



**GRADO EN INGENIERÍA EN SISTEMAS DE  
TELECOMUNICACIÓN**

Curso Académico 2019/2020

Trabajo Fin de Grado

**Colaboración en el desarrollo software del entorno  
web de aprendizaje de Kibotics**

Autor : Natalia Monforte Rodríguez

Tutor : Dr. José María Cañas Plaza



# Índice general

<b>Lista de figuras</b>	<b>5</b>
<b>Lista de tablas</b>	<b>7</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Robótica . . . . .	1
1.2. Tecnologías web . . . . .	4
1.2.1. Tecnologías web en el lado del cliente . . . . .	5
1.2.2. Tecnologías web en el lado del servidor . . . . .	5
1.3. Docencia robótica . . . . .	7
1.4. Motores de físicas . . . . .	9
<b>2. Objetivos</b>	<b>11</b>
2.1. Objetivos . . . . .	11
2.2. Metodología y planificación . . . . .	11
<b>3. Herramientas</b>	<b>15</b>
3.1. Lenguaje JavaScript . . . . .	15
3.2. Lenguaje HTML . . . . .	16
3.3. Lenguaje JSON . . . . .	17
3.4. Lenguaje Python . . . . .	18
3.5. Programación con Scratch . . . . .	19
3.6. Blender . . . . .	20
3.7. Simulador Websim . . . . .	21
3.7.1. A-Frame . . . . .	21

3.7.2. Sistema de físicas de <i>A-Frame</i> . . . . .	22
<b>4. Mejora de las físicas en WebSim</b>	<b>25</b>
4.1. Motor de físicas actual . . . . .	25
4.1.1. Gravedad, colisiones y fricciones materializadas por <i>CANNON</i> . . . . .	27
4.1.2. Fricción . . . . .	31
4.2. Motor de físicas complementario . . . . .	33
4.2.1. Diseño . . . . .	33
4.2.2. Nivel básico: modelo de fuerzas . . . . .	33
4.2.3. Nivel superior: controlador PD . . . . .	35
4.2.4. Timing . . . . .	46
4.2.5. Motor de físicas complementario para robots terrestres . . . . .	48
4.2.6. Motor de físicas complementario para drones . . . . .	49
<b>5. Nuevos ejercicios</b>	<b>51</b>
5.1. Sigue-líneas sofisticado . . . . .	51
5.2. Laberinto 3D para mBot . . . . .	52
5.3. Laberinto para drone . . . . .	53
5.3.1. Sin señalización . . . . .	54
5.3.2. Con señalización . . . . .	55
5.4. Fútbol competitivo . . . . .	56
<b>6. Conclusiones</b>	<b>59</b>
6.1. Valoración de los resultados . . . . .	59
6.2. Mejoras futuras . . . . .	59

# Índice de figuras

1.1. Ejemplos de robots en la actualidad . . . . .	3
4figure.caption.5	
6figure.caption.6	
1.4. Aprendizaje de programación robótica con <i>Lenobotics</i> . . . . .	7
1.5. Aprendizaje de programación robótica con <i>LEGO education</i> . . . . .	8
1.6. Interfaz de programación en <i>Kibotics</i> de un ejercicio en <i>Scratch</i> . . . . .	9
1.7. Robots soportados en la plataforma <i>Kibotics</i> . . . . .	9
3.1. Interfaz de programación con <i>Scratch</i> . . . . .	20
3.2. Interfaz de trabajo con <i>Blender</i> . . . . .	20
23figure.caption.13	
28figure.caption.18	
29figure.caption.20	
4.3. Escenarios de prueba de las colisiones de <i>A-Frame</i> . . . . .	31
34figure.caption.29	
36figure.caption.30	
37figure.caption.32	
37figure.caption.33	
38figure.caption.34	
38figure.caption.35	
39figure.caption.36	
4.11. Relación fricción - aceleración . . . . .	48
4.12. Tiempo - Velocidad Controlador PD plano horizontal . . . . .	49
4.13. Tiempo - Posición Controlador PD eje Y . . . . .	50

4.14. Tiempo - Velocidad Controlador PD eje Y . . . . .	50
5.1. Sigue-líneas sofisticado . . . . .	52
5.2. Laberinto 3D para mBot . . . . .	53
5.3. Laberinto para drone sin señalización . . . . .	54
5.4. Laberinto para drone con señalización . . . . .	55
5.5. Fútbol competitivo . . . . .	56
5.6. Evaluador del ejercicio Fútbol competitivo . . . . .	57

# Índice de cuadros

3.1.	Parámetros configurables del sistema de físicas de <i>A-Frame</i> . . . . .	24
4.1.	Resultados de las colisiones obtenidos con el escenario 1 . . . . .	30
4.2.	Resultados de las colisiones obtenidos con el escenario 2 . . . . .	30
4.3.	Resultados de las colisiones obtenidos con el escenario 3 . . . . .	30



# **Capítulo 1**

## **Introducción**

En este primer capítulo de la memoria se van a explicar los conceptos clave entorno a los cuales se ha desarrollado este Trabajo de Fin de Grado. Entender qué son la robótica y las tecnologías web y por qué son importantes es fundamental, ya que la combinación de ambos conceptos ha dado lugar a la docencia robótica.

Por otro lado, en este capítulo también se va a introducir el concepto de motor de físicas, ya que una importante parte del trabajo se ha basado en la generación de un motor de físicas complementario que permite recrear los movimientos realizados por los robots del entorno web de *Kibotics* con un mayor realismo.

### **1.1. Robótica**

La robótica es la ciencia que estudia la creación de máquinas automatizadas capaces de recrear comportamientos humanos o animales en función del software que lleven incorporados. Estas máquinas son las que se denominan robots.

Haciendo un breve repaso de la historia de los robots, cabe destacar que desde el 85 a.C. ya se empezaron a crear los primeros robots en la Antigua Grecia. En esa época la creación de robots se basaba en el intento de replicar personas por medio de máquinas. De hecho, esas máquinas ni siquiera se denominaban robots. El término robot fue acuñado en 1920 por Karel

Capek como homenaje a su obra teatral Rossum's Universal Robots, que trataba de una empresa encargada de fabricar humanos artificiales para facilitar la realización de tareas a los trabajadores de las fábricas. Así, la palabra Robot procede de Robbota, que en checo significa trabajo forzado o servidumbre<sup>1</sup>.

El primer robot del que se tiene contancia es el *Elektro*. Fue construido en 1937 y se le conoce como *el primer robot de la historia*. Este robot representaba a un humano de 2 metros de altura y 120 kg de peso. Era capaz de recrear movimientos humanos como caminar y de comunicarse utilizando hasta 700 palabras<sup>2</sup>. Posteriormente, George Charles Devol creó el primer robot industrial en 1948. Por ello, George Charles Devol es considerado el inventor de la robótica. Junto a Joseph F. Engelberger, creó la empresa Unimation, que fue la responsable de la creación de gran parte de los primeros robots industriales de la historia<sup>3</sup>.

Hoy en día, los robots están presentes prácticamente en cualquier ámbito de la vida de cualquier persona. Ya se han creado robots capaces de recrear casi cualquier actividad realizada por el ser humano o que nos facilita la realización de las mismas. Algunos de los robots más populares en la actualidad son lo siguientes. En la Figura 1.1 se ofrece una representación gráfica de ellos.

- Robots que se encargan de la limpieza del hogar (por ejemplo, Roomba).
- Robots sociales encargados de hacer compañía (por ejemplo, Pepper).
- Robots de cocina capaces (por ejemplo, Thermomix).
- Robots especialistas en labores de rescate (por ejemplo, Atlas).
- Drones (por ejemplo, Tello).
- Coches autónomos (por ejemplo, Tesla).

---

<sup>1</sup><https://revistaderobots.com/robots-y-robotica/que-es-la-robotica/>

<sup>2</sup>Ibidem

<sup>3</sup>Ibidem



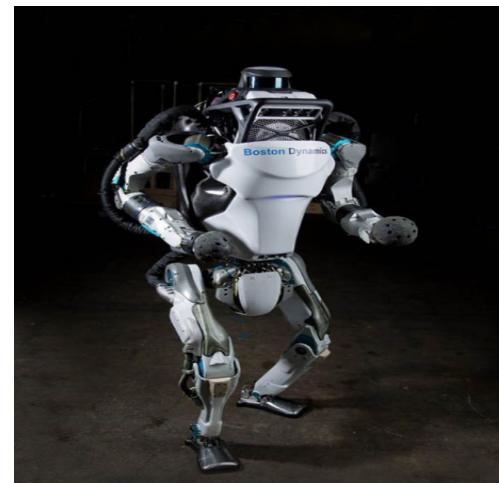
(a) Roomba



(b) Pepper



(c) Thermomix



(d) Atlas



(e) Tello



(f) Tesla

Figura 1.1: Ejemplos de robots en la actualidad

## 1.2. Tecnologías web

Las tecnologías web están en continúo desarrollo. Actualmente, existen tecnologías web tanto en el lado del cliente como en el lado del servidor. La idea de esta separación es marcar las diferentes partes de un sistema software para poder controlarlo de una forma más eficaz. Por este motivo, el frontend recoge los datos y el backend los procesa.

Por un lado, el frontend engloba todas aquellas tecnologías web del lado del cliente que se encargan de recopilar los datos. Principalmente, existen tres tecnologías de frontend: *HTML*, *CSS* y *JavaScript*. Estas tres tecnologías permiten al usuario interactuar con el servidor web, utilizando un navegador como intérprete. Por otro lado, el backend se encarga del almacenamiento de información en bases de datos, gestión de servidores y servir las vistas de las páginas web seleccionadas por el desarrollador en el lado del cliente. En el backend, el número de tecnologías es mucho más extenso. La programación backend incluye lenguajes como *PHP*, *Python*, *.NET* o *Java* y las bases de datos sobre las que se trabaja pueden ser *SQL*, *MongoDB* o *MySQL*. Todas estas tecnologías web permiten implementar comportamientos determinados de las aplicaciones web en el servidor. En la Figura 1.2 se muestra un esquema de la división de las tecnologías web entre ambos planos.

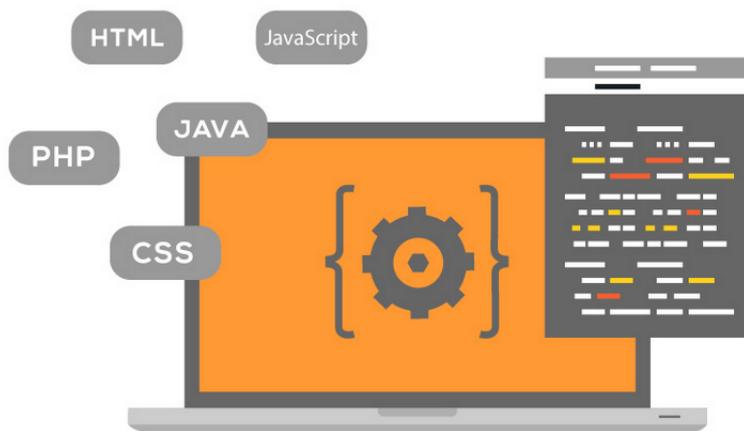


Figura 1.2: Tecnologías web en el lado del cliente y del servidor<sup>4</sup>

<sup>4</sup><https://www.ingeniovirtual.com/conceptos-basicos-sobre-tecnologias-de-desarrollo-web/>

### 1.2.1. Tecnologías web en el lado del cliente

Las tres tecnologías web del frontend que permiten la interacción entre usuario y servidor web son las siguientes:

- **HTML:** es un lenguaje de marcado que permite diferenciar los contenidos y definir la estructura de un sitio web. Permite dividir una página web en diferentes secciones: títulos, texto, imágenes, pie de página, etc. Es la base de toda página web.
- **CSS:** es un lenguaje de hojas de estilo que permite modificar la apariencia de una página web.
- **JavaScript:** es un lenguaje de programación interpretado que permite definir el comportamiento de una página web (por ejemplo, al hacer click en un enlace). Por ello, gracias a JavaScript el usuario puede interaccionar con la página web.

### 1.2.2. Tecnologías web en el lado del servidor

Uno de los lenguajes más utilizados en la programación del backend es el lenguaje *JavaScript*. *JavaScript* se creó para su uso en el frontend en un principio; sin embargo, gracias al motor *NodeJS*, este lenguaje puede ser interpretado en el lado del servidor sin necesidad de un navegador.

El código *JavaScript* del cliente y el servidor es independiente el uno del otro. Sin embargo, resulta de especial interés el hecho de que se pueda desarrollar código en un mismo lenguaje tanto en el frontend como en el backend por las facilidades y reducción de tiempos y esfuerzo que esto supone para los desarrolladores.

Se puede utilizar el mismo lenguaje en todos los contextos del desarrollo: en el cliente de escritorio con DOM, en el cliente móvil con *Cordova o React Native*, en el servidor con *Node.js* o en la base de datos con *MongoDB*. En cuanto a la tecnología empleada, las herramientas que se utilizan en el backend son principalmente: editores de código, compiladores, depuradores de código y gestores de bases de datos.

La comunicación entre cliente y servidor se realiza utilizando el protocolo *HTTP* (protocolo de transferencia de hipertexto). Este protocolo funciona mediante la emisión de una serie de peticiones y respuestas entre el cliente y el servidor usando diferentes métodos. Existen numerosos métodos *HTTP*, pero los más comunes son los siguientes<sup>5</sup>:

- **GET:** solicitud de datos de un recurso concreto.
- **PUT:** reemplazo de las representaciones actuales del recurso de la petición.
- **POST:** envío de datos a un recurso concreto, normalmente provocando el cambio de estado del servidor.
- **DELETE:** eliminación de un recurso.
- **HEAD:** solicitud de datos de un recurso concreto tal y como ocurre con el método *GET*, pero la respuesta no incluye cuerpo.

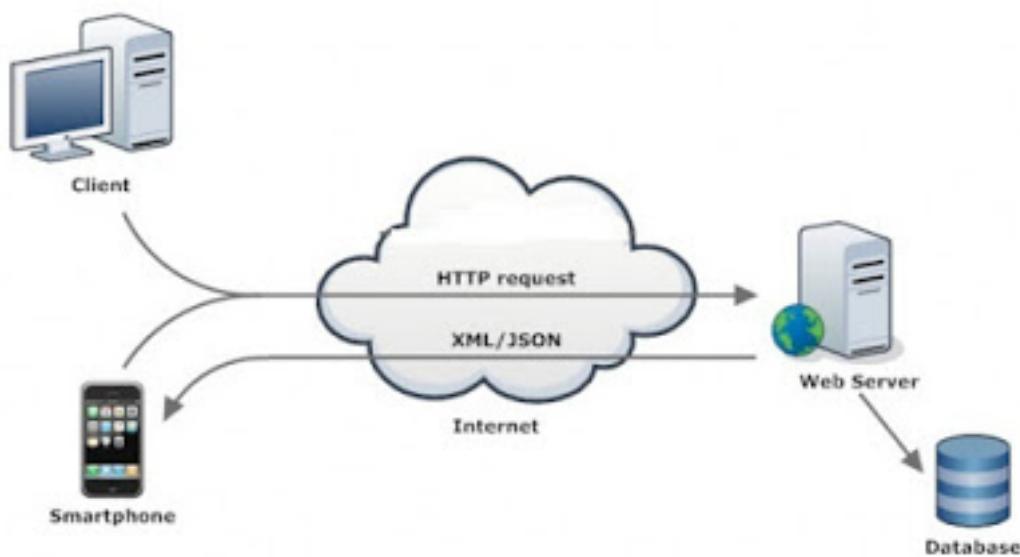


Figura 1.3: Ejemplo de una interacción *HTTP*<sup>6</sup>

<sup>5</sup><https://developer.mozilla.org/es/docs/Web/HTTP/Methods>

<sup>6</sup><http://dsanchezz.blogspot.com/2015/10/rest.html>

### 1.3. Docencia robótica

En la intersección entre la robótica y las tecnologías web se encuentra la docencia robótica. La docencia robótica tiene como objetivo el acercamiento a las tecnologías web de última generación que permiten el desarrollo software para la creación de robots desde edades tempranas.

Hoy en día, el plan de educación que se imparte en colegios e institutos no hace hincapié en la enseñanza de estas tecnologías, por lo que los niños no se encuentran especialmente motivados en el desarrollo de software robótico ya que desconocen su uso y posibilidades. En consecuencia, especialmente en los últimos años, han surgido algunas plataformas dedicadas a la docencia robótica que se encargan de impartir cursos de iniciación a la robótica desde edades muy tempranas para conseguir que los niños se sientan atraídos por esta rama desde el principio y aprendan a pensar como verdaderos programadores desde muy pequeños.

Un ejemplo de estas plataformas que se comentan es el de *Lenobotics*. *Lenobotics* es un programa que se encarga de impartir cursos de robótica en centros educativos para desarrollar las habilidades cognitivas de los niños que resultan necesarias para la programación<sup>7</sup>.



Figura 1.4: Aprendizaje de programación robótica con *Lenobotics*

<sup>7</sup><https://lenobotics.com/>

Por su parte, *LEGO education* también ofrece a los más pequeños la posibilidad de iniciarse en la robótica a través de sus kits de robótica. Estos kits permiten aprender unas primeras nociones de electrónica, robótica y programación<sup>8</sup>.



Figura 1.5: Aprendizaje de programación robótica con *LEGO education*

El presente trabajo se va a centrar en la plataforma *Kibotics*<sup>9</sup>. *Kibotics* es un entorno web para docencia en robótica y programación que permite a niños y adolescentes aprender programando. Esto quiere decir que *Kibotics* apuesta por una enseñanza puramente práctica, ya que resulta mucho más atractiva tanto la enseñanza como el aprendizaje de este modo que únicamente con clases teóricas.

*Kibotics* se basa en la utilización del simulador *WebSim* que, a su vez, está basado en la tecnología *A-Frame* para representar los mundos. Este simulador permite la creación de diferentes ejercicios para los robots que tienen soporte en la plataforma: piBot, mBot, fórmula 1 y drone Tello. Estos ejercicios podrán solucionarse tanto en *Scratch* (especialmente indicado para aquellos alumnos que no hayan programado anteriormente) como en *Python* (para aquellos niños que cuenten con nociones previas).

<sup>8</sup><https://education.lego.com/es-es>

<sup>9</sup><https://kibotics.org/>

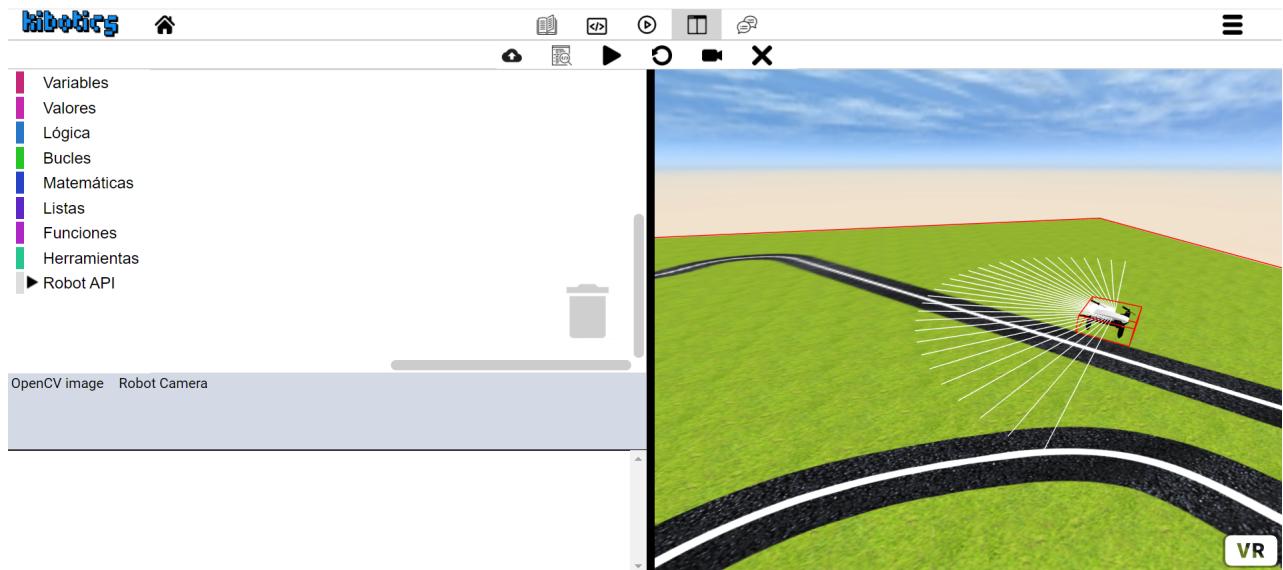


Figura 1.6: Interfaz de programación en *Kibotics* de un ejercicio en *Scratch*

Todos los robots cuentan con cámaras incorporadas en el hardware, lo que permite la creación de ejercicios que se deben solucionar mediante la utilización de la visión artificial además de los ejercicios que se puedan solucionar mediante el uso de los sensores de los robots (sensores infrarrojos, por ejemplo). La Figura 1.7 muestra los robots que soporta actualmente la plataforma.

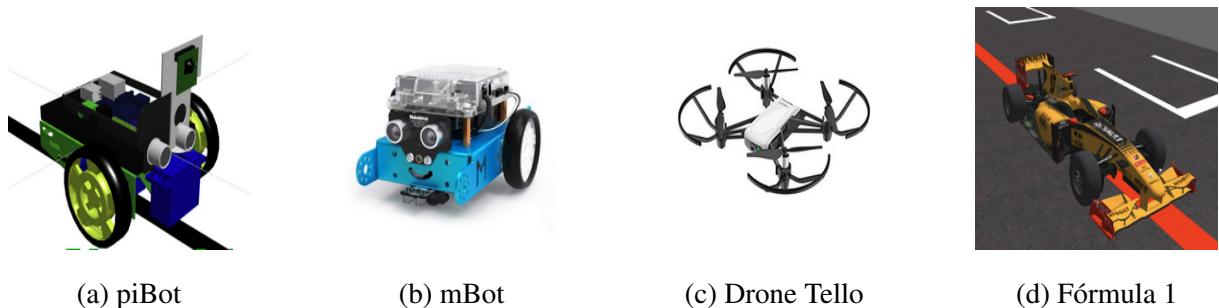


Figura 1.7: Robots soportados en la plataforma *Kibotics*

## 1.4. Motores de físicas

Un motor de físicas es un software capaz de realizar simulaciones de ciertos sistemas físicos como la dinámica del cuerpo en movimiento, la fricción y la elasticidad de una colisión.

Se emplean con mucha frecuencia en los videojuegos, para recrear con un mayor realismo el movimiento de los personajes.

Existen numerosos motores de físicas como *Box2D*, *Cocos2D*, *Ammo.js* o *CANNON*. El presente trabajo se va a centrar en este último, ya que es el que emplea *A-Frame* en la actualidad. En el capítulo 3: Herramientas se explicará con mayor detalle las peculiaridades de este motor de físicas en cuestión.

# **Capítulo 2**

## **Objetivos**

En este capítulo se explican los objetivos del presente trabajo, la metodología que se ha seguido para alcanzar dichos objetivos y la planificación que se ha llevado durante el proceso de investigación.

### **2.1. Objetivos**

El principal objetivo de este Trabajo de Fin de Grado es la mejora de las habilidades de desarrollo software mediante diversas actividades de programación.

Con este fin, se han abordado las siguientes tareas:

- Exploración del simulador *WebSim* y creación de nuevos mundos para el desarrollo de ejercicios más variados.
- Exploración de la tecnología *A-Frame* para la construcción de nuevos escenarios.
- Exploración del motor de físicas de *CANNON* para desarrollar unas físicas más realistas para el simulador *WebSim*.

### **2.2. Metodología y planificación**

Con el fin de asegurar el correcto desarrollo del Trabajo de Fin de Grado se estableció una reunión semanal con el tutor para compartir los progresos realizados durante la semana y en la

que el tutor me pudo orientar sobre dónde dirigir los esfuerzos cada semana. Paralelamente a las reuniones semanales, también se ha contado con un canal de slack en el que se encuentran todos los contribuyentes del entorno *Kibotics* en el que se han podido plantear todo tipo de dudas durante el proceso de aprendizaje.

El proceso de elaboración del Trabajo de Fin de Grado se ha dividido en cinco fases distintas:

- **FASE 0:** aprendizaje y primera toma de contacto con las tecnologías web necesarias para la elaboración del trabajo. Especialmente *A-Frame* y *JavaScript*.
- **FASE 1:** estudio del código de *Kibotics-WebSim*.
- **FASE 2:** creación de los primeros mundos utilizando las funcionalidades proporcionadas por *Blender* y *A-Frame*.
- **FASE 3:** estudio de las físicas de *A-Frame* (motor de *CANNON*) y elaboración de un motor de físicas complementario para el simulador *WebSim*.
- **FASE 4:** creación de nuevos ejercicios para incluir en la plataforma.

También se ha elaborado un blog en el que se han ido compartiendo los resultados y el trabajo que se ha realizado cada semana. El blog se ha implementado gracias al dominio gratuito que ofrece Github para crear un blog<sup>1</sup>. En el README de mi Github se ha incluido un enlace pinchable para acceder a dicho blog<sup>2</sup>.

Para integrar el código de las mejoras o aportaciones realizadas al código fuente de *Kibotics*, cabe destacar que se ha utilizado el sistema que ofrece Github para integrar código mediante la creación de nuevas ramas y parches. Para que los desarrolladores pudiesen añadir las nuevas funcionalidades al código fuente de *Kibotics*, se creaba una nueva rama actualizada con los últimos cambios de la rama principal. Sobre esta rama se desarrollaba la solución a cada incidencia. Una vez incluidos los cambios se explicaban en un comentario o commit y se subían a la nueva rama creada del repositorio de *Kibotics*. El siguiente paso consistía en solicitar la fusión de los cambios de esta rama con la rama principal, abriendo peticiones pull request o parches. Tras la

<sup>1</sup><https://roboticslaburjc.github.io/2019-tfg-natalia-monforte/>

<sup>2</sup><https://github.com/RoboticsLabURJC/2019-tfg-natalia-monforte>

solicitud de la fusión o parche los desarrolladores que cuentan con más experiencia verifican que los cambios son correctos y, si es así, integran los cambios a la rama maestra oficial, dando por resuelta la incidencia. Los comandos necesarios para realizar la integración del código a la rama creada son los siguientes:

```
git checkout -b issue-XXX  
git add -ruta-del-fichero-a-añadir  
git commit -m "Comentario para el commit"  
git push -u origin issue-XXX
```



# Capítulo 3

## Herramientas

En este capítulo se explica con un mayor grado de detalle qué herramientas han sido necesarias para el desarrollo del trabajo. Principalmente, se han utilizado los lenguajes de programación *JavaScript*, *HTML*, *JSON*, *Python* y *Scratch*. Por otro lado, se han utilizado aplicaciones como *Blender* para el modelado de objetos 3D y el simulador *WebSim* para la recreación de los mundos tridimensionales.

### 3.1. Lenguaje JavaScript

*JavaScript* es un lenguaje de programación interpretado de alto nivel que se encuentra bajo el estándar *ECMAScript*<sup>1</sup>. Este lenguaje es comúnmente conocido por su uso en los scripts de las páginas web. No obstante, dada su orientación a objetos y a ser un lenguaje de programación basada en prototipos y de un solo hilo, es usado en otros muchos entornos externos de la página web: *Node.js*, *Apache CouchDB* o *Adobe Acrobat*<sup>2</sup>.

La sintaxis es similar a la utilizada en *Java* y *C++*. De esta manera, se facilita el aprendizaje del lenguaje ya que está basado en conceptos ya conocidos por el programador. Las estructuras del lenguaje, tales como sentencias condicionales (if y switch) y bucles (while y for) funcionan de una manera muy similar a como lo hacen en los otros lenguajes de programación<sup>3</sup>.

---

<sup>1</sup>Especificación de lenguaje de programación en el que se definen tipos dinámicos y soporte de programación orientada a objetos

<sup>2</sup><https://developer.mozilla.org/es/docs/Web/JavaScript>

<sup>3</sup>Ibidem

Las siguientes características son las principales de *ECMAScript*:

- Lenguaje escruturado similar a la estructura utilizada en *Java* y *C++*.
- *ECMAScript 2015* añadió la palabra clave *let*, que permite que el alcance de la variable se corresponda con el bloque en el que esta se haya definido (*block scoping*).
- Tipado débil, es decir, el tipo de datos se asocia al valor de la variable en un preciso momento.
- El lenguaje está formado por objetos.
- Lenguaje interpretado, es decir, se compila justo-a-tiempo. No es necesario disponer de un compilador adicional, ya que cada navegador incluye un intérprete que se encarga de ejecutar el código.

El proyecto está enteramente programado en *JavaScript*, ya que se trata de una aplicación web que corre en el lado del cliente. Por ello, este lenguaje es el que mejores prestaciones y más necesidades cubre durante el desarrollo de la aplicación.

## 3.2. Lenguaje HTML

*HTML* es un lenguaje de marcado que define la estructura de una página web. *HTML* ofrece una serie de elementos que permiten clasificar diferentes partes de una misma página web en una misma clase para otorgarles una misma apariencia. Además, *HTML* permite cambiar el estilo de las palabras (por ejemplo, a cursiva, a negrita, agrandar o reducir el tamaño de letra, etc)<sup>4</sup>.

Las partes principales del elemento *HTML* son las siguientes:

- **Etiqueta de apertura:** se trata del nombre del elemento y se incluye entre paréntesis angulares (*<*). Indica el inicio del elemento.
- **Etiqueta de cierre:** similar a la etiqueta de apertura salvo que incluye, además, una barra de cierre (*/*) precediendo al nombre de la etiqueta. Indica el fin del elemento.

---

<sup>4</sup>[https://developer.mozilla.org/es/docs/Learn/Getting\\_started\\_with\\_the\\_web/HTML\\_basics](https://developer.mozilla.org/es/docs/Learn/Getting_started_with_the_web/HTML_basics)

- **Contenido:** todo aquello que se incluye entre la etiqueta de apertura y la de cierre.
- **Elemento:** es el conjunto formado por las etiquetas de apertura y cierre y el contenido del elemento.

Además, cada elemento puede incluir uno o más atributos que permiten añadir mas información acerca de ese elemento. Por ejemplo, añadir información sobre el estilo del elemento. A continuación, se incluye un fragmento de código *HTML* a modo de ejemplo.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mi código de prueba</title>
  </head>
  <body>
    <p class="editor-note">Esto es<strong>una prueba</strong></p>
  </body>
</html>
```

*Kibotics* emplea *HTML* para crear las plantillas de las diferentes páginas que sirve la aplicación web.

### 3.3. Lenguaje JSON

El acrónimo *JSON* significa JavaScript Object Notation (Notación de Objetos de JavaScript). Se trata de un formato para el intercambio de datos. Es un lenguaje sencillo para la escritura y lectura humana y, al mismo tiempo, resulta fácil para las máquinas interpretarlo y procesarlo. *JSON* está constituido por dos estructuras: una colección de pares nombre - valor y una lista ordenada de valores. Dado que estas convenciones son conocidas por otros lenguajes como *C*, *C++*, *C*, *Java*, *JavaScript*, *Perl*, *Python*, se trata de un lenguaje ideal para el intercambio de datos<sup>5</sup>.

---

<sup>5</sup><https://www.json.org/json-es.html>

En *JSON*, estas estructuras se presentan de la siguiente forma:

- **Objeto:** conjunto desordenado de pares nombre - valor. Un objeto va encerrado entre llaves ( ). La sintaxis es la siguiente:

```
objeto {  
    nombre1: valor1,  
    nombre2: valor2,  
    ...  
}
```

- **Array:** colección de valores. Van encerrados entre corchetes [ ]. Un valor puede ser una cadena de caracteres con comillas dobles, un número, true, false o null, un objeto o un array. Estas estructuras pueden anidarse.

La aplicación de *Kibotics* emplea *JSON* para crear los ficheros de configuración de los diferentes escenarios utilizados en los ejercicios que ofrece la aplicación. Mediante un parser se recopila la información necesaria del fichero de configuración *JSON* para poder construir el mundo.

## 3.4. Lenguaje Python

*Python* es un lenguaje de programación muy popular hoy en día. Fue creado por Guido van Rossum y lanzado en 1991. La popularidad de este lenguaje reside en su fácil comprensión para el usuario y en las facilidades que este ofrece a la programación. Las principales características son las siguientes<sup>6</sup>:

- Python es compatible con diferentes sistemas operativos (Windows, Mac, Linux, Raspberry Pi, entre otros).
- Su sintaxis se asemeja en gran medida al habla inglesa, lo que facilita su comprensión.

<sup>6</sup>[https://www.w3schools.com/python/python\\_intro.asp](https://www.w3schools.com/python/python_intro.asp)

- Su sintaxis permite a los programadores escribir códigos con menos líneas que con otro lenguaje, lo que facilita la labor de depurado.
- Python es un lenguaje interpretado, lo que significa que el código puede ser ejecutado tan pronto como se escribe. No es necesario disponer de un compilador externo.
- Este lenguaje puede ser tratado de una manera procedural, una manera orientada a objetos o una manera funcional.
- Se basa en la indentación. Utiliza los espacios en blanco para determinar el alcance de las estructuras.

Valorando todas las ventajas que ofrece *Python*, se ha seleccionado este lenguaje como uno de los que ofrece la plataforma *Kibotics* para que los usuarios puedan dar solución a los ejercicios que se plantean.

## 3.5. Programación con Scratch

Scratch es un lenguaje de programación visual que fue desarrollado por el Grupo Lifelong Kindergarten del MIT Media Lab. Hoy en día, se utiliza frecuentemente en la educación de niños, adolescentes y adultos ya que permite el aprendizaje de la programación sin tener un amplio conocimiento del código.

*Scratch* ofrece al usuario la posibilidad de programar construyendo una secuencia de código a partir de diversos bloques de acciones. El programador es capaz de construir la secuencia de código con rapidez y facilidad, ya que cada bloque incluye una secuencia de texto que explica la función que desempeña. Por ello, la secuencia de código finalmente podrá ser leída e interpretada como si de un texto se tratase.

Este es el segundo lenguaje ofrecido por *Kibotics* para dar solución a los ejercicios. Gracias a la interfaz gráfica de la que dispone, Scratch es sin duda una de las mejores alternativas para aquellas personas que no disponen de nociones previas de programación.

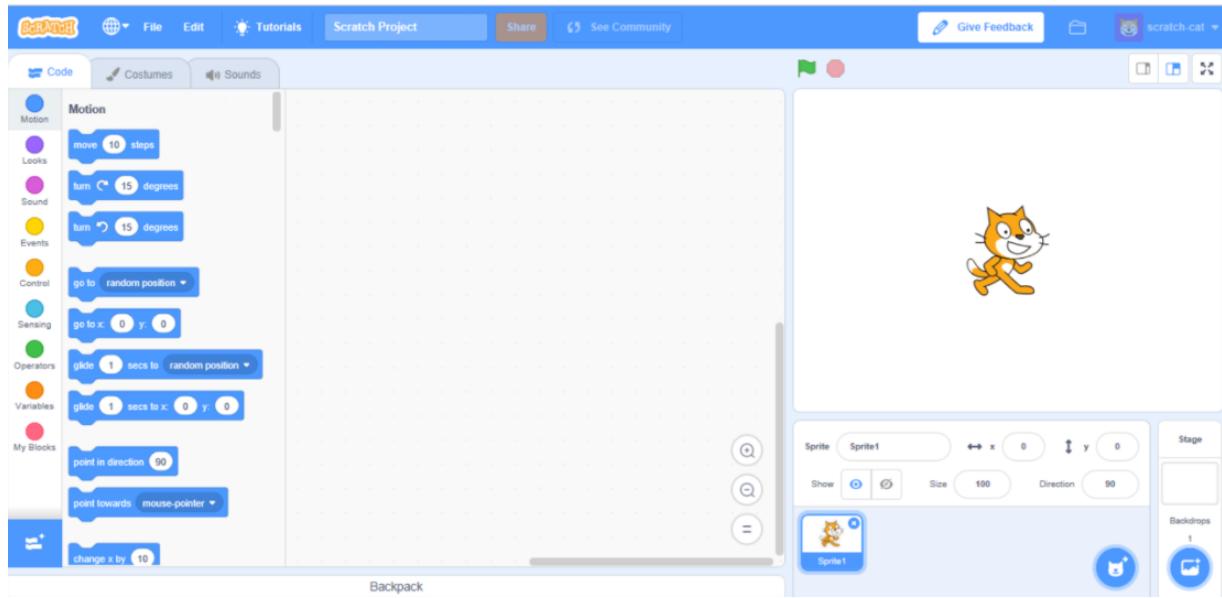


Figura 3.1: Interfaz de programación con *Scratch*

## 3.6. Blender

*Blender* es un programa informático multi plataforma, es decir, compatible para distintos sistemas operativos como Windows o Linux. Las funciones principales que se pueden realizar con *Blender* son el modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales. También se pueden realizar actividades relacionadas con la composición de vídeo.

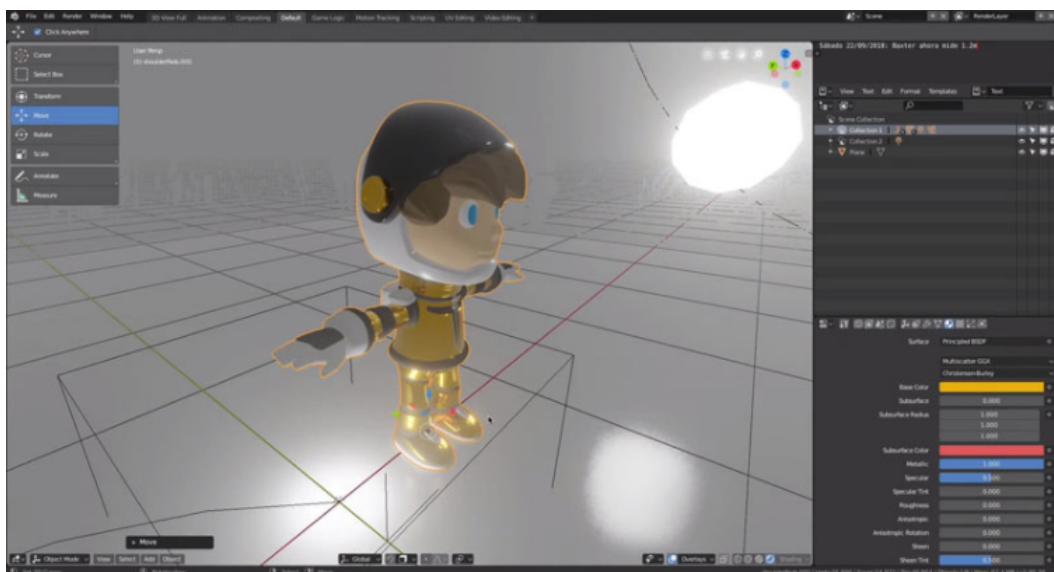


Figura 3.2: Interfaz de trabajo con *Blender*

Durante el presente trabajo se ha utilizado esta herramienta para modificar la rotación y apariencia de los robots de los que dispone la plataforma *Kibotics*. Además, *Blender* permite exportar los modelos en formato glTF (GL Transmission Format). glTF es un formato de archivo basado en el estandar *JSON*. Pemite la compresión de escenas y modelos 3D para minimizar el tiempo de ejecución de los programas en los que posteriormente se utilicen.

## 3.7. Simulador Websim

*Websim* es un simulador diseñado para el aprendizaje de conceptos básicos de programación de robots especialmente para niños. El simulador permite que los usuarios puedan programar fácilmente los movimientos de los robots, ya que simplemente tienen que acceder a la información que recogen sus sensores y enviar las órdenes precisas a los actuadores del robot. Estas órdenes se deberán programar, en *Python o Scratch*, dentro del editor que incorpora la interfaz de *Websim*.

El simulador está diseñado basándose en el uso del entorno *A-Frame*. A su vez, *A-Frame* se sirve del motor de físicas de *CANNON* para materializar los movimientos de los cuerpos dinámicos en la escena. A continuación, se explican con mayor detalle ambas tecnologías.

### 3.7.1. A-Frame

*A-Frame* es un marco web para crear escenas de realidad virtual. Sus principales características son las siguientes<sup>7</sup>:

- **Permite un uso sencillo de la realidad virtual:** para usar *A-Frame* basta con colocar las etiquetas `<script>`y `<a-scene>`. *A-Frame* se encarga del modelado 3D y la realidad virtual, no es necesaria la instalación de ningún paquete externo.
- **HTML declarativo:** *A-Frame* está basado en *HTML*, por ello es fácil y accesible para cualquier programador, puesto que *HTML* es un lenguaje ampliamente conocido.

---

<sup>7</sup><https://aframe.io/docs/1.0.0/introduction/features>

- **Arquitectura de componente de entidad:** *A-Frame* sigue el patrón ECS (entidad-componente-sistema). Se trata de un patrón de desarrollo de juegos basado en el principio de composición sobre herencia. De esta manera, se otorga una mayor flexibilidad en la definición de entidades ya que cada objeto de la escena se corresponde con una entidad y cada entidad, a su vez, está compuesta por uno o más componentes que contienen datos y estado de la entidad. Por tanto, una entidad puede verse modificada en tiempo de ejecución si alguno de los componentes que agrega modifica sus datos.
- **Multiplataforma:** *A-Frame* es compatible con plataformas tan variadas como *Vive*, *Rift*, *Windows Mixed Reality*, *Daydream*, *GearVR* y *Cardboard*.
- **Rendimiento:** las actualizaciones de *A-Frame* se realizan en la memoria y con poco gasto energético. *A-Frame* se encuentra optimizado para *WebVR*.
- **Inspector visual:** *A-Frame* cuenta con un inspector visual integrado. Este se despliega al presionar la combinación de teclas `ctrl + alt + i`. El inspector permite detectar el origen de problemas o desarrollar una mejor distribución de la escena con menos esfuerzo.
- **Componentes:** *A-Frame* cuenta con una gran cantidad de componentes con los que trabajar. Esta amplia variedad va desde componentes geométricos básicos o materiales hasta componentes como la teletransportación, la realidad aumentada o componentes personalizados por el usuario.
- **Contribuciones con otras empresas:** *A-Frame* ha sido utilizado por empresas como Google, Disney, Samsung, Toyota o CERN, entre otras. Además, algunas de ellas, como Google, Microsoft, Oculus y Samsung han llegado a realizar contribuciones.

### 3.7.2. Sistema de físicas de *A-Frame*

Las físicas de *A-Frame* soportan dos motores de físicas: *Ammo Driver* y *CANNON*. Actualmente, el motor que está en uso es el de *CANNON*<sup>9</sup>. No obstante, ya ha sido añadido el soporte de *Ammo Driver* al sistema de físicas, ya que se preveé que *CANNON* acabe quedando obsoleto

<sup>8</sup><https://aframe.io/>

<sup>9</sup><https://github.com/donmccurdy/aframe-physics-system>

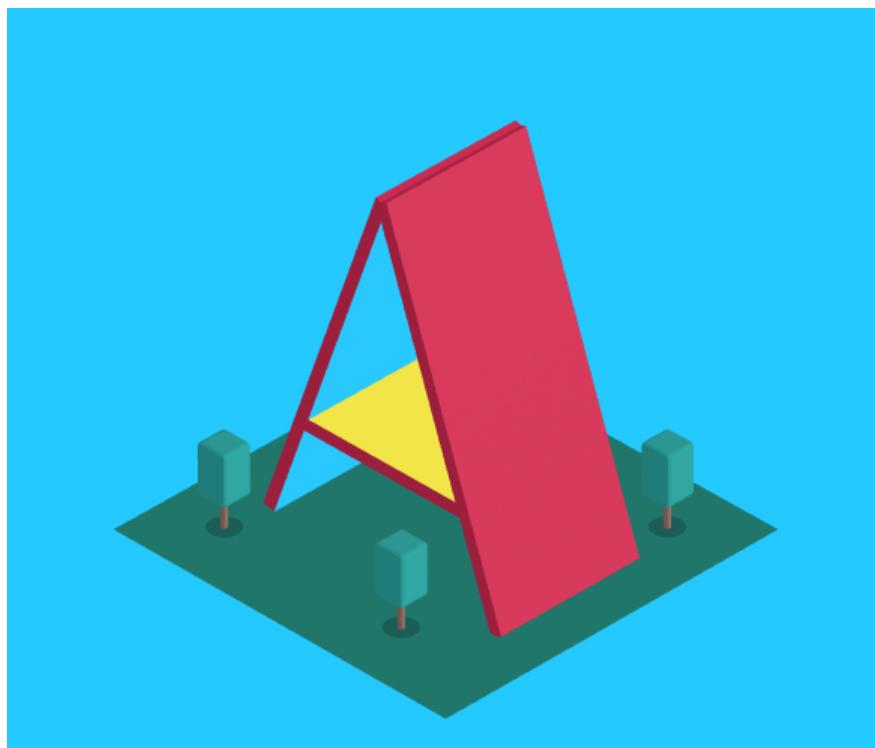


Figura 3.3: Ejemplo de construcción 3D con *A-Frame*<sup>8</sup>

con el paso del tiempo.

Para la instalación del sistema de físicas de *A-Frame* basta con incluir el siguiente script en el código *HTML* de la aplicación:

```
<script src="//cdn.rawgit.com/donmccurdy/aframe-physics-system/v4.0.1/dist/aframe-physics-system.min.js"></script>
```

El sistema de físicas de *A-Frame* cuenta con dos tipos de cuerpos: dinámicos y estáticos.

- **Cuerpo dinámico:** aquellos objetos de la escena que presentan libertad de movimiento. Estos objetos se ven afectados por la gravedad, la fricción y las colisiones.
- **Cuerpo estático:** aquellos objetos de la escena que no necesitan modificar su posición en la misma. Son cuerpos fijos y sin animaciones. Otros cuerpos dinámicos podrán colisionar con un cuerpo estático, pero el cuerpo estático no verá modificada su posición tras la colisión.

El sistema de físicas ofrece la posibilidad de añadir una malla de colisión a un objeto de la escena. Existen mallas de colisión de diferentes formas, por lo que se debe seleccionar aquella que se ajuste mejor al objeto en cuestión. Se puede elegir entre el ajuste automático, una caja, un cilindro, una esfera, un cuerpo convexo o una primitiva (plano, cilindro o esfera seleccionadas automáticamente en función de la primitiva *A-Frame* correspondiente).

Cada escena de *A-Frame* admite una serie de parámetros a los que se les puede modificar el valor para ajustar el mundo a las características deseadas. Si no se especifica el valor que se desea de un parámetro, este tomará el valor por defecto. Algunos de los parámetros que admite una escena son *debug*, que cuando está a true muestra las mallas de colisión de los objetos de la escena o *gravity*, *friction* y *restitution*, que se corresponden con la gravedad, fricción y coeficiente de restitución, respectivamente, del mundo simulado.

Atributo	Valor por defecto
debug	true
gravity	-9.8
friction	0.01
restitution	0.3

Cuadro 3.1: Parámetros configurables del sistema de físicas de *A-Frame*

# Capítulo 4

## Mejora de las físicas en *WebSim*

### 4.1. Motor de físicas actual

*A-Frame* utiliza el motor *CANNON*<sup>1</sup> para materializar las físicas. *CANNON* surgió como consecuencia de la necesidad de disponer de un motor de físicas en la web y presenta importantes semejanzas respecto a otros conocidos motores: *three.js* o *ammo.js*. Su ventaja es que su código se encuentra disponible enteramente en red<sup>2</sup>, escrito en lenguaje *JavaScript* y que su tamaño de archivo es considerablemente más pequeño que el de otros motores de físicas<sup>3</sup>.

Hasta el momento, *WebSim* contaba con un motor de físicas completo. El movimiento de los robots se recreaba mediante la actualización de la posición cada 50 ms en función de la velocidad y la rotación consignada por el usuario. No entraban en juego velocidades, aceleraciones ni fuerzas. Además, el hecho de actualizar la posición del robot constantemente hacía sobreescribir los cambios que introducía *CANNON*, por lo que las físicas no incluían el efecto de la gravedad, la fricción ni el coeficiente de restitución.

La actualización de la posición de los robots en función de la velocidad y rotación comandadas se realizaba mediante la función *updatePosition*.

---

<sup>1</sup><https://github.com/n5ro/aframe-physics-system>

<sup>2</sup><http://schteppe.github.io/cannon.js/docs/>

<sup>3</sup><http://schteppe.github.io/cannon.js/>

```
updatePosition(rotation, velocity, robotPos) {  
    if(simEnabled) {  
        let x = velocity.x / 10 * Math.cos(rotation.y * Math.PI / 180);  
        let z = velocity.x / 10 * Math.sin(-rotation.y * Math.PI / 180);  
        let y = (velocity.y / 10);  
        robotPos.x += x;  
        robotPos.z += z;  
        robotPos.y += y;  
    }  
    return robotPos;  
}
```

La motivación de crear un motor de físicas complementario es que las físicas implementadas no recreaban un movimiento realista y no eran suficientes para lo que se tiene disponible en un robot real. Por ejemplo, los robots reales terrestres tienen que superar la fuerza de rozamiento para poder subir una rampa o los drones deben ser capaces de ejercer la fuerza necesaria para superar la gravedad y emprender el vuelo. Con un motor de físicas realistas, las soluciones implementadas en el simulador serán igualmente válidas para un robot físico y, además, permite la creación de ejercicios mucho más diversos como una pista de hielo (en los que la fricción es extremadamente baja), arena (en los que la fricción es mucho más elevada) o escenarios multi-nivel con rampas (en los que los robots deberán ejercer fuerzas más grandes que en la superficie plana para poder ascender por la rampa).

Las implicaciones que tiene la materialización de la gravedad y de la fricción son las siguientes:

- **Materrialización de la gravedad:** los drones son capaces de volar en un mundo que materialice una gravedad de -9.8. Hasta el momento, los ejercicios de drones se configuraban con gravedad 0 para permitir volar al cuerpo.
- **Materialización de la fricción:** el desplazamiento de los robots no se realiza por imposición de una posición concreta, sino por la aplicación de la fuerza necesaria para alcanzar la velocidad objetivo. Al tener en cuenta la fricción, un robot deberá ejercer más fuerza

sobre superficies con altas fricciones y menos fuerza en superficies con fricciones más pequeñas.

#### 4.1.1. Gravedad, colisiones y fricciones materializadas por CANNON

##### Gravedad

Dado que *WebSim* se basa en la tecnología de *A-Frame*, la gravedad es un parámetro configurable dentro de una escena. Los ficheros de configuración de los escenarios de los ejercicios en formato *JSON* incluyen al principio del código las siguientes líneas que permiten seleccionar el valor deseado para la gravedad.

```
"scene": {  
    "gravity": -9.8  
}
```

Como se ha mencionado anteriormente, previamente al cambio introducido en las físicas, los ficheros de configuración de los ejercicios soportados en la plataforma tenían definida una gravedad con valor 0. Esta configuración era necesaria para conseguir hacer volar a los drones, puesto que con una gravedad de -9.8 cualquier cuerpo sólido de la escena caía hacia abajo como consecuencia de la atracción de la gravedad, por lo que no era posible hacer volar a los robots. Actualmente, todos los ejercicios están simulados con un valor de gravedad de -9.8.

##### Colisiones

Cualquier cuerpo sólido puede colisionar con otro cuerpo incluído en la escena. Las colisiones pueden tener naturaleza elástica o inelástica dependiendo del valor del coeficiente de restitución de los objetos. El coeficiente de restitución es la media de la conservación de la energía cinética cuando se produce un choque entre partículas. Cuando su valor es 1 el choque es perfectamente elástico y cuando es 0, es perfectamente inelástico.

$$\text{Coeficiente de restitución} = \frac{\text{Velocidad relativa tras la colisión}}{\text{Velocidad relativa antes de la colisión}} \quad (4.1)$$

*A-Frame* también ofrece la posibilidad de parametrizar el coeficiente de restitución. Este parámetro se puede configurar, al igual que la gravedad, al principio del fichero de configuración de los ejercicios incluyendo el siguiente código.

```
"scene": {  
  "physics": "restitution: 0.5"  
}
```

### Colisiones elásticas

Se dice que una colisión es elástica cuando, tras el choque, se conserva toda la energía cinética. Esta se transfiere por completo desde el objeto que colisiona hasta el objeto que ha sido colisionado. En la realidad, en todo choque parte de la energía se disipa en calor, por lo que este tipo de colisiones es considerada ideal. Visualmente, el efecto que tiene es que el objeto que colisiona se queda parado y el objeto colisionado comienza a moverse a la velocidad que se movía el objeto que colisionó con él. La Figura 4.1 muestra un ejemplo de colisión elástica.

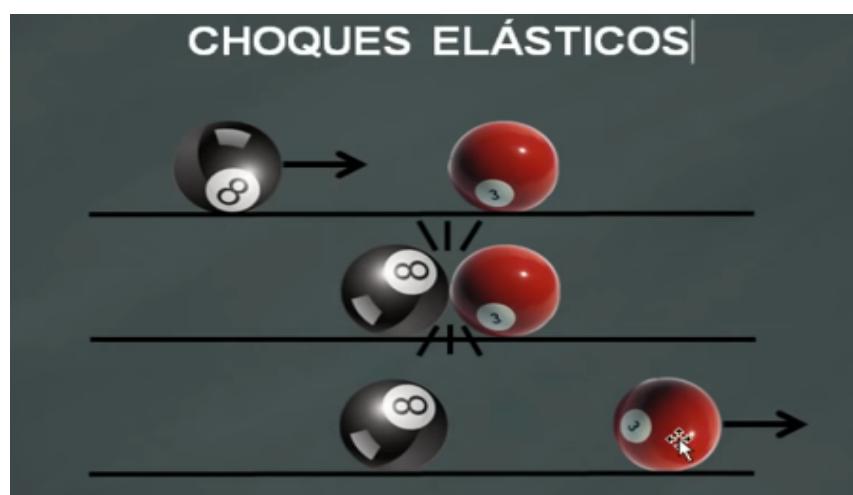


Figura 4.1: Colisión elástica<sup>4</sup>

<sup>4</sup><https://www.youtube.com/watch?v=b9iOIr5DYj8>

## Colisiones inelásticas

Por otro lado, cuando se produce una colisión inelástica el objeto que colisiona continúa teniendo parte de la energía cinética, otra parte se transfiere al objeto que ha sido colisionado y la parte restante se disipa en forma de calor. En este tipo de choques el efecto visual es que tanto el objeto que colisiona como el objeto colisionado avanzan a la misma velocidad tras el choque. La Figura 4.2 muestra un ejemplo de colisión inelástica.

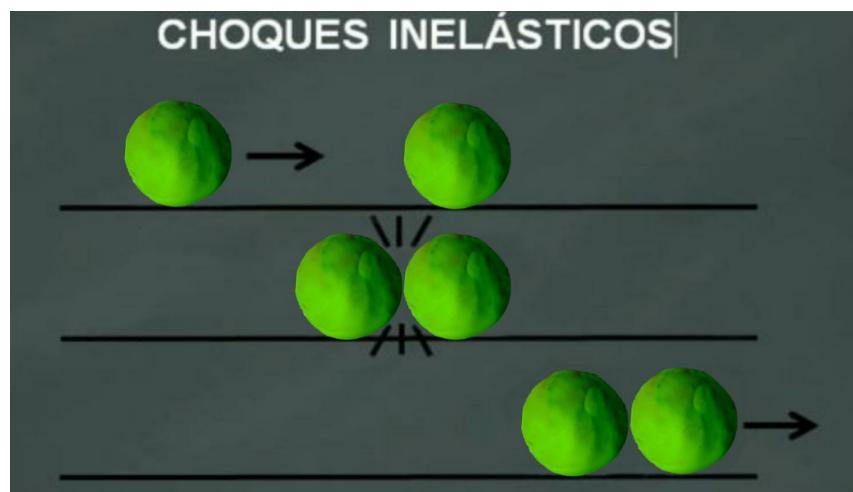


Figura 4.2: Colisión inelástica<sup>5</sup>

## Pruebas de las colisiones

Se han realizado pruebas de colisiones en tres escenarios diferentes variando el valor del coeficiente de restitución y la masa de los objetos. Los escenarios utilizados han sido los siguientes:

- **Escenario 1:** dos pelotas cayendo por dos rampas.
- **Escenario 2:** una pelota fija en el suelo y otra cayendo por una rampa.
- **Escenario 3:** una pelota cae por una rampa y colisiona con una pared.

Los resultados obtenidos durante las pruebas se detallan en los Cuadros 4.1, 4.2 y 4.3. En este link<sup>6</sup> se ofrece un vídeo con los resultados de las pruebas realizadas en el tercer escenario.

<sup>5</sup>Elaboración propia.

<sup>6</sup><https://youtu.be/T214FNFxehs>

## 4.1. MOTOR DE FÍSICAS ACTUALIZADO

### PÁTULO 4. MEJORA DE LAS FÍSICAS EN WEBSIM

---

	<b>Misma masa</b>
<b>Restitucion = 0</b>	No hay rebote
<b>Restitucion = 0.4</b>	Sí hay rebote
<b>Restitucion = 1</b>	La pelota rebota al entrar en contacto con cualquier superficie de la escena

	<b>Diferente masa</b>
<b>Restitucion = 0</b>	Las dos pelotas avanzan pegadas en la dirección de la de mayor masa
<b>Restitucion = 0.4</b>	La pelota más pesada hace cambiar la dirección del movimiento de la más ligera, que se desplaza a mayor velocidad
<b>Restitucion = 1</b>	Las dos pelotas avanzan separadas en la dirección de la de mayor masa. La pelota de menor masa coge mayor velocidad

Cuadro 4.1: Resultados de las colisiones obtenidos con el escenario 1

	<b>Misma masa</b>
<b>Restitucion = 0</b>	Ambas pelotas avanzan pegadas a la misma velocidad
<b>Restitucion = 0.4</b>	La pelota que permanecía en el suelo se mueve más rápido que la que cayó por la rampa. No avanzan pegadas
<b>Restitucion = 1</b>	La pelota rebota al entrar en contacto con cualquier superficie de la escena

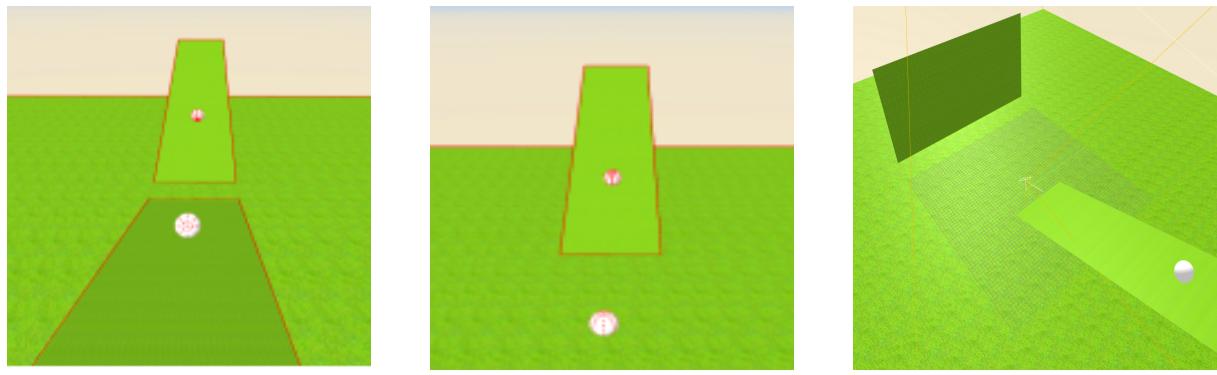
	<b>Diferente masa</b>
<b>Restitucion = 0</b>	- Masa pelota rampa ¡Masa pelota suelo: ambas pelotas se quedan juntas y paradas - Masa pelota rampa ¿Masa pelota suelo: ambas pelotas avanzan hacia adelante juntas y a la misma velocidad
<b>Restitucion = 0.4</b>	- Masa pelota rampa ¡Masa pelota suelo: la pelota que ha caído por la rampa rebota hacia arriba - Masa pelota rampa ¿Masa pelota suelo: la pelota que estaba en reposo avanza con más velocidad que la que cayó por la rampa
<b>Restitucion = 1</b>	La pelota rebota al entrar en contacto con cualquier superficie de la escena

Cuadro 4.2: Resultados de las colisiones obtenidos con el escenario 2

	<b>Misma masa</b>
<b>Restitucion = 0</b>	Permanece junto a la pared, sin rebote
<b>Restitucion = 0.4</b>	Sí hay rebote
<b>Restitucion = 1</b>	El rebote es muy elevado y vuelve a subir la rampa prácticamente a la misma velocidad que la bajó

	<b>Diferente masa</b>
<b>Restitucion = 0</b>	Permanece junto a la pared, sin rebote
<b>Restitucion = 0.4</b>	Cuanto más pequeña es la masa, más grande es el rebote
<b>Restitucion = 1</b>	Cuanto más pequeña es la masa, más grande es el rebote

Cuadro 4.3: Resultados de las colisiones obtenidos con el escenario 3



(a) Escenario 1

(b) Escenario 2

(c) Escenario 3

Figura 4.3: Escenarios de prueba de las colisiones de *A-Frame*

#### 4.1.2. Fricción

Cualquier cuerpo dinámico de una escena de *A-Frame* presenta una fuerza de rozamiento que se opone al movimiento. Existen dos tipos de rozamiento: el rozamiento estático y el rozamiento dinámico. *A-Frame* incluye tres parámetros diferentes que permiten variar el rozamiento de una superficie. No obstante, no se puede seleccionar un valor concreto para la fuerza de rozamiento estática y dinámica, ya que el atributo *friction* hace variar los dos tipos de rozamiento en conjunto. Por tanto, se puede incrementar o disminuir tanto el rozamiento estático como el dinámico, pero no se pueden parametrizar con un valor concreto.

Los ficheros de configuración de los escenarios de los ejercicios en formato *JSON* incluyen al principio del código las siguientes líneas que permiten seleccionar el valor deseado para los atributos *friction* (rozamiento estático y dinámico) y *linear-damping* y *angular-damping* (rozamiento dinámico).

```
"scene": {
    "physics": "friction: 0.5"
}
attr": {
    "static-body": {
        "linearDamping":1.2,
        "angularDamping":1.2
    }
}
```

### Rozamiento estático

Dos superficies rígidas en reposo no se desplazan una respecto a la otra siempre y cuando la fuerza paralela al plano tangente sea suficientemente pequeña. Cuando el coeficiente de rozamiento estático de una superficie es excesivamente pequeño, los objetos que se encuentran sobre esa superficie no pueden permanecer en reposo. Una forma de calcular el coeficiente de rozamiento estático es hacer variar la inclinación de una rampa. Cuando se alcanza un ángulo de inclinación con el cual el cuerpo comienza a descender, se dice que se ha llegado al ángulo crítico. A partir del ángulo crítico se puede obtener el coeficiente de rozamiento estático gracias a la siguiente igualdad:

$$\tan(\text{ángulo crítico}) = \text{coeficiente de rozamiento estático} \quad (4.2)$$

### Rozamiento dinámico

Cuando dos superficies están en contacto, el movimiento de una respecto a la otra genera fuerzas tangenciales llamadas fuerzas de fricción o rozamiento, las cuales tienen sentido opuesto al movimiento. La magnitud de esta fuerza depende del coeficiente de rozamiento dinámico. En *A-Frame*, el rozamiento dinámico se puede configurar utilizando los atributos *linear-damping* y *angular-damping* a nivel de objeto, además del atributo *friction* a nivel de escena como se ha mencionado anteriormente.

### Pruebas de la fricción

Se han realizado diferentes pruebas de fricción colocando un objeto en una rampa y variando la inclinación de la misma. Los resultados obtenidos han sido los siguientes:

- **Prueba del rozamiento estático:** se coloca un objeto sobre una rampa y se procede a la variación de la inclinación de la misma. El objeto permanece en la misma posición hasta que se alcanza un ángulo de inclinación tan elevado que el cuerpo comienza a descender (ángulo crítico). Si se configura el atributo *friction* con un valor más elevado, el ángulo crítico se alcanza más tarde, es decir, es mayor.
- **Prueba del rozamiento dinámico:** se mantiene fija la inclinación de la rampa y se varía el valor de los atributos *friction*, *linear-damping* y *angular-damping*. Con unos valores

elevados de esos tres parámetros, los robots no son capaces de subir la rampa. Sin embargo, a medida que se va reduciendo el valor de los atributos los robots comienzan a poder subir la rampa y cada vez lo hacen con mayor facilidad.

## 4.2. Motor de físicas complementario

### 4.2.1. Diseño

Este motor de físicas se utiliza para controlar el movimiento de los cuerpos dinámicos en la escena. Se le denomina complementario puesto que sólo se encarga de aplicar la fuerza autónoma al robot, dejando como tarea de *CANNON* materializar la fricción y la gravedad. Esta idea de motor complementario es opuesta a la implementación que existía hasta el momento. La función *updatePosition* actuaba como un motor completo que sobreescribía a *CANNON*. No existía combinación alguna entre ambos motores.

La arquitectura del motor de físicas complementario presenta dos niveles. El nivel superior conecta directamente con el cerebro de la aplicación, el cual es capaz de dar instrucciones en posición y en velocidad. No obstante, ambos tipos de instrucciones se basan en comandar una velocidad objetivo. En el caso de las instrucciones en posición el robot se moverá a la velocidad objetivo hasta que se alcance la posición consignada en la instrucción y en el caso de las instrucciones en velocidad, el movimiento no termina. Este nivel más alto se basa en un controlador PD que traduce las consignas de velocidad que le llegan del cerebro, en la fuerza necesaria a aplicar al robot para alcanzar dichas consignas. Estas fuerzas son las que entiende el núcleo o nivel más bajo del motor. Conociendo la masa y el momento de inercia, las fuerzas obtenidas se podrán traducir en aceleraciones. La Figura 4.4 muestra un esquema del diseño del motor.

### 4.2.2. Nivel básico: modelo de fuerzas

El núcleo del motor de físicas es un modelo de fuerzas en el que, a partir de la definición de la masa y el momento de inercia del robot, se calcula la aceleración o torque a aplicar. De esta

---

<sup>7</sup>Elaboración propia.

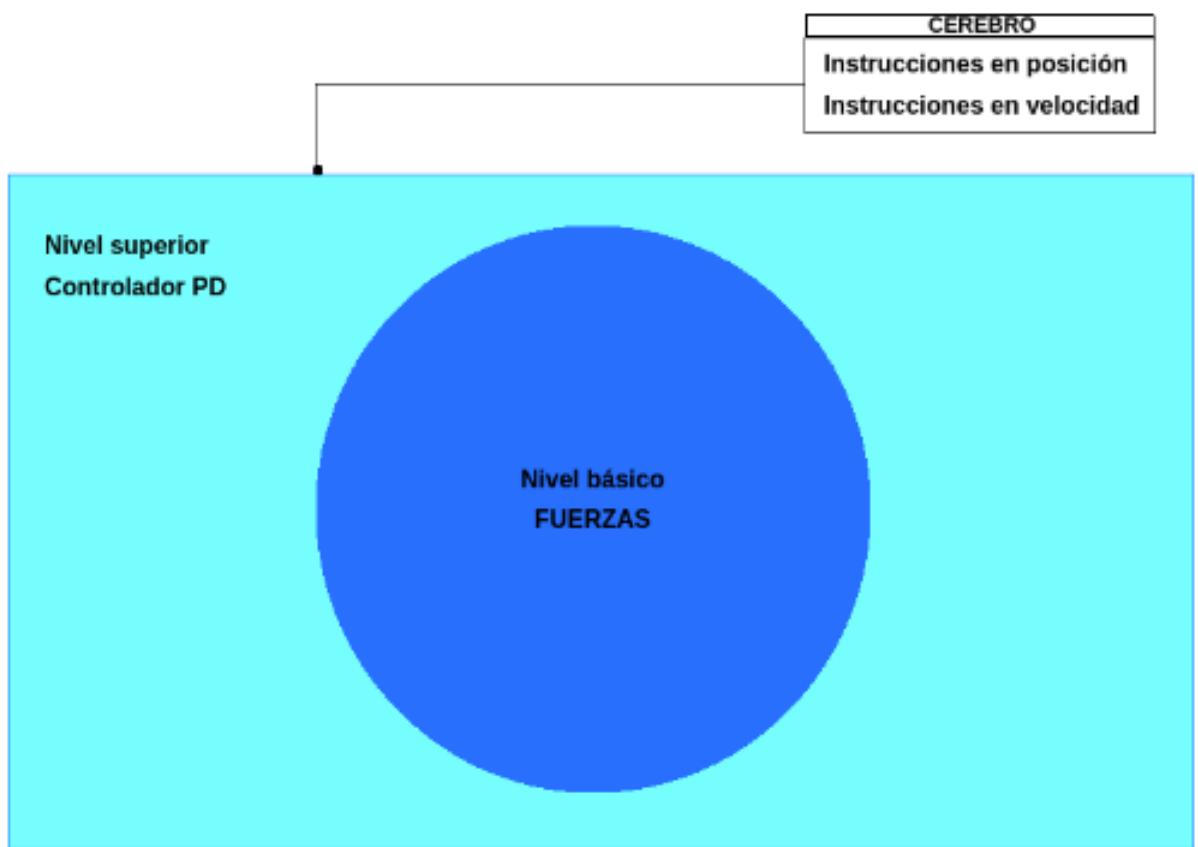


Figura 4.4: Diseño del motor de físicas complementario<sup>7</sup>

manera, un robot muy pesado deberá ejercer una fuerza y, por tanto, una aceleración mayor que un robot más ligero para alcanzar una misma velocidad.

Para que el modelo funcione es necesaria la definición de los siguientes parámetros:

- **Fuerza máxima:** es la fuerza máxima que se puede aplicar a un robot.
- **Torque máximo:** es el momento de fuerza máximo que se puede aplicar a un robot, es decir, es la fuerza máxima de giro aplicable.
- **Masa:** es necesaria para obtener el valor de la aceleración lineal.
- **Momento de inercia:** es la medida de la inercia rotacional cuando un cuerpo gira. Es el equivalente a la masa en un movimiento angular. Es necesario para obtener el valor de la aceleración angular.

```
/* Masa */
const mass = this.robot.body.mass;

/* Inercia */
const inertia = this.robot.body.inertia.x;

/* Fuerza y torque máximos */
const fMax = 100;
const tMax = 100;

/* Aceleración lineal y angular máxima */
const accelerationMax = fMax / mass;
const angularAccelerationMax = tMax / inertia;
```

#### 4.2.3. Nivel superior: controlador PD

El nivel superior del motor de físicas complementario está formado por un controlador PD que traduce las consignas de velocidad que le llegan directamente del cerebro.

El controlador PD es una variante del controlador PID (controlador proporcional, integral y derivativo) que no incluye la componente integral. Se trata de un controlador por realimentación que calcula la desviación o error entre una medida y el valor que se desea obtener. Cada componente tiene una utilidad diferente y depende de distintos factores:

- **Componente proporcional:** depende del error actual y su función es minimizar el error del sistema.
- **Componente derivativa:** depende de los errores pasados y permite estabilizar el sistema reduciendo la oscilación del valor de salida.
- **Componente integral:** es una predicción de los errores futuros y se utiliza cuando el componente derivativo no consigue reducir el error del sistema. Su uso es complejo ya

que puede producir la desestabilización del sistema.

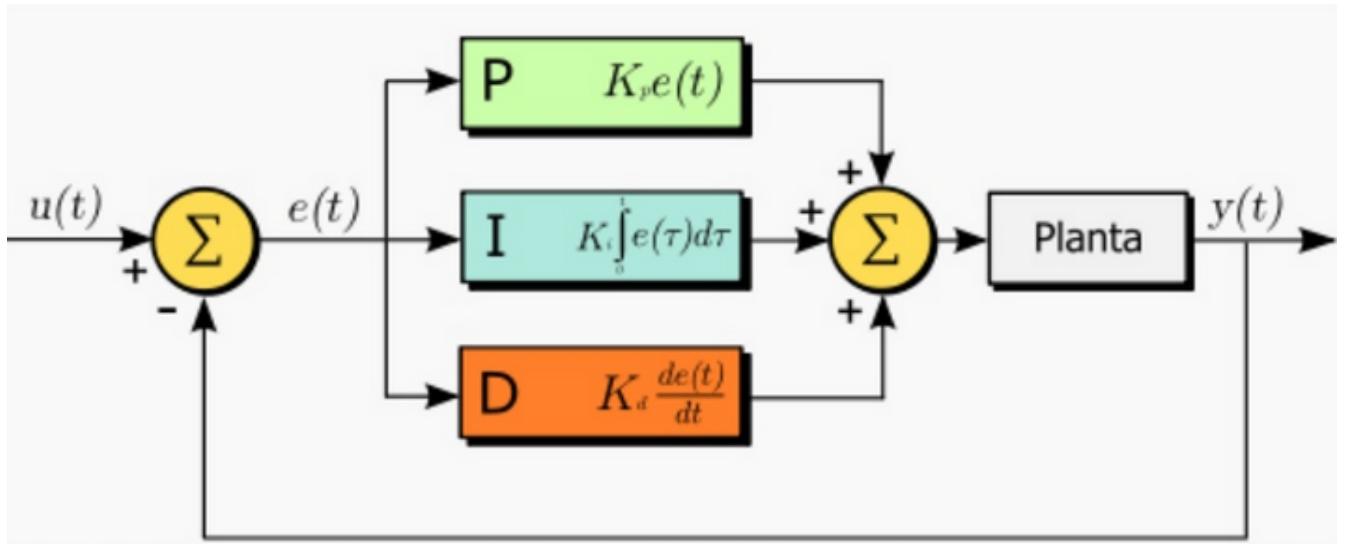


Figura 4.5: Controlador PID<sup>8</sup>

Mediante un sencillo algoritmo basado en la suma de estos tres componentes, el controlador es capaz de ajustar su salida a un valor de referencia. Además, el modelo incluye tres constantes que se emplean para ponderar los componentes anteriores. En este caso sólo se van a tener en cuenta los componentes proporcionales y derivativos puesto que se van a implementar controladores PD porque el error del sistema no es excesivamente elevado.

$$c(t) = K_p \cdot e(t) + K_i \cdot \int e(t) dt + K_d \cdot \frac{\partial e(t)}{\partial t}$$

Esta capa del motor incluye cuatro controladores PD diferentes que se ejecutan dependiendo del tipo de robot que esté realizando el movimiento (robot terrestre o drone) y del tipo de movimiento que se efectúe (avance lineal, giro, vuelo o suspensión en el aire). A continuación, se detallan los tipos de controladores que incluye el modelo:

- **Controlador PD en velocidad del plano XZ:** coge como entrada la velocidad resultante del plano horizontal en ese instante y genera como salida la fuerza que debe ejercer el robot para alcanzar la velocidad objetivo. Es empleado en el movimiento lineal tanto de robots terrestres como de drones.

<sup>8</sup><https://es.slideshare.net/quasar.0360.7912/sintonizacion-de-controladores-pid>

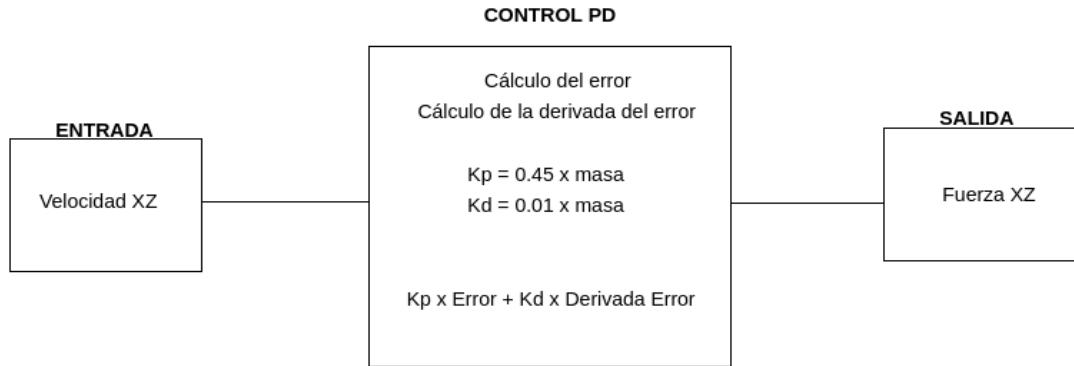


Figura 4.6: Diseño controlador PD en velocidad del plano XZ<sup>9</sup>

- **Controlador PD en velocidad del eje Y:** coge como entrada la componente Y de la velocidad en ese instante y genera como salida la fuerza que debe ejercer el robot para alcanzar la velocidad objetivo. Es empleado en el vuelo de los drones.

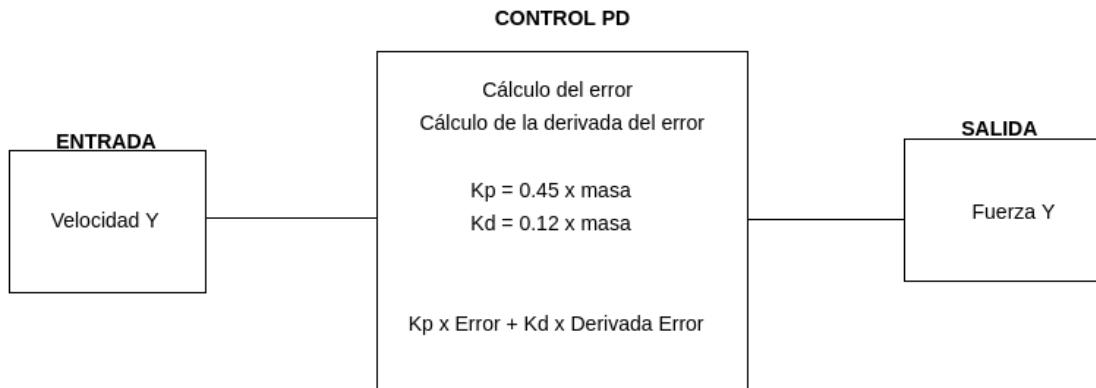


Figura 4.7: Diseño controlador PD en velocidad del eje Y<sup>10</sup>

- **Controlador PD en velocidad angular:** coge como entrada la componente Y de la velocidad angular en ese instante y genera como salida el torque que debe ejercer el robot para alcanzar la velocidad de giro objetivo. Es empleado durante los giros tanto de robots terrestres como de drones. No obstante, las constantes proporcional y derivativa y el torque máximo varían en función del tipo de robot, puesto que en el caso del drone la fricción no

<sup>9</sup>Elaboración propia.

<sup>10</sup>Elaboración propia.

## 4.2. MOTOR DE FÍSICAS COMPLEMENTARIAS Y MEJORA DE LAS FÍSICAS EN WEBSIM

opone resistencia durante el giro, ya que este se encuentra volando sin mantener contacto con ninguna superficie.

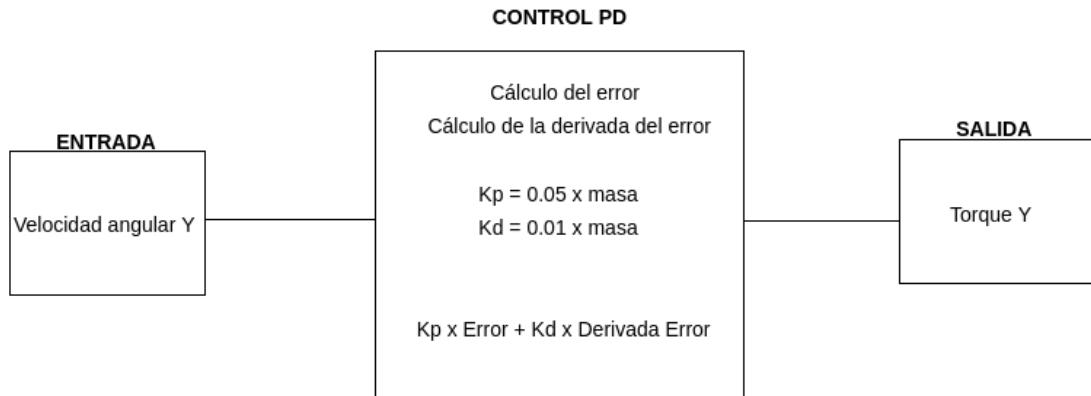


Figura 4.8: Diseño controlador PD en velocidad angular para drone<sup>11</sup>

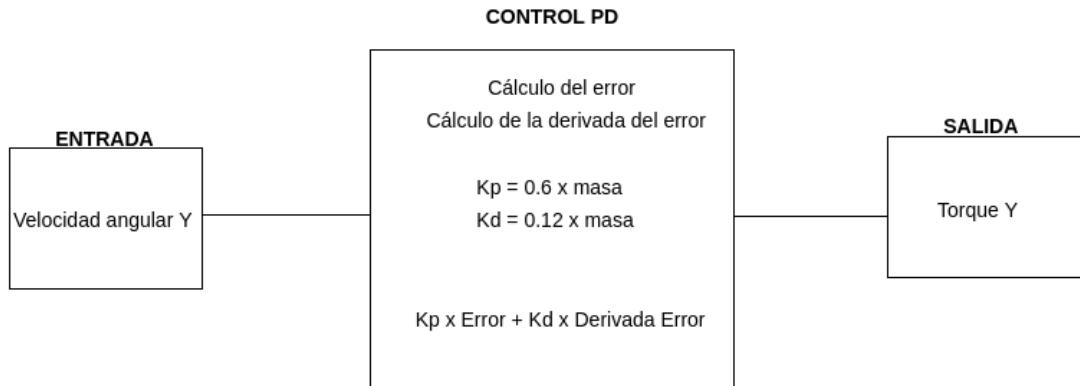


Figura 4.9: Diseño controlador PD en velocidad angular para robot terrestre<sup>12</sup>

- **Controlador PD en posición:** coge como entrada la componente Y de la posición en ese instante y genera como salida la fuerza que debe ejercer el robot para mantener una posición de referencia. Es empleado cuando un drone debe permanecer inmóvil durante el vuelo en una posición concreta.

<sup>11</sup>Elaboración propia.

<sup>12</sup>Elaboración propia.

<sup>13</sup>Elaboración propia.

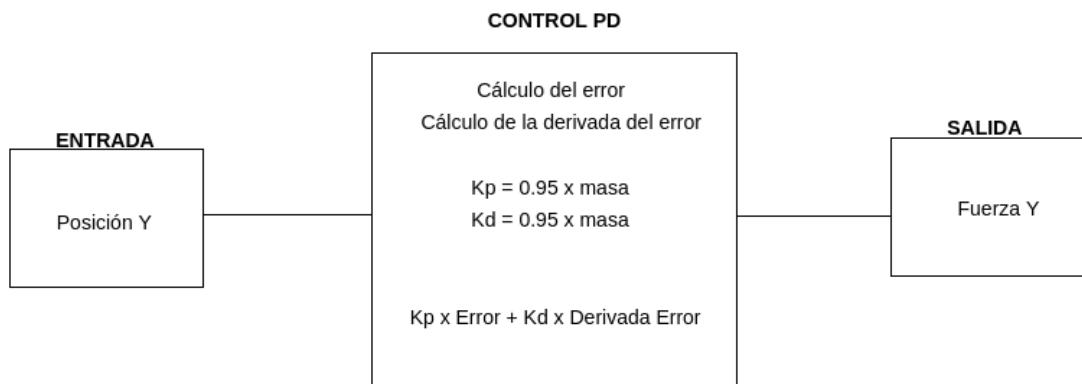


Figura 4.10: Diseño controlador PD en posición<sup>13</sup>

A continuación se incluyen los fragmentos de código que ha sido necesario incluir en el fichero *interfacesRobot.js* del *Simcore* de *WebSim* para el correcto funcionamiento de los controladores PD.

**Adición de variables a la clase robot ya creada.** De esta manera, el motor complementario es escalable a ejercicios multirobot. Cada robot mantiene un registro propio del valor de sus variables.

```

export class RobotI {
    constructor(robotId) {
        this.errorY = 0;
        this.errorXZ = 0;
        this.errorW = 0;
        this.errorActualY = 0;
        this.errorActualXZ = 0;
        this.errorActualW = 0;
        this.derivadaErrorY = 0;
        this.derivadaErrorXZ = 0;
        this.derivadaErrorW = 0;
        this.forcePD = 0;
        this.accelerationPD = 0;
    }
}

```

## 4.2. MOTOR DE FÍSICAS COMPLEMENTARIAS Y MEJORA DE LAS FÍSICAS EN WEBSIM

```
        this.commandedVelocityY = 0;
        this.commandedVelocityXZ = 0;
        this.commandedVelocityW = 0;
        this.accelerationPDY = 0;
        this.accelerationPDXZ = 0;
        this.accelerationPDW = 0;
        this.resultVelocity = 0;
        this.refPos = 0;
        this.init = true;
        this.stop = true;
        this.motorIterations = 0;
    }
```

### **Función auxiliaryPhysics.**

```
/*Actualización del contador de iteraciones*/
auxiliaryPhysics() {
    this.motorIterations = getTickCounter();

    /* Sólo para Drone */
    /* Si la velocidad de vuelo objetivo es 0: */
    if ((this.velocity.y <= 0.0001) || (this.velocity.y <= -0.0001)){
        if (this.init == true) {
            /* Si el movimiento aún no ha comenzado la aceleración debe ser 0 */
            this.accelerationPDY = 0;
        } else {
            /* Si el movimiento ya ha comenzado: */
            if (this.stop == true) {
                /* Si se acaba de quedar en suspensión, guardo la posición de referencia */
                this.refPos = this.robot.body.position.y;
            }
            /* Entra el controlador PD en posiciones */
            this.stop = false;
            this.accelerationPDY = this.controladorPDVerticalPos();
        }
    }
    /* Si la velocidad objetivo no es 0, se ejecuta el controlador PD en velocidades */
}
```

```
    } else {
        this.init = false;
        this.stop = true;
        this.accelerationPDY = this.controladorPDVerticalVel();
    }

    /* Se utiliza la aceleración calculada con los controladores para obtener la velocidad a aplicar utilizando las fórmulas MRUA y la combinación entre motores */

    this.commandedVelocityY = this.robot.body.velocity.y +
        this.motorIterations*this.accelerationPDY;
    this.robot.body.velocity.set(this.robot.body.velocity.x,
        this.commandedVelocityY, this.robot.body.velocity.z);
}

/* Movimiento en el plano horizontal: drone y robot terreste */

/* Velocidad resultante = RaízCuadrada(Vx + Vz) */
this.resultVelocity = Math.sqrt(Math.pow(this.robot.body.velocity.x, 2) +
Math.pow(this.robot.body.velocity.z, 2));

/* Entra el controlador PD */
this.accelerationPDXZ = this.controladorPDHorizontal(this.resultVelocity);

/* Se utiliza la aceleración calculada con los controladores para obtener la velocidad a aplicar utilizando las fórmulas MRUA y la combinación entre motores */
this.commandedVelocityXZ = this.resultVelocity +
this.motorIterations*this.accelerationPDXZ;
let rotation = this.getRotation();

/* La velocidad comandada resultante se descompone en las velocidades Vx y Vz */
this.robot.body.velocity.set(this.commandedVelocityXZ*Math.cos(rotation.y*Math.PI/180),
this.robot.body.velocity.y,this.commandedVelocityXZ*Math.sin(-rotation.y*Math.PI/180));

/* Movimiento angular: drone y robot terreste */

/* Entra el controlador PD angular */
this.accelerationPDW = this.controladorPDAngular();
```

```

/* Se utiliza la aceleración calculada con el controlador para obtener
la velocidad angular a aplicar utilizando las fórmulas MRUA y la
combinación entre motores */
this.commandedVelocityW = this.robot.body.angularVelocity.y +
this.motorIterations*this.accelerationPDW;
this.robot.body.angularVelocity.set(0, this.commandedVelocityW, 0);

setTimeout(this.auxiliaryPhysics.bind(this), 20);
}

```

### **Controlador PD en velocidad del plano XZ.**

```

/* Código del controlador PD en velocidad */
controladorPDHorizontal(resultVelocity) {
    /* Definición constantes para el controlador, fuerza máxima y aceleración máxima */
    const mass = this.robot.body.mass;
    const kp = 0.45*mass;
    const kd = 0.01*mass;
    const fMax = 100000000;
    const accelerationMax = fMax / mass;

    /* Cálculo del error y derivada del error*/
    this.errorActualXZ = this.velocity.x - resultVelocity;
    this.derivadaErrorXZ = this.errorActualXZ - this.errorXZ;
    this.errorXZ = this.errorActualXZ;

    /* Salida del controlador */
    this.forcePD = kp*this.errorActualXZ + kd*this.derivadaErrorXZ;

    /* Obtención de la aceleración teniendo en cuenta la masa */
    this.accelerationPD = this.forcePD / mass;

    /* Límite de aceleración aplicable en cada iteración */
    if (Math.abs(this.accelerationPD) > angularAccelerationMax) {
        if (this.accelerationPD > 0) {
            this.accelerationPD = angularAccelerationMax;
        } else {

```

```
        this.accelerationPD = - angularAccelerationMax;  
    }  
}  
return this.accelerationPD;  
}
```

### **Controlador PD en velocidad del eje Y.**

```
/* Código del controlador PD en velocidad */  
controladorPDVerticalVel() {  
    /* Definición constantes para el controlador, fuerza máxima y  
    const mass = this.robot.body.mass;  
    const kp = 0.45*mass;  
    const kd = 0.12*mass;  
    const fMax = 1000000;  
    const accelerationMax = fMax / mass;  
  
    /* Cálculo del error y derivada del error*/  
    this.errorActualY = this.velocity.y - this.robot.body.velocity.y;  
    this.derivadaErrorY = this.errorActualY - this.errorY;  
    this.errorY = this.errorActualY;  
  
    /* Salida del controlador */  
    this.forcePD = kp*this.errorActualY + kd*this.derivadaErrorY;  
  
    /* Obtención de la aceleración teniendo en cuenta la masa */  
    this.accelerationPD = this.forcePD / mass;  
  
    /* Límite de aceleración aplicable en cada iteración */  
    if (Math.abs(this.accelerationPD) > angularAccelerationMax) {  
        if (this.accelerationPD > 0) {  
            this.accelerationPD = angularAccelerationMax;  
        } else {  
            this.accelerationPD = - angularAccelerationMax;  
        }  
    }  
    return this.accelerationPD;  
}
```

### Controlador PD en velocidad angular.

```
/* Código del controlador PD en velocidad angular*/
controladorPDAngular() {
    /* Definición constantes para el controlador, torque máximo, inercia
    y aceleración máxima */
    const mass = this.robot.body.mass;
    const inertia = this.robot.body.inertia.x;
    const angularAccelerationMax = tMax / inertia;

    /* Distinción entre robot terrestre o volador. Init es true cuando un
    robot comienza el vuelo por primera vez */
    if (this.init == false) {
        var kp = 0.6*mass;
        var kd = 0.12*mass;
        var tMax = 100;
    } else {
        var kp = 0.05*mass;
        var kd = 0.01*mass;
        var tMax = 1;
    }

    /* Cálculo del error y derivada del error*/
    this.errorActualW = this.velocity.ay - this.robot.body.angularVelocity.y;
    this.derivadaErrorW = Math.abs(this.errorW - this.errorActualW);
    this.errorW = this.errorActualW;

    /* Salida del controlador */
    this.forcePD = kp*this.errorActualW + kd*this.derivadaErrorW;

    /* Obtención de la aceleración teniendo en cuenta la inercia */
    this.accelerationPD = this.forcePD / inertia;

    /* Límite de aceleración aplicable en cada iteración */
    if (Math.abs(this.accelerationPD) > angularAccelerationMax) {
        if (this.accelerationPD > 0) {
            this.accelerationPD = angularAccelerationMax;
        } else {

```

```
        this.accelerationPD = - angularAccelerationMax;  
    }  
}  
return this.accelerationPD;  
}
```

### **Controlador PD en posición.**

```
/* Código del controlador PD en posición */  
controladorPDVerticalPos() {  
    /* Definición constantes para el controlador, fuerza máxima y  
    const mass = this.robot.body.mass;  
    const kp = 0.95*mass;  
    const kd = 0.95*mass;  
    const fMax = 100000000;  
    const accelerationMax = fMax / mass;  
  
    /* Cálculo del error y derivada del error*/  
    this.errorActualY = this.refPos - this.robot.body.position.y;  
    this.derivadaErrorY = this.errorActualY - this.errorY;  
    this.errorY = this.errorActualY;  
  
    /* Salida del controlador */  
    this.forcePD = kp*this.errorActualY + kd*this.derivadaErrorY;  
  
    /* Obtención de la aceleración teniendo en cuenta la masa */  
    this.accelerationPD = this.forcePD / mass;  
  
    /* Límite de aceleración aplicable en cada iteración */  
    if (Math.abs(this.accelerationPD) > angularAccelerationMax) {  
        if (this.accelerationPD > 0) {  
            this.accelerationPD = angularAccelerationMax;  
        } else {  
            this.accelerationPD = - angularAccelerationMax;  
        }  
    }  
    return this.accelerationPD;  
}
```

#### 4.2.4. Timing

Este concepto es de especial relevancia ya que es el que hace que el motor sea complementario y no completo, es decir, es lo que permite que el motor complementario se combine con el de *CANNON* y no sobreescriba sus modificaciones.

El motor de físicas complementario se ejecuta cada 20 ms gracias a un *timeout* que lo invoca de forma periódica. Pero, ¿cuándo se ejecuta el motor de *CANNON*?

```
setTimeout (this.auxiliaryPhysics.bind(this), 20);
```

*CANNON* actualiza sus físicas en cada iteración del bucle de renderizado de *A-Frame*. Además, no lleva una cuenta explícita del tiempo y tampoco lo tiene en cuenta a la hora de modificar las posiciones y velocidades de los objetos de la escena. Puesto que la frecuencia de ejecución de *CANNON* es superior a la del motor complementario, ha sido necesario calcular el número de veces que se ejecuta el código *CANNON* entre dos iteraciones del motor de físicas complementario para poder realizar una correcta combinación entre ambos. El motor complementario deberá aplicar en cada iteración una aceleración X veces superior a la que corresponde, siendo X el número de veces que ha entrado *CANNON* desde la última vez que se ejecutó el motor complementario.

$$\text{Aceleración autónoma} = \text{iteracionesCANNON} \times \text{aceleración calculada}$$

El cómputo de las iteraciones de *CANNON* se ha podido realizar gracias a la creación de un nuevo componente auxiliar que hace incrementar en uno un contador por cada tick de renderizado que ejecuta *A-Frame*.

Las variables y funciones que se han añadido al código original para llevar a cabo la implementación han sido las que se incluyen a continuación. Las funciones *tickCounter*, *getTickCounter* y *setTickCounter* se han utilizado para contabilizar el número de veces que se ejecuta el motor de *CANNON* entre dos iteraciones del motor complementario.

```
export var tickCounter = 0;

export function getTickCounter() {
    return tickCounter;
}

export function setTickCounter(value) {
    tickCounter = value;
}
```

Para crear el nuevo componente se han utilizado las herramientas que ofrece *A-Frame* para el registro de nuevos componentes. La siguiente línea de código realiza el registro del nuevo componente auxiliar que se utiliza para contabilizar las iteraciones de *CANNON*. A continuación, también se ha incluido el código del tick del nuevo componente registrado.

```
/* Registro del nuevo componente */
AFRAME.registerComponent("iterations", iterationsObj);

/* Función tick del componente "iterations" */
export var iterationsObj = {
    schema: {
        count: { type: 'number', default: 0 },
        position: { "x":0, "y":0, "z":0 }
    },
    tick: function(){
        setTickCounter(getTickCounter() + 1);
        console.log('Tick de renderizado de A-FRAME');
    }
}
```

Gracias a la correcta combinación de ambos motores, se consigue que el motor de físicas complementario calcule la fuerza autónoma a aplicar a los robots dinámicos y *CANNON* materialice la gravedad y fricción de la escena.

#### 4.2.5. Motor de físicas complementario para robots terrestres

El motor de físicas complementario permite dotar a *WebSim* de un motor de físicas más realistas puesto que únicamente actúa sobre la fuerza autónoma del robot, dejando libertad a *CANNON* para materializar tanto fricción como gravedad. Gracias a ello, se simulan mundos en los que los requisitos se asemejan en mayor medida a las características que se encuentran en el mundo real y en los robots físicos.

En consecuencia, se van a poder desarrollar ejercicios más variados como pistas de hielo o ejercicios multinivel con rampas que interconecten los múltiples niveles. En la Figura 4.11 se muestra cómo influye en la aceleración autónoma que aplica el motor de físicas complementario el valor de fricción que se parametriza en el escenario. Cuanto mayor es la fricción, más grande es la aceleración autónoma que se debe aplicar.

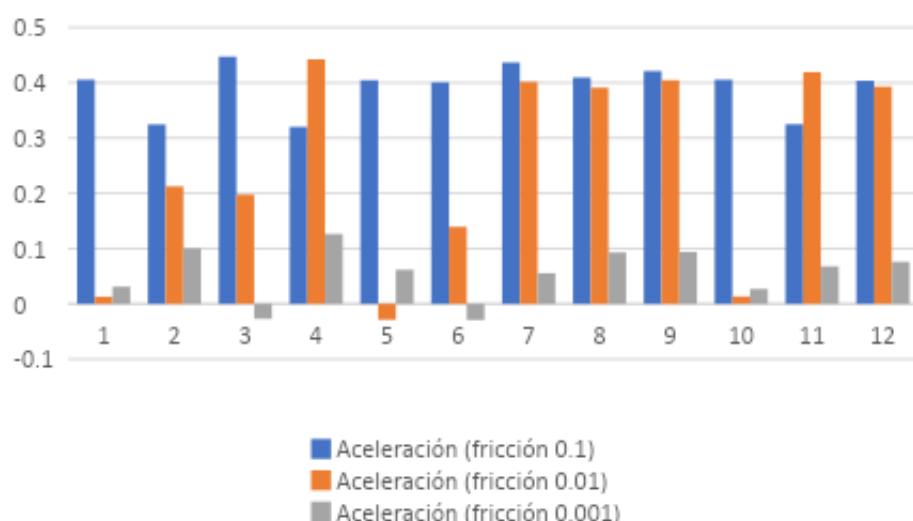


Figura 4.11: Relación fricción - aceleración

La Figura 4.12 muestra la diferencia entre la progresión de la velocidad a lo largo del tiempo con y sin el motor de físicas complementario. Se observa que con el motor de físicas complementario la velocidad tarda unos milisegundos en estabilizarse y que cada vez se va ajustando mejor al valor objetivo.

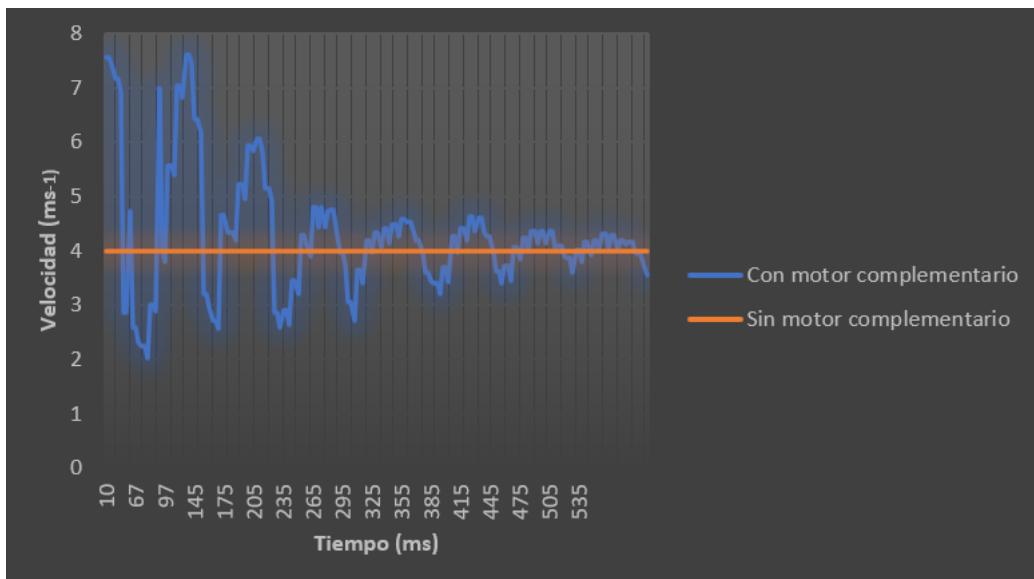


Figura 4.12: Tiempo - Velocidad Controlador PD plano horizontal

#### 4.2.6. Motor de físicas complementario para drones

El motor de físicas complementario permite dotar a *WebSim* de unas físicas más realistas puesto que permite a *CANNON* materializar la gravedad. Gracias a ello, se pueden simular mundos en los que, a pesar de existir una gravedad de -9.8, la fuerza autónoma que hace aplicar el motor complementario permita volar al drone.

Por otro lado, el control PD en posición incrementa el realismo de las físicas también a nivel visual puesto que hace recrear un movimiento más fluido y no produce una brusca frenada como ocurría con la imposición de la posición con *updatePosition*. La fluidez del movimiento se consigue gracias a la fase de estabilización característica del controlador PD hasta que consigue aproximar su salida al valor de referencia. En la Figura 4.13 se aprecia como el controlador tarda unos ms en estabilizar la posición y al principio la posición tiende a caer como consecuencia de la atracción de la gravedad. La Figura 4.14 muestra la progresión de la velocidad a lo largo del tiempo cuando el drone despega.

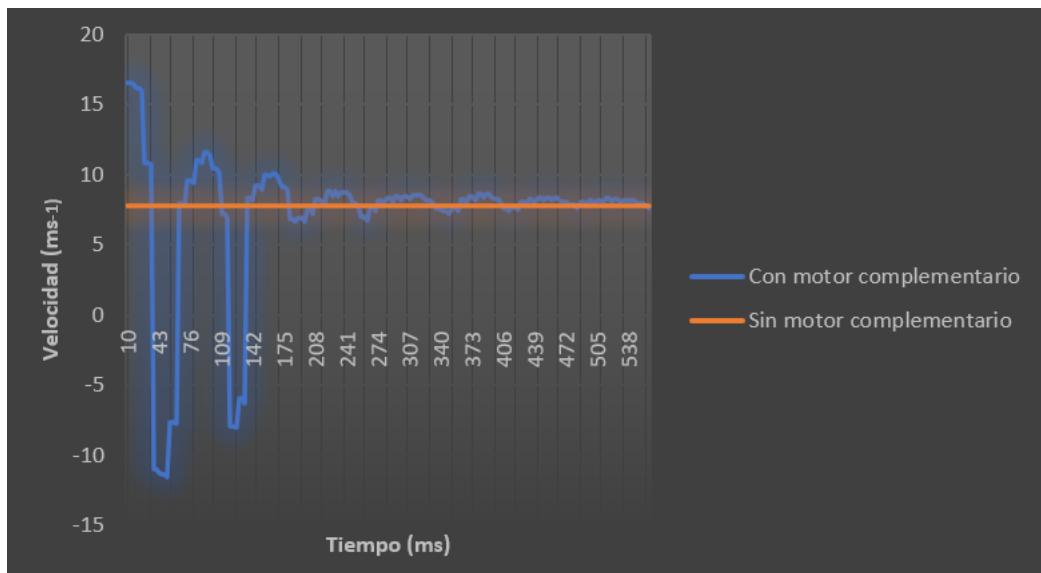


Figura 4.13: Tiempo - Posición Controlador PD eje Y

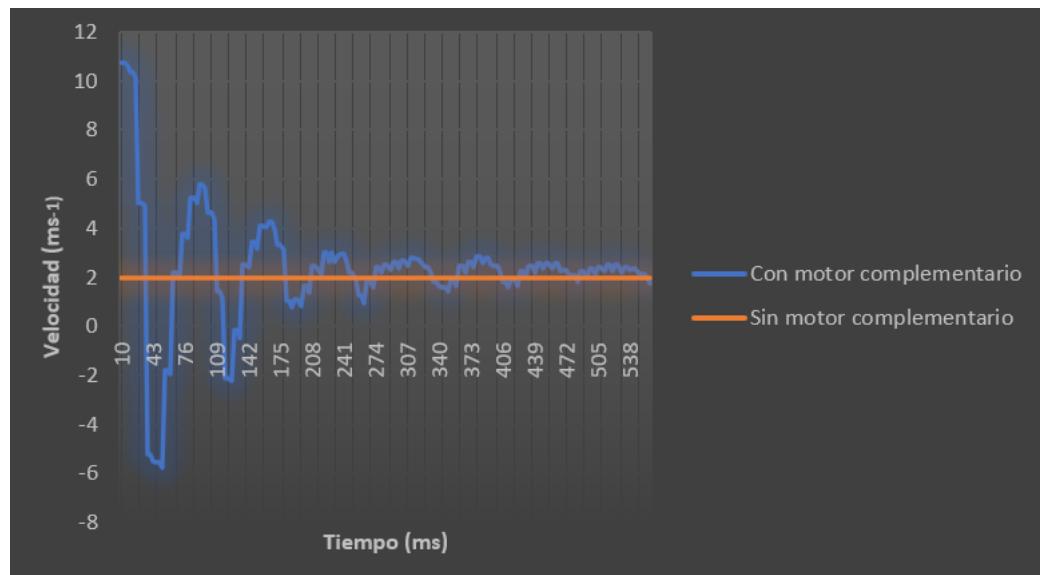


Figura 4.14: Tiempo - Velocidad Controlador PD eje Y

# Capítulo 5

## Nuevos ejercicios

En este capítulo se presentan los nuevos ejercicios que se han creado para enriquecer el contenido ofrecido en el entorno de *Kibotics*. Todos ellos permiten explotar las mejoras que ofrece el motor de físicas complementario implementado.

### 5.1. Sigue-líneas sofisticado

Este ejercicio<sup>1</sup> se basa en la propuesta realizada para la competición *Robocup Junior Australia 2019*<sup>2</sup>. Se trata de una versión mejorada del sigue-líneas tradicional, en el que se incluyen trayectorias más complejas y diversas y dos niveles diferentes conectados a través de una rampa. Actualmente, este ejercicio está disponible en la plataforma *Kibotics* como un ejercicio a resolver en *Python o Scratch* y el robot que se ha seleccionado para su solución ha sido el mBot.

El ejercicio aprovecha las ventajas del motor de físicas complementario en la subida de las rampas. Durante la subida, *CANNON* materializa la fricción oponiendo fuerzas de rozamiento al movimiento, mientras que el motor complementario se encarga de aplicar la fuerza autónoma necesaria para lograr que el robot ascienda por la rampa. La fuerza resultante será más grande que la necesaria para avanzar por el suelo plano sin inclinación y, además, será necesario tener en cuenta el valor de la fricción y la inclinación de la rampa para hacer más fácil o difícil la subida. Tal y como ocurriría en la realidad con un robot físico, se alcanzará un grado de

---

<sup>1</sup><https://www.youtube.com/watch?v=PjTr13M3o5k>

<sup>2</sup><https://www.robocupjunior.org.au/>

inclinación o un valor de fricción con los cuales el mBot no será capaz de avanzar por la rampa puesto que el diseño del motor incluye un límite de fuerza máxima aplicable<sup>3</sup>.

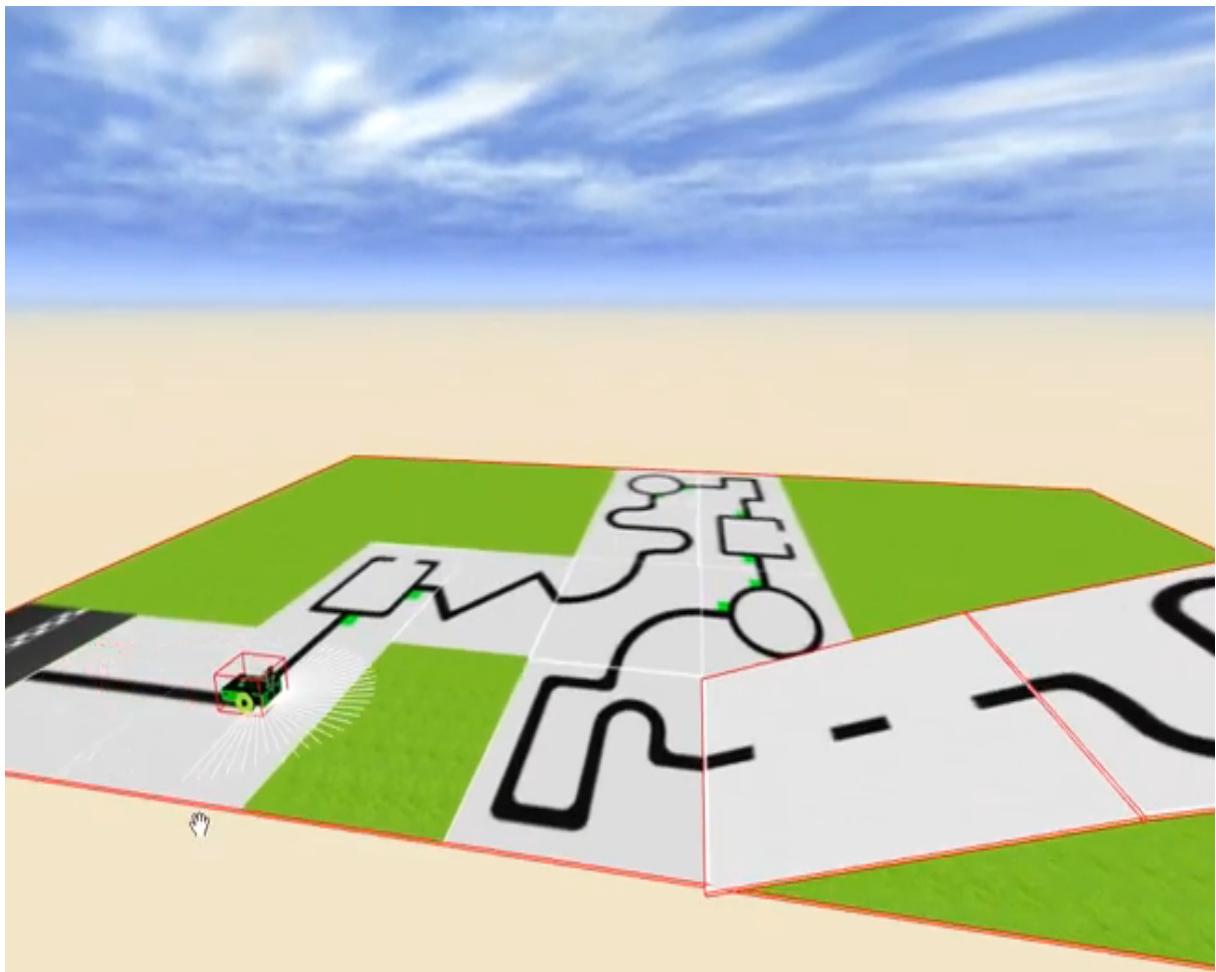


Figura 5.1: Sigue-líneas sofisticado

## 5.2. Laberinto 3D para mBot

Este segundo ejercicio<sup>4</sup> también se basa en la propuesta realizada para la competición *Robocup Junior Australia 2019*<sup>5</sup>. El objetivo de este ejercicio es hacer llegar al robot al punto en el que se encuentra otro robot perdido para rescatarle del laberinto.

<sup>3</sup><https://www.youtube.com/watch?v=8mtK8DVkDZo>

<sup>4</sup><https://www.youtube.com/watch?v=R9JXZCeSNFo>

<sup>5</sup><https://www.robocupjunior.org.au/>

Este ejercicio está disponible para el mBot en el entorno de *Kibotics* para *Python* y *Scratch*. Además, al tratarse de un laberinto multinivel, se ve beneficiado por el motor de físicas complementario en la subida de la rampa, igual que ocurría en el ejercicio anterior<sup>6</sup>.



Figura 5.2: Laberinto 3D para mBot

### 5.3. Laberinto para drone

Se han incluido dos ejercicios nuevos para el drone Tello. El objetivo de ambos ejercicios es que el drone encuentre la salida del laberinto. Este ejercicio aprovecha el motor de físicas complementario en el movimiento del drone, tanto durante el vuelo (controlador PD en velocidad) como cuando el drone permanece quieto a una cierta altura (controlador PD en posición)<sup>7</sup>.

<sup>6</sup><https://www.youtube.com/watch?v=76kgItpMpGk>

<sup>7</sup><https://www.youtube.com/watch?v=jSGI6KJbzTQ>

A continuación, se explican las dos versiones que se han realizado sobre este ejercicio.

### 5.3.1. Sin señalización

La primera versión<sup>8</sup> está pensada para que el usuario logre hacer que el drone encuentre la salida mediante el uso de instrucciones en posición. Por ejemplo, 'avanza 2 metros' o 'gira a la derecha'.

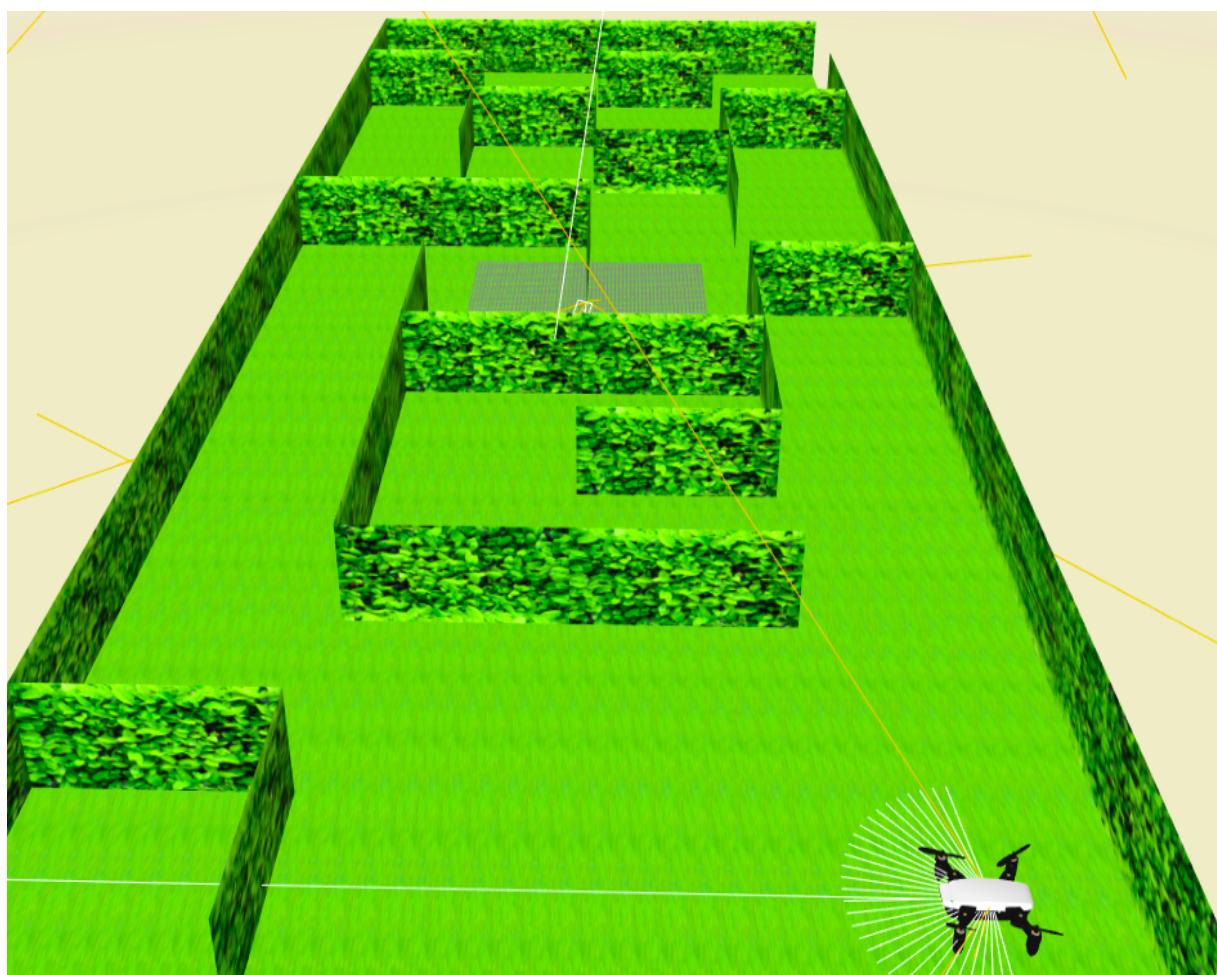


Figura 5.3: Laberinto para drone sin señalización

<sup>8</sup><https://www.youtube.com/watch?v=RLjZPhP6P3g>

### 5.3.2. Con señalización

La segunda versión<sup>9</sup> se ha implementado con el objetivo de que se utilicen los sensores y cámaras del drone para que se capten las señales que se han colocado por las paredes y que guían al drone para encontrar la salida. Gracias a la inteligencia artificial, el drone será capaz de reconocer las señales y traducirlas en un movimiento determinado.

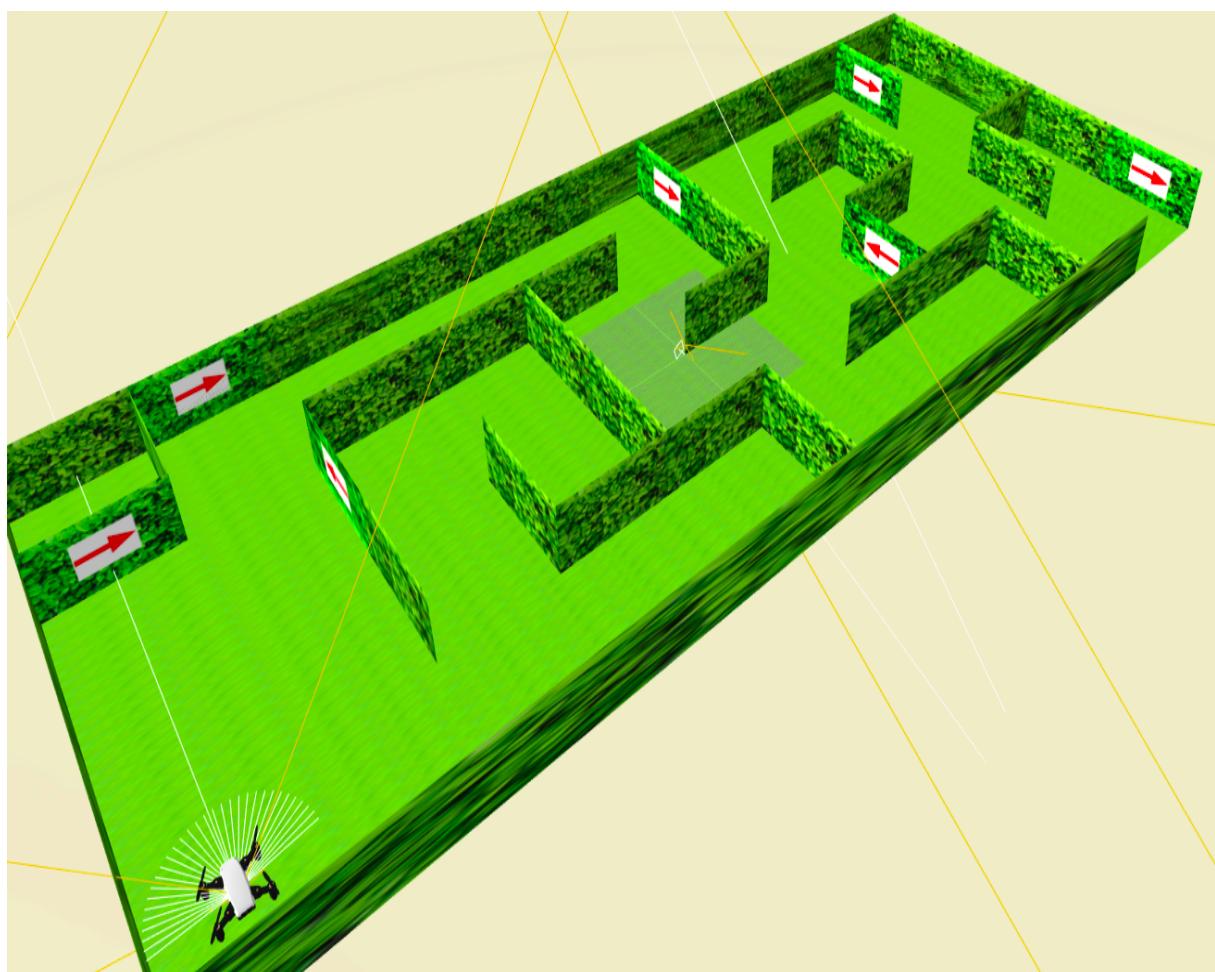


Figura 5.4: Laberinto para drone con señalización

<sup>9</sup><https://www.youtube.com/watch?v=9tSqJQ7pqLc>

## 5.4. Fútbol competitivo

Este último ejercicio<sup>10</sup> también se basa en una de las propuestas presentadas para la competición *Robocup Junior Australia 2019*<sup>11</sup>. El objetivo de este ejercicio competitivo es meter más goles que el contrincante, es decir, simular un partido de fútbol. El primero que llegue a diez goles, será el ganador. Para ello, se ha implementado un evaluador que lleva la cuenta de los goles metidos por cada equipo<sup>12</sup>(Figura 5.6).

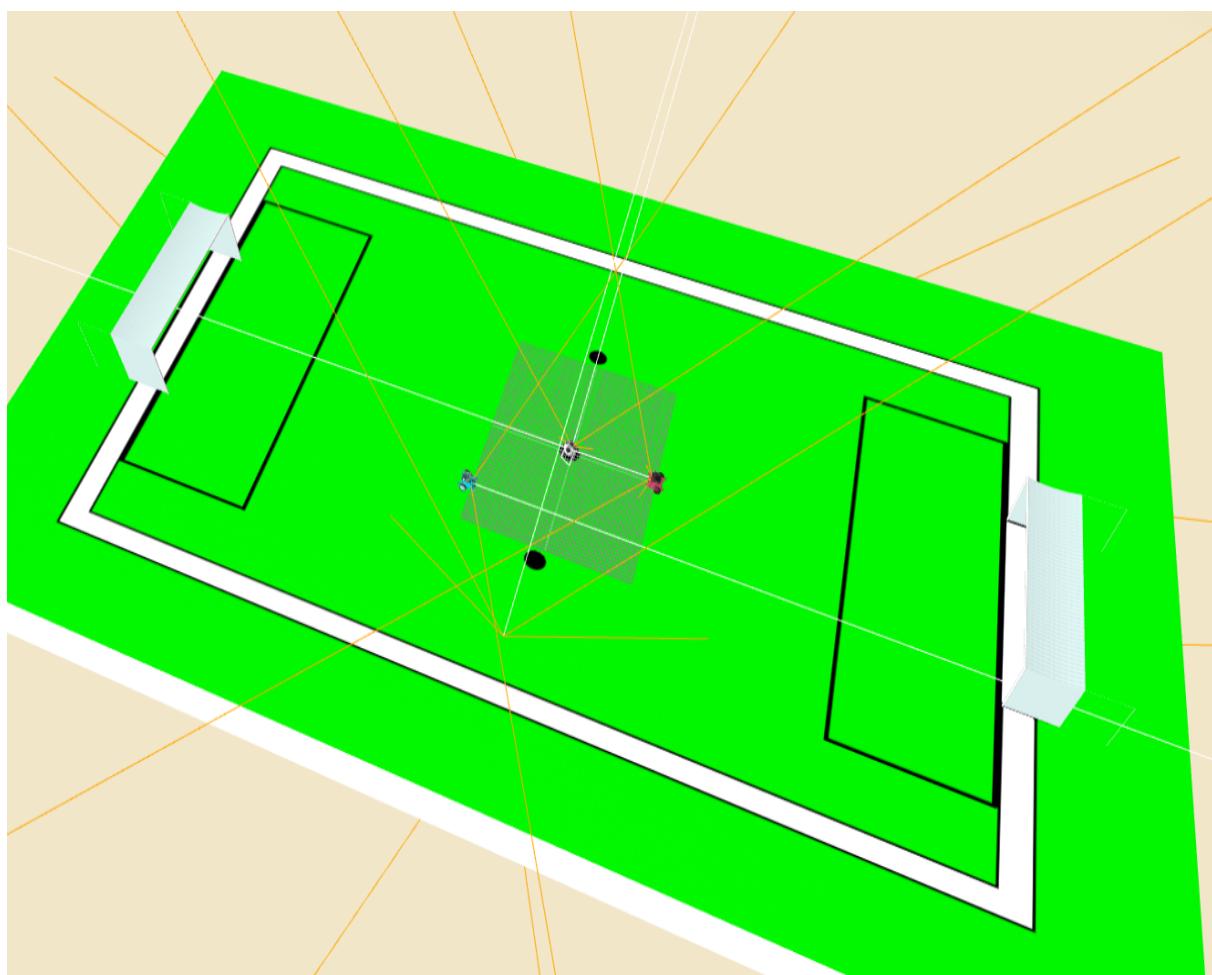


Figura 5.5: Fútbol competitivo

Este ejercicio saca provecho del estudio realizado en las físicas y del motor de físicas complementario tanto en los choques entre los robots y el balón como en el movimiento del balón. Ha sido necesaria la definición de un coeficiente de restitución adecuado para hacer realistas

<sup>10</sup><https://www.youtube.com/watch?v=FYtAFll4keU>

<sup>11</sup><https://www.robocupjunior.org.au/>

<sup>12</sup><https://www.youtube.com/watch?v=lgwECFpTgNk>

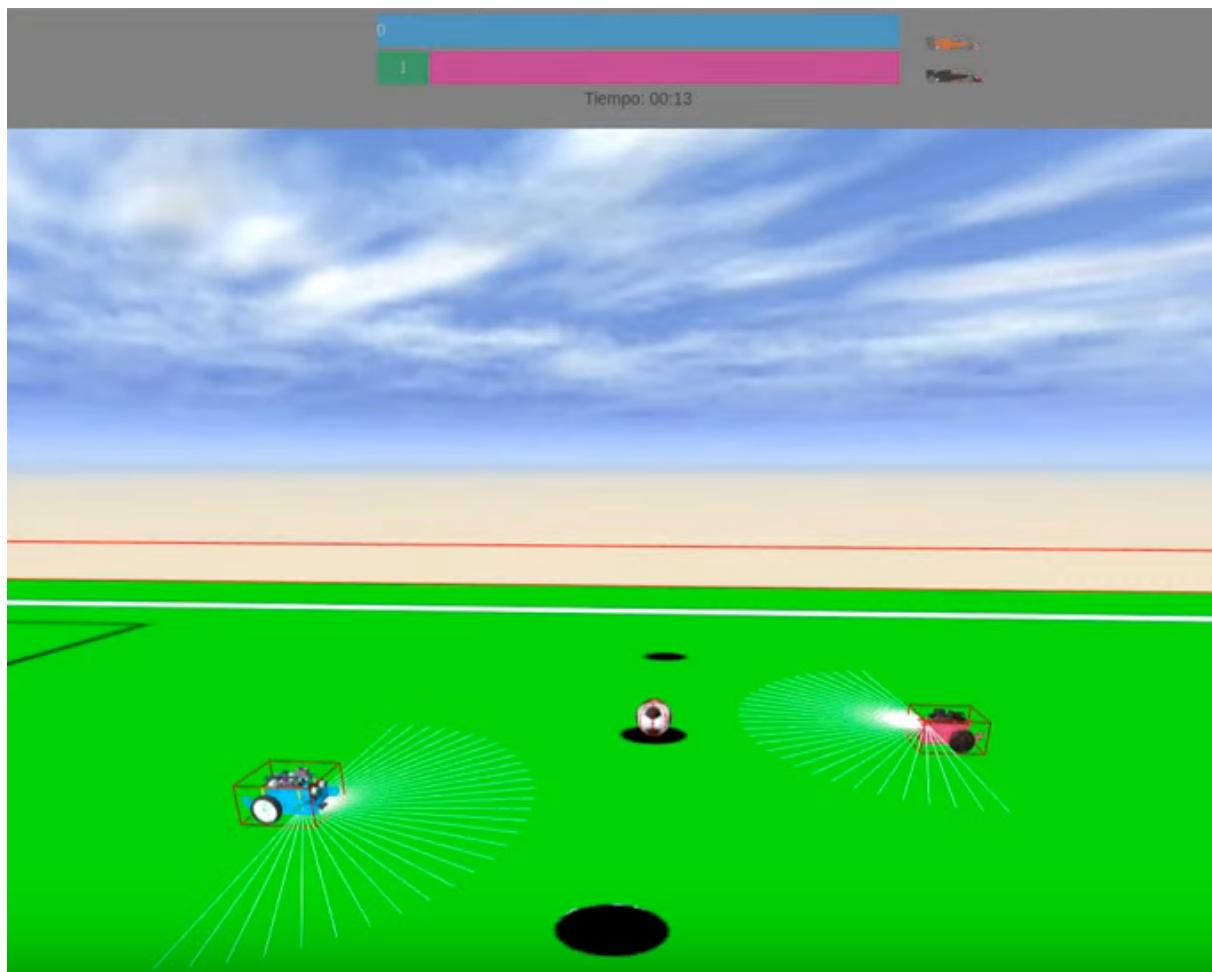


Figura 5.6: Evaluador del ejercicio Fútbol competitivo

tanto los choques como el movimiento de la pelota. En este vídeo<sup>13</sup> se observa un movimiento no realista del balón, mientras que en este otro<sup>14</sup> se ha realizado el ajuste del coeficiente de restitución y la fricción, por lo que la pelota va girando sobre sí misma mientras va avanzando, haciendo mucho más realista el movimiento.

<sup>13</sup>[https://www.youtube.com/watch?v=7W-FB3E0B\\_I](https://www.youtube.com/watch?v=7W-FB3E0B_I)

<sup>14</sup><https://www.youtube.com/watch?v=PIJRqBGPeH4>



# **Capítulo 6**

## **Conclusiones**

- 6.1. Valoración de los resultados**
- 6.2. Mejoras futuras**

