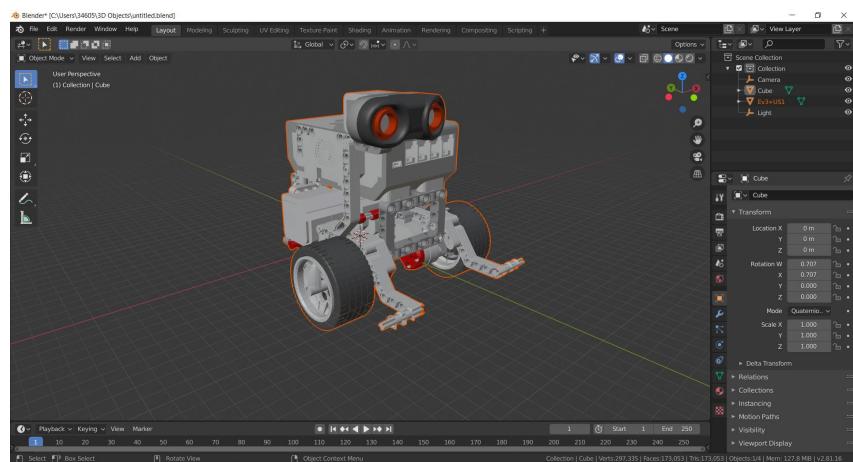


Figura 3.4: Robot con sensor de ultrasonidos

Para la tarea del modelaje 3D, he usado el programa de *Blender*.

Ahora que tenemos los tres modelos, voy a dividir el soporte del robot simulado en los diferentes sensores que implementar.

3.2. Soporte de sensores

3.2.1. Sensor de color

El primer sensor que vamos a analizar es el sensor de color. El Sensor de color es un sensor digital que puede detectar el color o la intensidad de la luz que ingresa por la pequeña ventana de la cara del sensor. Este sensor puede utilizarse en tres modos diferentes: Modo color, Modo intensidad de la luz reflejada y Modo intensidad de la luz ambiental.

La tasa de muestreo del sensor de color es de 1 kHz.



- **En Modo color**, el Sensor de color reconoce siete colores: negro, azul, verde, amarillo, rojo, blanco y marrón, además de Sin color. Esta capacidad de diferenciar los colores significa que su robot puede estar programado para clasificar pelotas o bloques de colores, y realizar acciones diferentes con cada color detectado.

Este tipo de acciones, están ya contempladas en la funcionalidad del *HAL API* de *Kibotics*. Aunque en este caso utiliza una cámara simulada para comprobar que color está viendo.

getImage(cameraID): Método que devuelve *robot*.



Modo intensidad de la luz ambiental

Figura 3.5: Sensor color

Sensor de color en sus 3 usos

```

1   function getImage(cameraID) {
2     /**
3      * Returns a screenshot from the robot camera
4      */
5     if (!cameraID || (this.camerasData.length === 1) ||
6         (cameraID > this.camerasData.length - 1)) {
7       return this.camerasData[0]['image'];
8     } else {
9       return this.camerasData[cameraID]['image'];

```

```

10      }
11
12  }
```

El *LEGO EV3* no tiene una camara instalada, pero para el robot simulado, es lo mas sencillo de implementar, ya que puede analizar la imagen simulada y sacar el color RGB, para posteriormente dar nombre al color que ve.

getColorRGB(): Método que devuelve *RGB* en tres valores.

```

1  function getObjectColorRGB(lowval, highval) {
2
3      /**
4       * This function filters an object in the scene with a given color, uses OpenCVjs to
5       * filter
6       * by color and calculates the center of the object.
7       *
8       * Returns center: CenterX (cx), CenterY (cy) and the area of the object detected in the
9       * image.
10
11  */
12
13  if (lowval.length === 3) {
14      lowval.push(0);
15  }
16  if (highval.length === 3) {
17      highval.push(255);
18  }
19  var image = this.getImage();
20  var binImg = new cv.Mat();
21  var M = cv.Mat.ones(5, 5, cv.CV_8U);
22  var anchor = new cv.Point(-1, -1);
23  var lowThresh = new cv.Mat(image.rows, image.cols, image.type(), lowval);
24  var highThresh = new cv.Mat(image.rows, image.cols, image.type(), highval);
25  var contours = new cv.MatVector();
26  var hierarchy = new cv.Mat();
27
28  cv.morphologyEx(image, image, cv.MORPH_OPEN, M, anchor, 2,
29      cv.BORDER_CONSTANT, cv.morphologyDefaultBorderValue()); // Erosion followed by
30          dilation
31
32  cv.inRange(image, lowThresh, highThresh, binImg);
33  cv.findContours(binImg, contours, hierarchy, cv.RETR_CCOMP, cv.CHAIN_APPROX_SIMPLE);
34  if (contours.size() > 0) {
35
36      let stored = contours.get(0);
37      var objArea = cv.contourArea(stored, false);
```

```

33
34     let moments = cv.moments(stored, false);
35     var cx = moments.m10 / moments.m00;
36     var cy = moments.m01 / moments.m00;
37
38 }
39 return {center: [parseInt(cx), parseInt(cy)], area: parseInt(objArea)};
40 }

```

Una vez realizado este paso podemos ponerle un nombre al color, como hace el *LEGO EV3* real.

- **En Modo intensidad de la luz reflejada**, el Sensor de color mide la intensidad de la luz que se refleja desde una lámpara emisora de luz color rojo. El sensor utiliza una escala de 0 (muy oscuro) a 100 (muy luminoso). Esto significa que su robot puede estar programado para moverse sobre una superficie blanca hasta detectar una línea negra o para interpretar una tarjeta de identificación con código de color. Esto en el robot simulado, es diferente, ya que no podemos ver como una magnitud física como es la luz se refleja un objeto, pero este efecto depende del color que se este viendo en la imagen, la luminosidad del color se puede calcular con esta función:

getLightness(valueMin, valueMax): Método que devuelve *Luminosidad*.

```

1   function getLightness(valueMin, valueMax) {
2
3     let image = this.getObjectColorRGB(valueMin, valueMax);
4     let L = ((image.center[0]- image.center[1])/2)*100/255;
5
6     /**
7      * Returns lightness with a percent
8     */
9
10    return L;
11
12 }

```

Esta función puede resultar útil, para, por ejemplo, poder seguir una línea, cuando haya colores muy similares, o cuando se esté utilizando el robot en una mesa o superficie alta, detectar antes, donde está el borde.

- **En Modo intensidad de la luz ambiental**, el Sensor de color mide la intensidad de la luz que ingresa en la ventana desde su entorno, como la luz del sol o el haz de una linterna.

El sensor utiliza una escala de 0 (muy oscuro) a 100 (muy luminoso). Esta funcionalidad, no puede ser implementada en la plataforma, ya que no tenemos un foco de luz que se puede analizar, ni tampoco una magnitud dentro del entorno que represente la luz.

3.2.2. Sensor de ultrasonido

El primer sensor que vamos a analizar es el sensor de color. El Sensor de color es un sensor digital que puede detectar el color o la intensidad de la luz que ingresa por la pequeña ventana de la cara del sensor. Este sensor puede utilizarse en tres modos diferentes: Modo color, Modo intensidad de la luz reflejada y Modo intensidad de la luz ambiental.

La tasa de muestreo del sensor de color es de 1 kHz.

Se han creado las siguientes funciones para ello:

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

```

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



Figura 3.6: Sensor de ultrasonidos

- ***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 3.1 se recopilan todas las funciones del *HAL API* que extienden la plataforma para dar soporte a *drones*.

Cuadro 3.1: 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*.

3.2.3. 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 repositorio[?] y se ha retocado en *Blender* (figura 3.7) 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 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 despega del suelo.

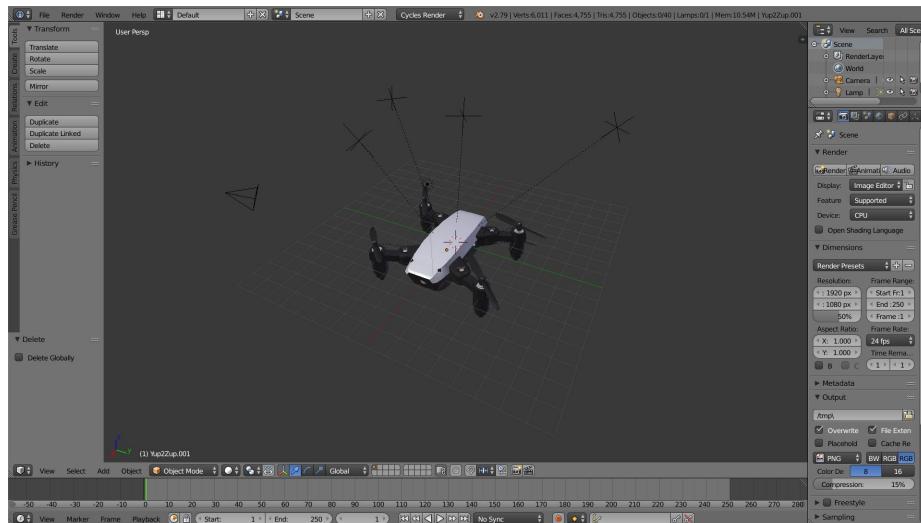


Figura 3.7: Drone en Blender

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*¹.

El drone implementado en el entorno de *WebSim* se puede ver en la figura 3.8.

¹https://github.com/RoboticsLabURJC/2019-tfg-ruben-alvarez/blob/master/upgrades/drones/drone_animation.gltf

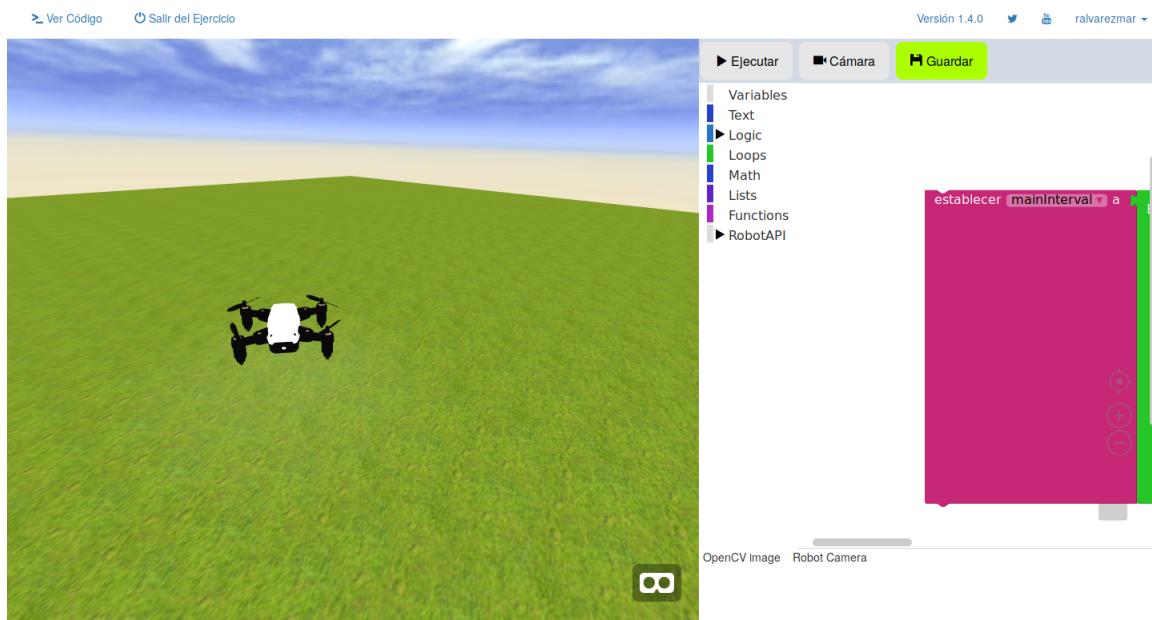


Figura 3.8: Escenario de WebSim con drone integrado

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

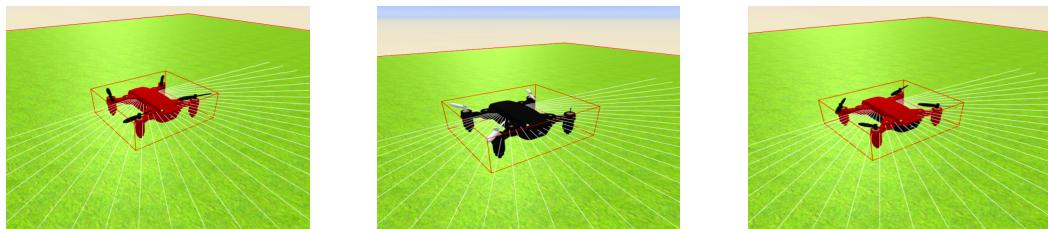


Figura 3.9: Modelos de *drone* de distintos colores

3.2.4. 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.

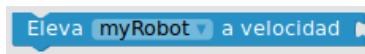


Figura 3.10: Bloque de velocidad de ascenso

- **Obtener velocidad de elevación.** Devuelve la velocidad vertical del robot.



Figura 3.11: Bloque que obtiene la velocidad vertical del robot

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

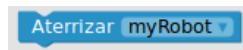


Figura 3.12: Bloque de aterrizaje

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



Figura 3.13: Bloque de despegue

Como ejemplo ilustrativo se muestra en detalle la traducción del bloque aterrizar, 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": "Aterrizar %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'), Blockly.Variables.NAME_TYPE);
25      var value_robotvar = Blockly.JavaScript.valueToCode(block, 'ROBOTVAR', Blockly.JavaScript.ORDER_ATOMIC);
26      var code = variable_name + '.aterrizar();\n';
27      return code;
28    };
  
```

```

29 Blockly.Python['land'] = function(block) {
30   var variable_name = Blockly.Python.variableDB_.getName(block.getFieldValue('NAME'), Blockly.Variables.NAME_TYPE);
31   var value_robotvar = Blockly.Python.valueToCode(block, 'ROBOTVAR', Blockly.Python.ORDER_ATOMIC);
32   var code = variable_name + '.aterrizar()\r\n';
33   return code;
34 };
35 }

```

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:

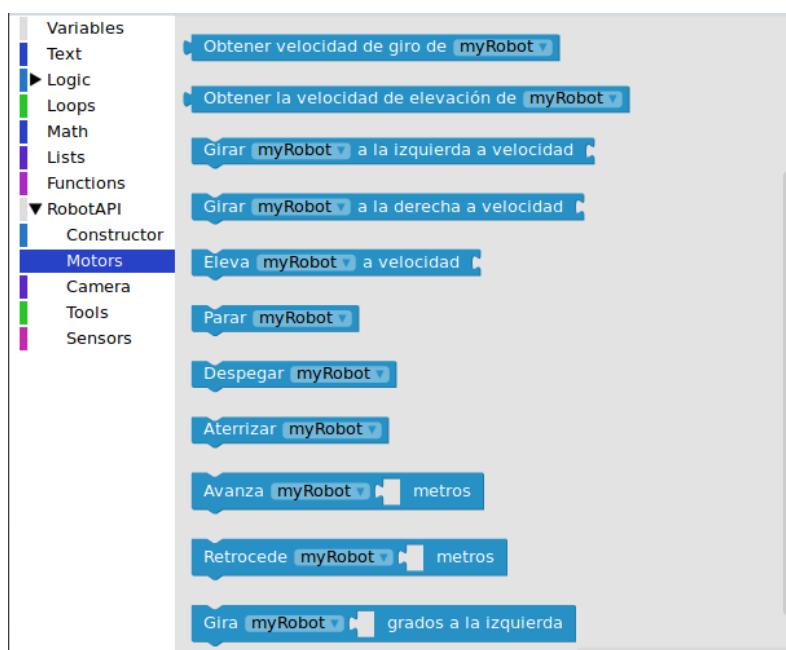


Figura 3.14: Espacio de trabajo de *Scratch* con los bloques del drone incorporados

3.2.5. 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.

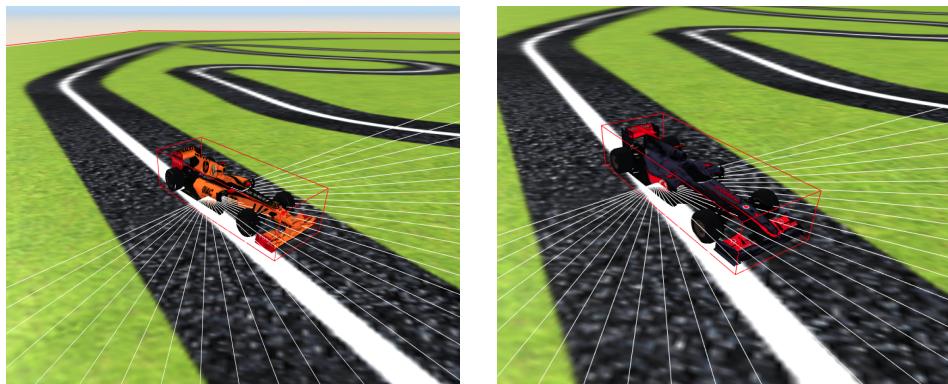


Figura 3.15: Modelos de coches Fórmula 1

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

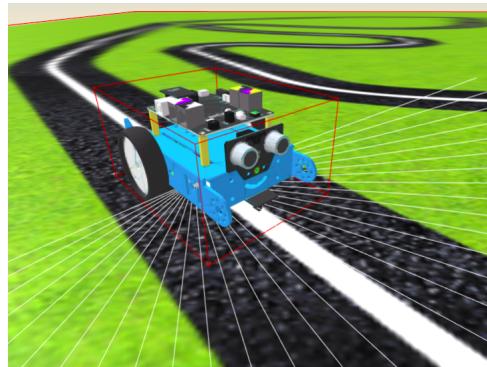


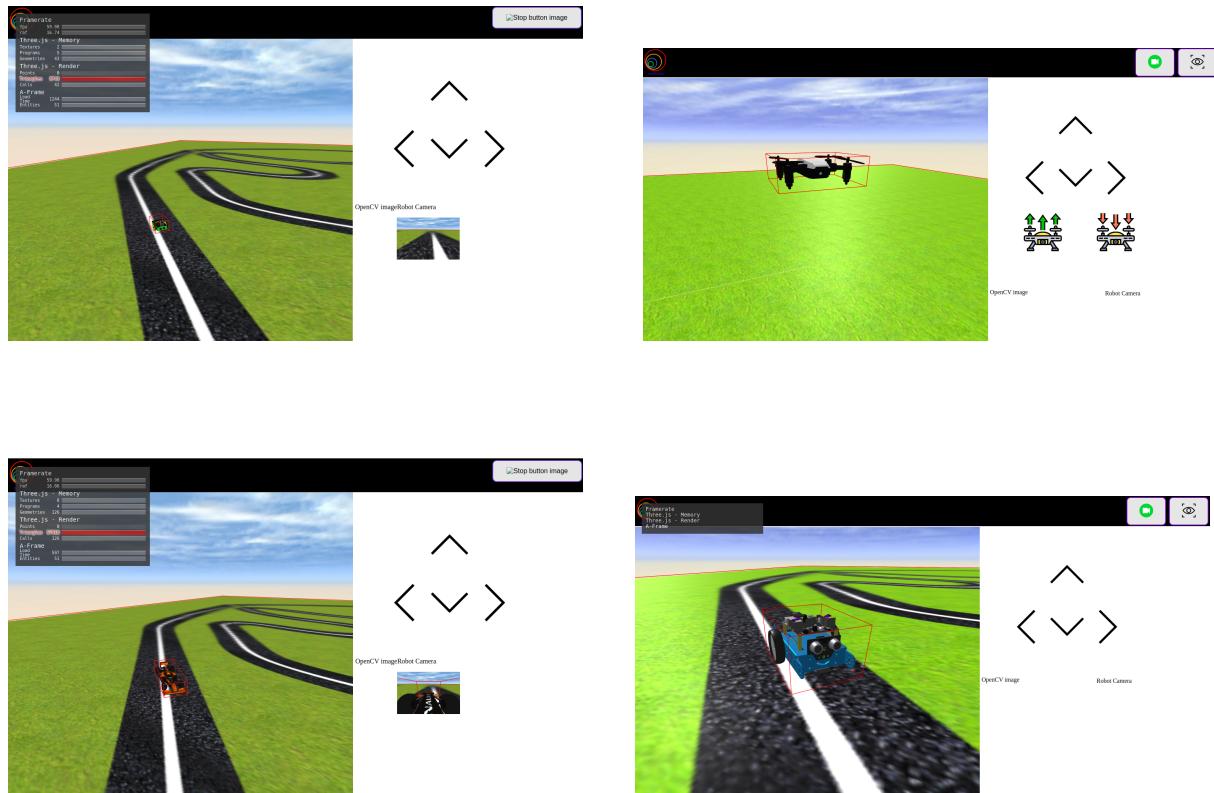
Figura 3.16: Modelo mBot

3.3. 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.

3.3.1. 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="speed"></button>
5     </div>
6     <div id="bottomArrows">
7       <button onclick class="buttonArrow" id="left"></button>
8       <button onclick class="buttonArrow" id="brake"></button>
9       <button onclick class="buttonArrow" id="right"></button><br>

```

```

10  </div>
11  <div id="takeoff">
12    <button onclick id="up"></
      button>
13    <button onclick id="down"><
      /button>
14  </div>
15 </div>

```

Listing 3.1: Código HTML del teleoperador del drone

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

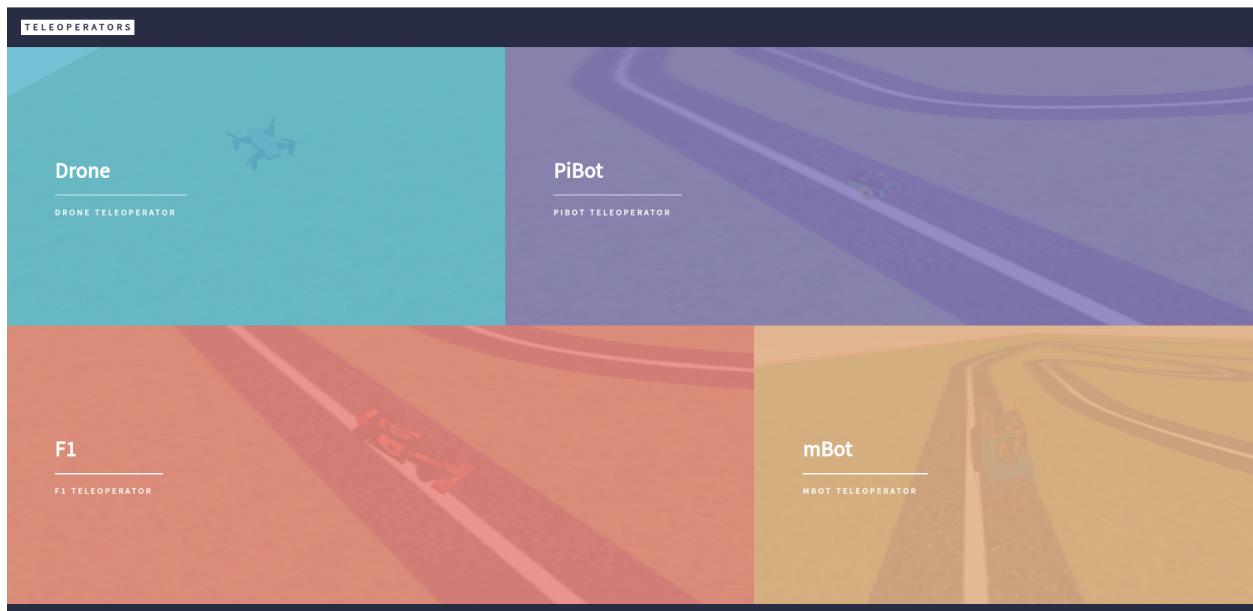


Figura 3.18: Interfaz que permite acceder a cada uno de los teleoperadores

```

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
18      <article>
19          <span class="image">
20              
21          </span>
22          <header class="major">
23              <h3><a href="pibot.html" class="link">PiBot</a></h3>
24              <p>PiBot teleoperator</p>
25          </header>
26
27      </article>
28
29      <article>
30          <span class="image">
31              
32          </span>
33          <header class="major">
34              <h3><a href="f1.html" class="link">F1</a></h3>
35              <p>F1 teleoperator</p>
36          </header>
37
38      </article>
39
40      <article>
41          <span class="image">
42              
43          </span>
44          <header class="major">
45              <h3><a href="mBot.html" class="link">mBot</a></h3>
46              <p>mBot teleoperator</p>
47          </header>
48
49      </article>
50
51  </section>
52
53  </div>
54
55  </div>
56
57 </body>

```

Listing 3.2: Código HTML de la interfaz para acceder a los teleoperadores

3.3.2. Arquitectura

Estos teleoperadores tienen una arquitectura *software* similar a las aplicaciones creadas de *Scratch* o *JavaScript*. En la figura 3.19 se muestra un esquema con el diseño de esta aplicación:

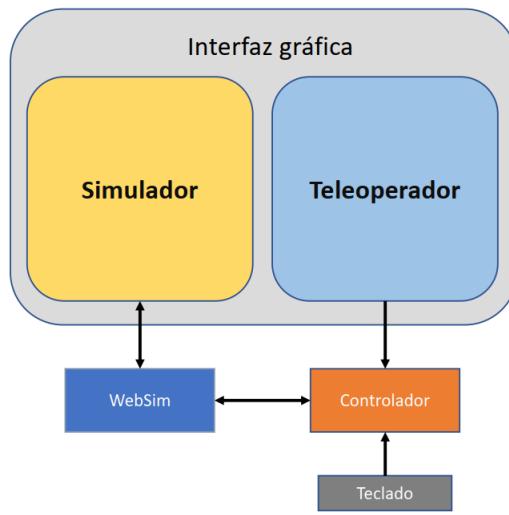


Figura 3.19: Arquitectura de la aplicación teleoperadores

Siendo el bloque teleoperador la interfaz creada en *HTML* (*listing 3.1*). 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 }
```

3.3.3. 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, que deja de tener estos elementos directamente en el código fuente.

Estos archivos se han creado en *JSON* y se ha programado un *script* (*listing 3.3*) para cargar cada fichero de configuración. Para ello se crea una variable en el *index.html* del editor (*listing*

3.4) 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 3.3: *script* que carga los ficheros de configuración

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

Listing 3.4: 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.

3.4. 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*.

3.4.1. Sigue-líneas visión

Este ejercicio consiste en seguir una línea blanca en el suelo 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.

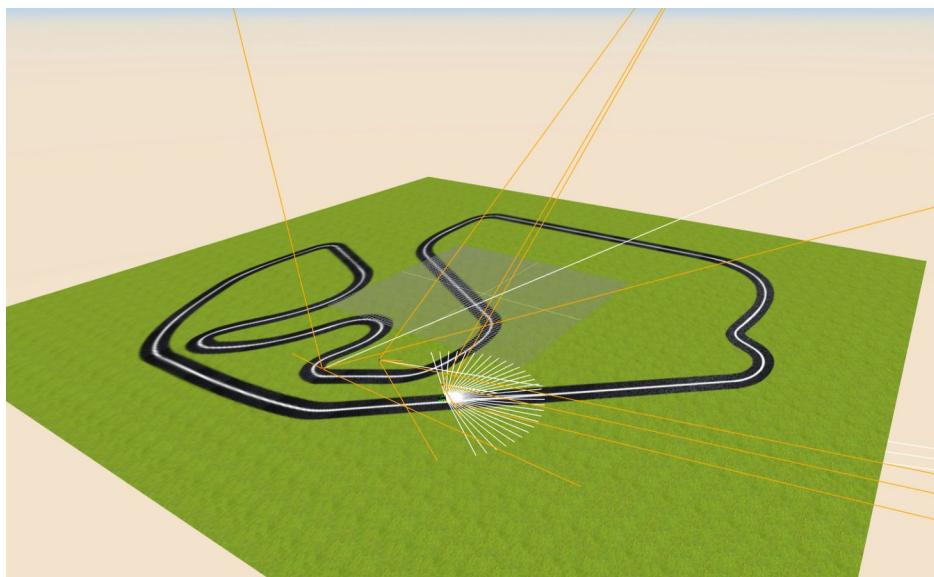


Figura 3.20: Escenario para el ejercicio *piBot sigue-líneas con cámara*

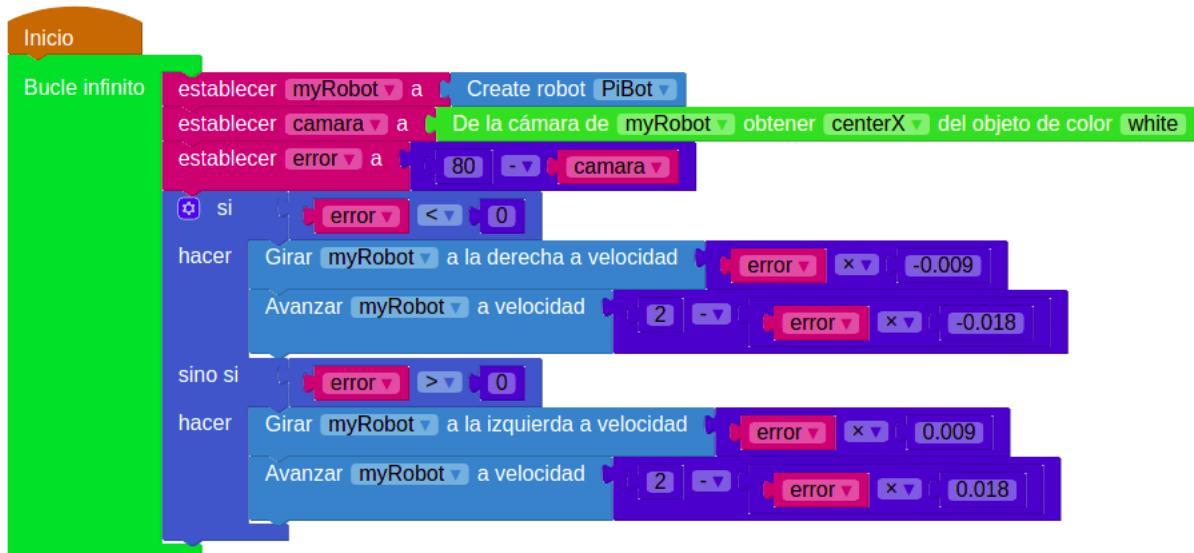


Figura 3.21: Solución en *Scratch* para el ejercicio sigue-líneas visión

En esta solución² 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.

²https://youtu.be/fRqW_BETk

3.4.2. Sigue-líneas infrarrojos

Este ejercicio consiste en seguir una línea negra en el suelo sobre fondo blanco haciendo uso de los sensores infrarrojos del *robot* que apuntan hacia abajo.

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 *undescribedColors* para realizar el filtro y poder pasar “white” como atributo a la función *getObjectColor()*.



Figura 3.22: Escenario para el ejercicio para el *robot piBot* sigue-líneas infrarrojos

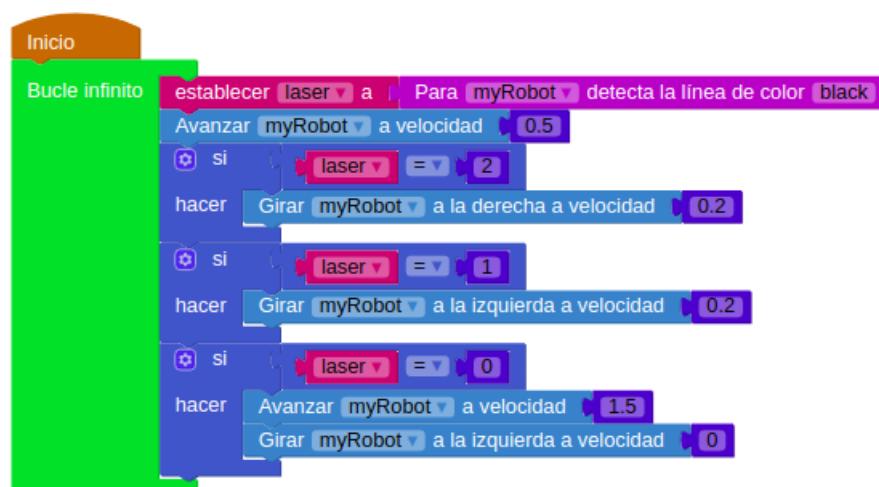


Figura 3.23: Solución en *Scratch* para el ejercicio sigue-líneas infrarrojos

En esta solución³ 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 dónde detecte la línea.

3.4.3. 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*.

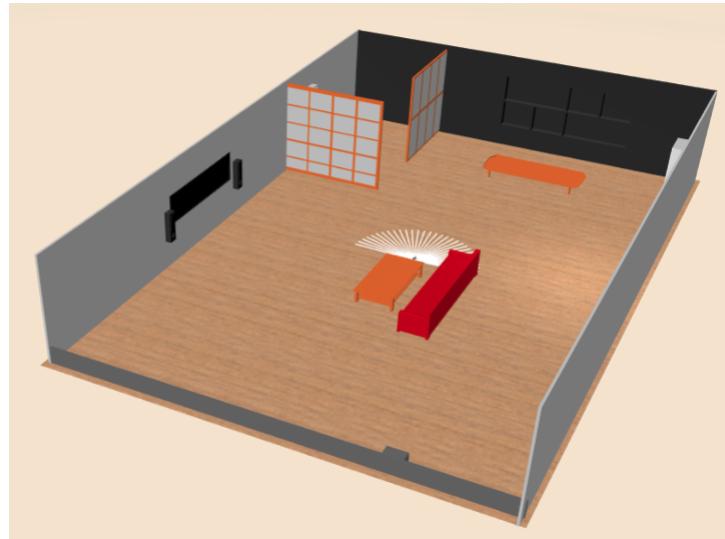


Figura 3.24: Escenario para el ejercicio choca-gira

³<https://youtu.be/d4HETtPmaha>

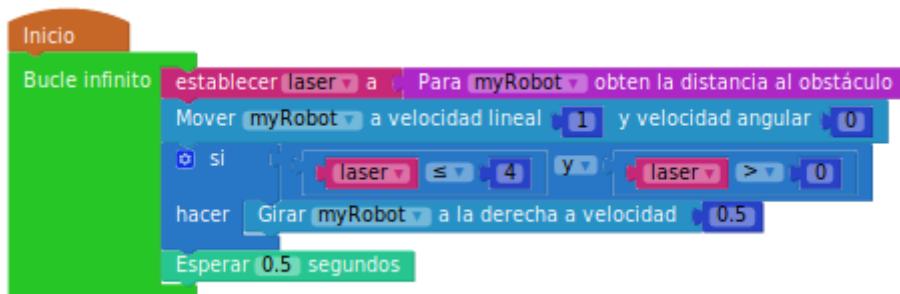


Figura 3.25: Solución en *Scratch* para el ejercicio choca-gira

En esta solución⁴ 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.

3.4.4. 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 negro, 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 3.26 se puede ver una secuencia con la animación de una 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": [

```

⁴https://youtu.be/lXL31Tbgp_E

```

14  {
15    "type": "a-sphere",
16    "id": "redBall",
17    "position": "4 15 20",
18    "color": "#000000",
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:
      alternate; dur: 10000; loop: true",
29    "animation__6": "property: position; from: -10 15 10 ;to: 20 8 -30; delay: 50000; dir:
      alternate; dur: 10000; loop: true"
30  }
31 ]
32 }
```

Haciendo especial mención al campo *objects*, en el que se crea una pelota negra con la animación indicada en todos los campos *animation* y genera el siguiente elemento en *HTML*, que incluye en el *DOM*:

```

1 <a-sphere id="blackBall" position="12 1 25" color="#000000" radius="1.5"
2   animation="property: position; from: 4 15 20 ;to: 0 15 -20; dir: alternate; dur: 10000;
3     loop: true"
4   animation__2="property: position; from: 0 15 -20 ;to: 0 2 -20 ; delay: 10000; dir:
      alternate; dur: 10000; loop: true"
5   animation__3="property: position; from: 0 2 -20 ;to: 4 2 20 ; delay: 20000; dir: alternate
      ; dur: 10000; loop: true"
6   animation__4="property: position; from: 4 2 20 ;to: 4 15 20; delay: 30000; dir: alternate;
      dur: 10000; loop: true"
7   animation__5= "property: position; from: 4 15 20 ;to: -10 15 10; delay: 40000; dir:
      alternate; dur: 10000; loop: true"
8   animation__6="property: position; from: -10 15 10 ;to: 20 8 -30; delay: 50000; dir:
      alternate; dur: 10000; loop: true">
9 </a-sphere>
```

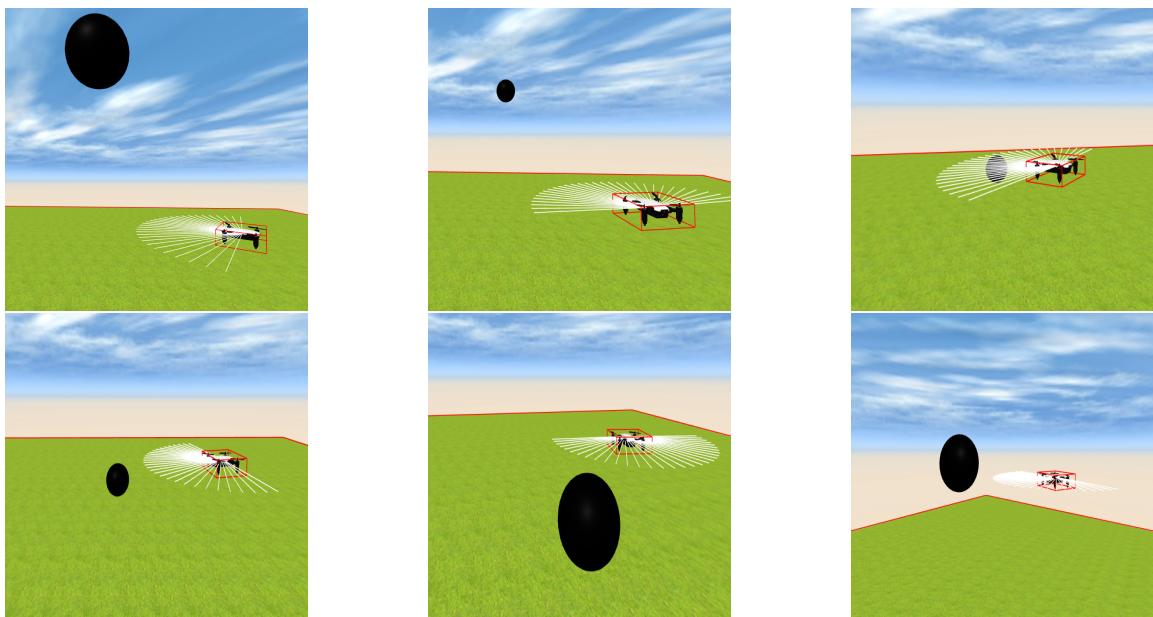


Figura 3.26: Secuencia del ejercicio *drone sigue-pelota*

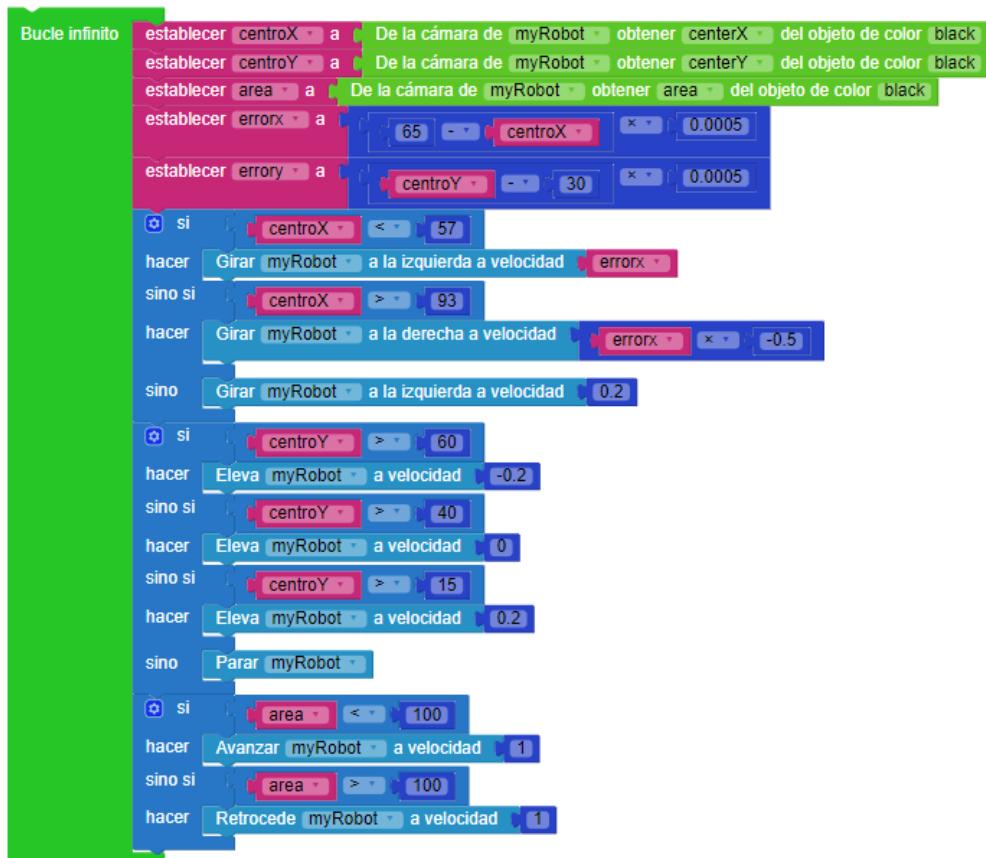


Figura 3.27: Solución en *Scratch* para el ejercicio *sigue pelota drone*

La solución de este ejercicio⁵ se ha realizado obteniendo toda la información que aporta la cámara, tanto la posición del objeto en el eje X e Y de la imagen como su área. Según los valores obtenidos se comandan distintas velocidades angulares y lineales.

3.4.5. 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*.

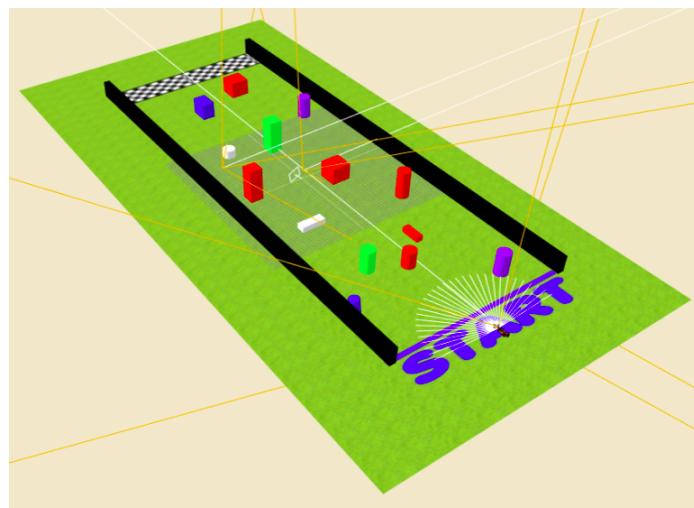


Figura 3.28: Escenario para el ejercicio atravesia bosque

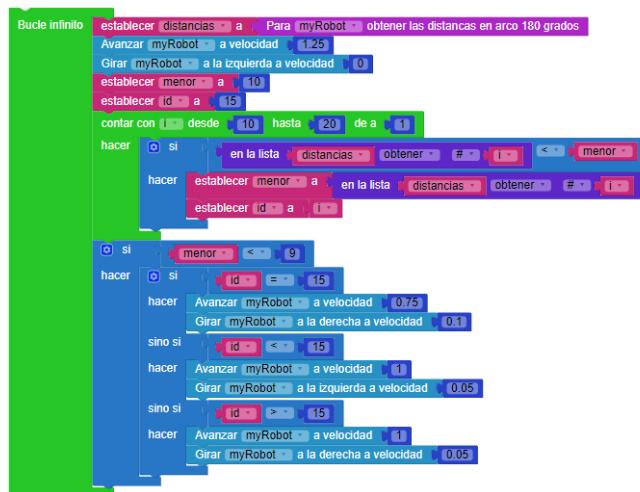


Figura 3.29: Solución en *Scratch* para el ejercicio atravesia bosque

⁵<https://youtu.be/XQrNaWxRg7U>

En esta solución⁶ se obtienen todos los valores que devuelve el sensor de ultra-sonidos y, según donde detecte el obstáculo, gira en un sentido u otro.

3.4.6. Cuadrado con drone

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

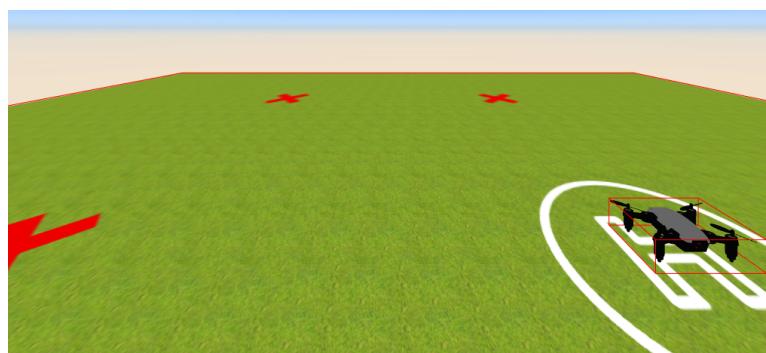


Figura 3.30: Escenario de WebSim para el ejercicio drone cuadrado

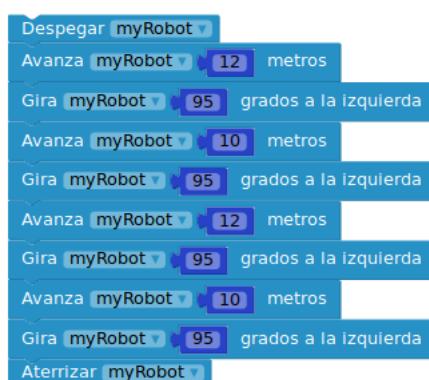


Figura 3.31: Solución en *Scratch* para el ejercicio cuadrado drone

Su solución⁷ consiste en avanzar los metros necesarios para llegar a la cruz y, cuando se ha completado el desplazamiento, girar 90 grados aproximadamente para así “dibujar” un cuadrado repitiendo este proceso.

⁶<https://youtu.be/z3n47wWHDFc>

⁷<https://www.youtube.com/watch?v=XjQNhNCKoJA>

3.5. 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.

3.5.1. 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 3.32) y añadido funcionalidad a cada botón para guardar el código de cada robot o mostrar el código en caso de tener uno guardado.

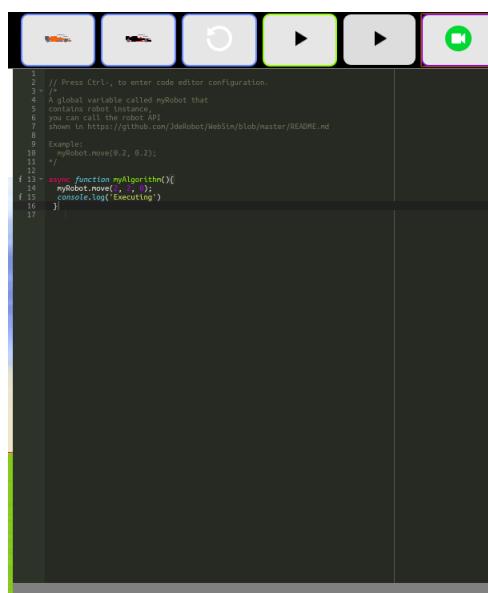


Figura 3.32: Editor de *JavaScript* para ejercicios competitivos

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 3.5*). 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 3.33.

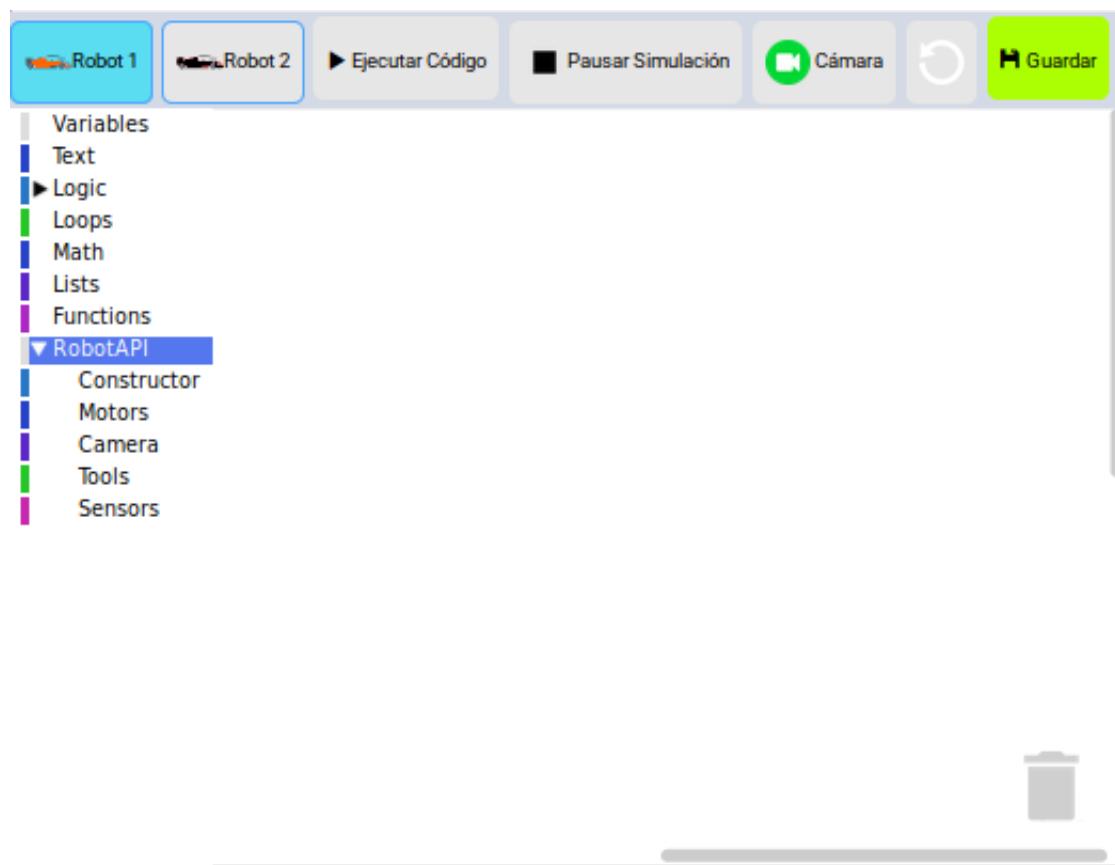


Figura 3.33: Editor de *Scratch* para ejercicios competitivos

```

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 3.5: 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 3.6: 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 3.5.1) 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;
22 }

```

Listing 3.7: Funciones que crean el objeto para ejecutar el evaluador periódicamente

3.5.2. 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 uno⁸. 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.

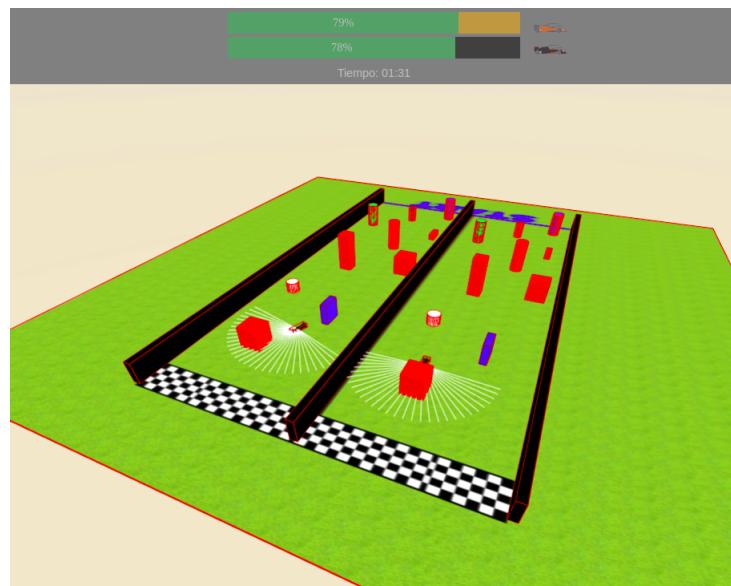


Figura 3.34: Escenario y evaluador para el ejercicio atravesia-bosque

Las funciones necesarias para el evaluador se definen a continuación:

⁸https://www.youtube.com/watch?v=v_aMvNPtncg

- 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");
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 }
```

3.5.3. 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 distancia⁹.

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 3.8: 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:{
```

⁹https://www.youtube.com/watch?v=OaA7_wsXhk8

```

15         x:robot2.getPosition().x,
16         z:robot2.getPosition().z
17     },
18     dist: 0
19 }
20 }
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 3.9: Función que realiza la funcionalidad para llenar la barra de progreso

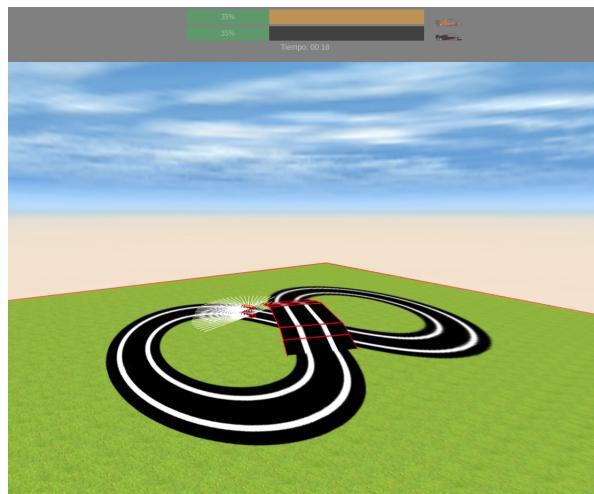


Figura 3.35: Ejercicio y evaluador sigue-líneas competitivo

3.5.4. 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 movimiento¹⁰.



Figura 3.36: Evaluador y escenario con dos robots para ejercicio gato-ratón

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 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({
```

¹⁰https://www.youtube.com/watch?v=xA9Emhdk_HQ

```

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       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*[?]) que muestra la distancia entre *drones* y el tiempo que lleva de ejecución. Se puede ver este evaluador en la figura 3.36. 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 3.10: 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);
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 3.11: 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) +
16                      Math.pow(pos2.y-pos1.y,2) +
17                      Math.pow(pos2.z-pos1.z,2));
18  line.addPointXY(x,dist+10);
19  x=x+0.5;
20  myPanel.addElement(line);
21 }
22 }
```

Listing 3.12: Código JavaScript que calcula la distancia entre *drones*, la representa e incorpora un cronómetro al *DOM*