



Proyecto: Epic Racing

Juego de carreras en python/pygame

Autor: Georgiana - Mihaela, Lohan
NIE: X8384862-M
Tutor: Anahí Mula de la Banda

Fecha de entrega: 4 de junio de 2020
Seseña Nuevo, Toledo (España)

Abstract

La creatividad es la capacidad a través de la cual podemos mejorar y elaborar ideas nuevas, diferentes, originales, que nos permite afrontar y resolver los diferentes problemas que la vida nos plantea. [1] **Método:** Estudio avanzado y aplicación del lenguaje de programación **python** haciendo hincapié en el desarrollo de videojuegos mediante éste maravilloso lenguaje de programación. El juego se ha estructurado con la arquitectura MVC (modelo, vista, controlador) gracias a su facilidad de uso y sin usar ningún motor gráfico siendo protagonista la librería de dibujo **pygame**. **Resultados:** Se consigue elaborar un videojuego de carreras bien estructurado y divertido tomando como referencia otros juegos. **Conclusiones:** Programar es divertido, siempre y cuando se tenga la capacidad de organización y seguimiento de unas pautas considerando todos los factores que influyen en la realización de nuestro proyecto y por supuesto, perseverancia.

Palabras clave: python, pygame, MVC (modelo, vista controlador), creatividad, mundo, jugador, pista, objeto, principal, revoluciones, frenar, velocidad, girar.

Creativity is the ability of the human being to create some other ideas that are new and interesting, is, the ability to analyze and evaluate new ideas, solving problems that arise in the course of human life and challenges that arise in academic and everyday. [1] **Method:** Advanced study and application of the programming language **python** emphasizing the development of video games using this wonderful programming language. The game has been structured with the MVC architecture (model, view, controller) thanks to its ease of use and without using any graphics engine, with the drawing library **pygame** taking center stage. **Results:** It is possible to create a well-structured and fun racing video game taking other similar games as a reference. **Conclusions:** Programming is fun, as long as we have the ability to organize and follow certain guidelines, all the factors that influence the realization of our project and of course, perseverance.

Keywords: python, pygame, MVC (model, controller view), creativity, world, player, track, object, main, revolutions, brake, speed, spin.

Índice de Contenidos

1. Justificación del proyecto	1
2. Introducción	2
3. Objetivos	3
4. Desarrollo	4
4.1. Desarrollo del prototipo	4
4.1.1. Arquitectura de diseño MVC	4
4.1.2. Desarrollo de clase Player	6
4.1.3. Desarrollo de clases Track y World	12
4.1.4. Desarrollo del controlador	14
4.1.5. Desarrollo de la clase View	16
4.1.6. Estructura de archivos	17
4.1.7. Prototipo final	18
4.2. Desarrollo de las extensiones	19
4.2.1. Sistema de revoluciones y marchas del motor	19
4.2.2. Efectos de frenado y fantasma de mejor vuelta	22
4.2.3. Panel de visualización de resultados y scoreboard en línea	23
4.2.4. Sistema de menús	25
5. Conclusiones	28
Referencias	29
Anexo A. Bases de datos	30
Anexo B. Diagrama de Gantt	37
Anexo C. Manual de usuario	38
Anexo D. Como se ha generado este documento	45

Índice de Figuras

1.	Escena típica de <i>Epic Racing</i>	1
2.	Diseño MVC de la aplicación.	5
3.	Interacción de los objetos en el mundo.	5
4.	Estructura final sistema de objetos e interacción.	18
5.	Prototipo de la aplicación.	18
6.	Componentes de la interfaz gráfica del juego, velocímetro, número de vuelta y contador de revoluciones.	22
7.	Marcas de frenado y fantasma de mejor vuelta.	23
8.	Panel de resultados y scoreboard de mejor vuelta.	24
9.	Menú principal del juego.	25
10.	Menú de opciones.	26
11.	Esquema final de la aplicación.	27
B.1.	Diagrama de Gantt	37
C.1.	Ubicación archivo main.py necesario para ejecutar el juego	38
C.2.	Moverse por carpetas con el comando CD	38
C.3.	Ejecutar el juego	39
C.4.	Formulario para crear perfil de usuario	39
C.5.	Menú principal	40
C.6.	Opción menú Configuraciones	40
C.7.	Opción menú Ayuda	41
C.8.	Menú para iniciar carrera	42
C.9.	Ventana juego iniciado	43
C.10.	Ventana scoreboard partida	44

Índice de Tablas

1.	Características y métodos principales de cada clase principal.	6
2.	Eventos controlador del automóvil.	14
3.	Controles del menú.	14
A.1.	Campos de la tabla de la base de datos.	30

Índice de Códigos

1.	player.py - Función que actualiza la velocidad del automóvil.	8
2.	player.py - Función que desacelera el vehículo.	8
3.	player.py - Rotación del vehículo.	9
4.	player.py - Actualización del vehículo.	11
5.	world.py - Ejemplo de creación de las pistas.	12
6.	controller.py - Función que recibe todos los input del teclado.	15

7.	player.py - Método responsable de los cambios en el vehículo.	20
A.1.	Código CGI en php para la actualización y descarga de puntajes	31

1. Justificación del proyecto

Epic Racing consiste en un juego de carreras, en donde el usuario (jugador) puede conducir un coche a través de un circuito bien definido, intentando batir tiempos de vuelta predefinidos para cada pista.

El proyecto nace como fruto de una motivación personal por aprender el desarrollo de videojuegos, lo que conlleva el estudio del patrón de diseño, planteamiento de las mecánicas, diseño gráfico, entre otras varias áreas del conocimiento. Se decidió además implementar el juego sin usar ningún motor gráfico ni físico, solo una librería de dibujo en python. Esto permite entender mejor cómo se debe estructurar una aplicación *from scratch*, definiendo tanto la mecánica de interacción del usuario como la respuesta del juego en sí.



Figura 1: Escena típica de *Epic Racing*.

La industria de los videojuegos ha presentado un gran auge en las últimas décadas [2], [3], siendo una área muy lucrativa dentro del desarrollo de software. Es por ello que este proyecto además pretende ser una primera aproximación al área, sembrando las bases de lo que puede ser un desarrollo de un producto real en el mercado.

2. Introducción

En el contexto del proyecto de Desarrollo de Aplicaciones Multiplataforma se ha optado por realizar un proyecto personal, llamado “Epic Racing”, que consiste en un juego de carreras donde el jugador puede conducir un vehículo a través de un circuito bien definido, intentando batir tiempos de vuelta predefinidos.

El propósito es construir un juego *from scratch* en python, usando el patrón de diseño MVC (Model-View-Controller) [4] que permite arquitecturar aplicaciones gráficas sencillas con facilidad [5]. El proyecto se contempló considerando dos etapas básicas, (1) el levantamiento de un prototipo, que corresponde a la mínima unidad funcional de la aplicación para la evaluación de la mecánica del juego, y (2) programación de extensiones al software, añadiendo características adicionales para incrementar la experiencia del usuario.

El prototipo considera un juego en 2D, con una vista *top-view*, en donde el jugador puede mover el vehículo a través de un circuito cerrado. La mecánica del juego es sencilla y las pistas limitadas. Posteriormente como extensión se añadieron elementos como decoraciones, una interfaz gráfica, una plataforma de almacenamiento de puntajes online (scoreboard) y sonidos para lograr un producto lo más completo posible.

En los siguientes capítulos del documento se detallará el desarrollo tanto del prototipo como las extensiones, explicando las decisiones tomadas durante el proceso y las principales mecánicas implementadas para lograr a cabo del producto.

Nota importante: *Uno de los requerimientos del proyecto era su realización desde el punto de vista de una empresa que va a vender su producto. He de decir que he convalidado las prácticas del curso y que mi caso va ser el de un programador anónimo que ha programado un juego que va estar disponible en Internet, gratuitamente bajo licencia GPLv2 (GNU General Public Licence). Por lo tanto, no considero necesario hacer un gráfico de presupuesto ya que no va generar ningún ingreso. Los recursos necesarios para la realización de este proyecto ha sido un ordenador, Internet, conocimientos de programación y tiempo.*

3. Objetivos

El objetivo del proyecto es lograr un juego completo de carreras, en donde el jugador pueda elegir entre varios tipos de vehículos y pistas para poder lograr los mejores tiempos en cada circuito. El juego tiene que ser lo más autocontenido posible, debe ser fácil de usar y debe funcionar en varios sistemas operativos, ya sea Windows, Linux o MacOS.

Cada parte del proceso de desarrollo, tanto el levantamiento del prototipo o extensiones, consideraron los siguientes objetivos:

1. Prototipo:

- Programación de una aplicación gráfica interactiva, en donde el jugador puede mover, usando el teclado del ordenador, un vehículo a través de un circuito.
- El movimiento del vehículo debe estar gobernado por una mecánica de físicas no realistas, pero que logren una sensación de dificultad al jugador.
- La aplicación gráfica debe poseer una interfaz de usuario amigable mediante el uso de menús, diálogos y alertas.

2. Extensiones:

- El juego se puede extender a través de un paquete de características adicionales que contemplen circuitos y distintos tipos de automóviles.
- Implementación de una mecánica de sonidos y efectos visuales atractivos al jugador para incrementar la experiencia final.
- Para entregar al usuario una sensación de competitividad el producto debe contemplar un sistema de puntajes online, en donde se compara el desempeño de cada jugador en una determinada pista.
- El software debe ser configurable de manera fácil por el usuario, para ello se contempló el diseño de un menú de opciones gráficas.

4. Desarrollo

4.1. Desarrollo del prototipo

4.1.1. Arquitectura de diseño MVC

En cualquier desarrollo de software medianamente complejo si se carece de un patrón/esquema de diseño de software es muy fácil perder el sentido de la orientación sobre lo que se desarrolla, lo anterior es muy grave dado que el manejo de la información y de los objetos puede ser perjudicial.

Para el desarrollo de “Epic Racing”, haciendo hincapié en lo anterior, se optó por el uso del patrón de arquitectura de software MVC (modelo, vista, controlador), el cual permite una comunicación entre usuario y modelo a través de un controlador que administrar el *input* del usuario (teclado) y una vista que permite mostrar, ilustrar, o reflejar el estado de los modelos, en algún canal *output* como lo es la pantalla y altavoces del ordenador. Este patrón es muy importante dado que para desarrollar un videojuego es necesario tener una comunicación casi en tiempo real entre lo lógico y el usuario.

Como lenguaje de programación se escogió a **python** por su facilidad de uso, inherente diseño basado en objetos, fácil ejecución multiplataforma y simplicidad en la instalación y uso de librerías. Como librería gráfica se escogió a **pygame** [7] por su facilidad de uso en comparación a otras como *glfw* o *PyOpenGL*.

Como prototipo el juego debe tener dos grandes modelos: el vehículo y la pista. El vehículo será una entidad lógica (muchas veces conocida como *Actor*) que básicamente posee cuatro propiedades: una textura, una cierta velocidad, posición y aceleración. La pista es una colección de texturas en una cierta región de límites conocidos, dichas texturas pertenecerán a dos grandes sub-conjuntos: las texturas de los caminos (la que en conjunto forman un circuito), y las decoraciones, como árboles, rocas, etc.

Dichos modelos tendrán una comunicación mutua, el jugador (automóvil) debe conocer lo que está “pisando”, por lo que necesita una referencia a la pista; y la pista necesita la posición del vehículo para poder ser representada en pantalla, por lo tanto necesita de una referencia a vehículo. Adicionalmente se necesita de una plataforma en la que se almacenen las texturas para ambos modelos, por lo que se introduce un objeto central denominado **World** (o mundo) el cual instanciará tanto al jugador, como a la pista y permitirá las comunicaciones mutuas.

Dado ese manejo de los modelos es fácil implementar tanto la vista como el controlador; ambos operarán sobre la información de *World*, el controlador exclusivamente modificará el modelo del vehículo (objeto **Player**), y posteriormente el modelo del vehículo modificará el modelo de la pista (objeto **Track**). Y la vista imprimirá en pantalla tanto al

jugador como a la pista, ambos pertenecientes a *World*.

La siguiente figura ilustra un concepto básico de la arquitectura planteada:

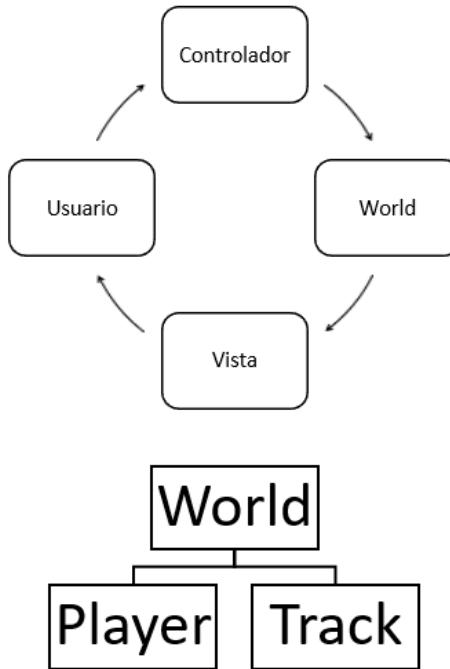


Figura 2: Diseño MVC de la aplicación.

Dado que la pista posee varios objetos los cuales comparten propiedades similares como textura, o posiciones, estos fueron agregados a *Track* como **Object**, el cual tiene como información una textura (y su correspondiente dimensión), y una posición.

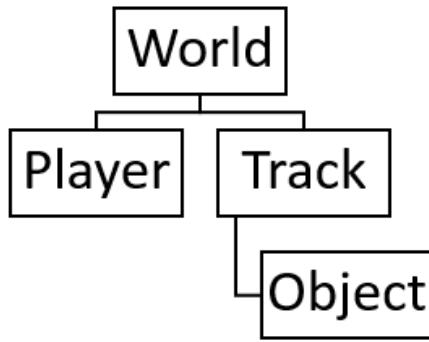


Figura 3: Interacción de los objetos en el mundo.

Una vez que se tienen identificados los objetos principales es necesario implementar sus métodos, lo que significa responder la pregunta siguiente: *¿Qué es lo que hace este modelo?*. La tabla 1 ilustra las principales características de cada clase.

Tabla 1: Características y métodos principales de cada clase principal.

Player	Track	World
<ul style="list-style-type: none"> ■ Acelera ■ Frena ■ Gira ■ Se dibuja ■ Comprueba colisiones 	<ul style="list-style-type: none"> ■ Se dibuja con respecto a la posición del jugador ■ Instancia objetos ■ Retorna objetos (getters) ■ Permite añadir elementos, como marcas 	<ul style="list-style-type: none"> ■ Instancia tanto a <i>Player</i> como a <i>Track</i> ■ Permite crear y cargar diversas pistas ■ Retorna tanto a <i>Player</i> como a la pista que haya cargado

NOTA IMPORTANTE: *Uno de los requisitos para la realización del proyecto era la creación de un diagrama de casos de usos. Debido a que este proyecto contempla una arquitectura MVC no he considerado necesaria la creación de un diagrama de casos de usos puesto que en el punto 4 se explica detalladamente las acciones del programa y a parte se adjunta la figura 2 y figura 3 describiendo gráficamente el concepto básico de la arquitectura MVC y la interacción de los objetos en el mundo.*

4.1.2. Desarrollo de clase Player

Una vez identificadas las responsabilidades de cada objeto es necesario programar los métodos que permitan realizar cada una de las tareas. Sin duda el más complejo de todos los objetos es *Player*; cabe mencionar que sólo se explicarán los métodos más básicos, dado que en realidad *Player* posee 49 métodos, 36 constantes y más de 90 variables de instancia, en +1500 líneas de código ya que es el que más desarrollo lógico tiene. Para lograr sus métodos es necesario tener un par de variables imprescindibles: la textura (la cual, como todas las texturas del juego, fueron cargadas usando una función de pygame), la posición en el mapa (la cual se almacenará convenientemente en dos variables *posx* e *posy*), la velocidad (almacenada en dos variables *velx* e *vely*), una aceleración (la cual será almacenada en una lista de distintas aceleraciones para cada cambio, denominada *acel*), entre otras.

Adicionalmente cada vehículo posee una velocidad de rotación máxima (denominada *maxrotvel*), un agarre a la pista conocida (almacenada en *agarre*), un ángulo de rotación (*angle*), un roce específico sobre la pista (*roce*), una desaceleración (entiéndase por frenando, almacenado en *desaceI*) y una velocidad máxima (*highVelocity*). El juego ofrece tres tipos de vehículos, los cuales poseen distintos valores iniciales de cada variable mencionada:

1. Tipo 1 (vehículo clásico): En este tipo de vehículo la velocidad máxima que puede alcanzar es la más baja, la desaceleración es la más alta, el agarre es alto por lo que

puede tomar las curvas a mayor velocidad sin salir despegado de ella, el roce es bajo por lo que no pierde velocidad rápidamente al dejar de acelerar. Todas estas configuraciones hacen que el tipo de vehículo 1 sea el más fácil para jugar.

2. Tipo 2 (vehículo moderno): Este tipo de vehículo posee una combinación de valores tal que la dificultad de conducción es promedio, por lo que lo hace ideal para jugadores casuales o que no busquen un desafío.
3. Tipo 3 (vehículo súper): Este vehículo puede alcanzar la máxima velocidad, así como también las aceleraciones son comparativamente mayores que los dos tipos anteriores, en desmedro de su velocidad de rotación y de frenado, por lo que exige al jugador saber hasta cuando acelerar y frenar antes de una curva, así como también exige conocer cada tipo de curva para saber donde tomarla, debido a su baja velocidad de rotación, y su baja velocidad de rotación máxima. Por lo que se tiene un tipo de dificultad alto.

Una vez que se tienen todos los valores de las principales constantes de cada vehículo es necesario definir las físicas involucradas en el juego, las cuales no serán realistas (dada la complejidad); todas las funciones implementadas son de carácter heurístico, se construyeron para ofrecer una experiencia de juego lo más divertida posible a la vez que desafiante en un juego de prueba y error hasta dar con el mejor resultado, tanto en jugabilidad como dificultad.

Si pos_x e pos_y son la posición del jugador en un cierto instante de tiempo t en la coordenada x e y entonces se tiene que la posición completa es $(\text{pos}_x, \text{pos}_y)$, si el vehículo acelera para cada instante de tiempo lo que es modificado es la velocidad; considerando el ángulo del vehículo θ la ecuación para actualizar la velocidad corresponde a:

$$\text{vel}_x(t_{i+1}) = \text{vel}_x(t_i) + \text{acel} \cdot \text{ecos}(\theta) \cdot t \quad (1)$$

$$\text{vel}_y(t_{i+1}) = \text{vel}_y(t_i) + \text{acel} \cdot \text{esin}(\theta) \cdot t \quad (2)$$

En donde $\text{ecos}(\theta)$ y $\text{esin}(\theta)$ son funciones trigonométricas definidas especialmente para satisfacer el sistema de coordenadas escogido:

$$\text{ecos}(\theta) = \text{sgncos}(\theta) \cdot |\cos(\theta)| \quad \text{en donde} \quad \text{sgncos}(\theta) = \begin{cases} 1 & \text{si } 270 \leq \theta \leq 0 \vee 0 \leq \theta < 90 \\ -1 & \text{si } -90 < \theta < 270 \\ 0 & \text{si } \theta = 90 \end{cases} \quad (3)$$

$$\text{esin}(\theta) = \text{sgnsin}(\theta) \cdot |\sin(\theta)| \quad \text{en donde} \quad \text{sgnsin}(\theta) = \begin{cases} 1 & \text{si } 180 < \theta < 360 \\ -1 & \text{si } 0 < \theta < 180 \\ 0 & \text{si } \theta = 0 \vee \theta = 180 \end{cases} \quad (4)$$

La velocidad, correspondiente al módulo del vector definido por $(\text{vel}_x, \text{vel}_y)$, $\sqrt{\text{vel}_x^2 + \text{vel}_y^2}$, posee una cota superior (`highVelocity`), por lo que el incremento dado la aceleración no

puede superar dicho límite. La función para acelerar corresponde a:

Código 1: player.py - Función que actualiza la velocidad del automóvil.

```
413 def acelerate(self, t):
414     """
415     Acelera el coche (modifica la aceleración).
416     :param t: Tiempo t
417     :return: void
418     """
419     if self.get_vel() < self.maxvel:
420         if self.automatic:
421             cambio = self.get_cambio()
422             if cambio == 0:
423                 cambio = 1
424             self.velx += self.acel[cambio] * ecos(self.angle) * t
425             self.vely += self.acel[cambio] * esin(self.angle) * t
426             self.tiempoAcelerando += t
427             self.highVelocity = max(self.highVelocity, self.get_vel())
428     ...
```

Para desacelerar el vehículo es lo mismo que lo anterior salvo que la velocidad disminuye por cada tiempo. La cota inferior es 0, y no posee cota superior (una vez detenido no retrocederá). Así:

Código 2: player.py - Función que desacelera el vehículo.

```
540 def desacelerate(self, acel, t, marks):
541     """
542     Frena el coche.
543     :param acel: Constante de desaceleración
544     :param t: Tiempo t
545     :param marks: Lista de marcas
546     :return: void
547     """
548     prev_vel = self.get_vel()
549     if self.velx != 0:
550         self.velx -= acel * ecos(self.angle) * t
551     if self.vely != 0:
552         self.vely -= acel * esin(self.angle) * t
553     if abs(self.velx) < LOWEST_VALUE_VEL_DESACEL:
554         self.velx = 0
555     if abs(self.vely) < LOWEST_VALUE_VEL_DESACEL:
556         self.vely = 0
557     if self.get_vel() > prev_vel: # Se comprueba que no se este acelerando
558         self.velx = 0
559         self.vely = 0
```

```
560     else : # Si se desacelero
561         ...

```

Una vez se tienen las aceleraciones y desaceleraciones es tiempo de introducir la rotación del vehículo. Esta rotación debe ser en función de una velocidad de rotación (definida para cada tipo de vehículo), un tiempo t (dado por el controlador), y una dirección, la cual será 1 o -1 dependiendo de hacia “donde” se gire. La variación angular crecerá mientras la velocidad sea mayor igual a una cota inferior denominada MINROTVEL (que indica la velocidad mínima de rotación, esto ayuda a que no se pueda rotar si el jugador no posee velocidad), y menor a una cierta cota superior definida por la velocidad máxima de rotación multiplicado por el agarre: esto hace que mientras más agarre más rápido se pueda rotar (intuitivamente). Heurísticamente se logró la siguiente fórmula para cada instante de tiempo t :

$$\theta_+ = \begin{cases} \text{mod} \left(\max \left(\min(DTHETA1, maxangvel), -maxangvel \right), 360 \right) & \text{si } vel < MINROTVEL \\ \text{mod} \left(\max \left(\min(DTHETA2, maxangvel), -maxangvel \right), 360 \right) & \text{si no} \end{cases} \quad (5)$$

En donde:

$$ROTDIR = ROTVEL \cdot t \cdot direccion \quad (6)$$

$$vel = \sqrt{vel_x^2 + vel_y^2} \quad (7)$$

$$DTHETA1 = ROTDIR \cdot \cos \left(\frac{vel \cdot 90}{2 \cdot maxrotvel \cdot agarre} \right) \quad (8)$$

$$DTHETA2 = ROTDIR \cdot \max \left(\cos \left(\frac{vel \cdot 90}{1.3 \cdot agarre \cdot (maxvel - 2 \cdot maxrotvel)} \right), 0.4 \right) \quad (9)$$

La función encargada de la actualización de la velocidad angular, la rotación de las texturas, entre otros, corresponde a `rotate()`. Si bien la fórmula es complicada esta se obtuvo en base a un estudio empírico, se fueron modificando valores y agregando constantes hasta obtener un resultado plausible. Esta función además se encarga de la rotación de la textura del automóvil, modificando las cinco posiciones del vehículo:: las cuatro ruedas y el centro del vehículo. Todas ellas son recalculadas en función del nuevo ángulo y guardadas para el posterior análisis de colisiones.

Código 3: `player.py` - Rotación del vehículo.

```
1195 def rotate( self , direction , t , controller=True):
1196     """
1197     Rota el coche.
1198     :param direction: Dirección de rotación
1199     :param t: Tiempo t
1200     :param controller: Controlador del programa

```

```
1201 :return: void
1202 """
1203     if controller : # Si se mueve el coche mediante el controlador
1204         actual_vel = self.get_vel()
1205         # Si se esta bajo la maxima velocidad de rotacion ->creciente
1206         if MINROTVEL < actual_vel < self.maxrotvel * self.agarre:
1207             self.angle = (self.angle + max(
1208                 min(ROTVEL * t * direction * 1 * cos((actual_vel * 90) / (
1209                     2 * self.maxrotvel * self.agarre)),
1210                     self.maxangvel,
1211                     - self.maxangvel)) % 360
1212             elif self.maxrotvel * self.agarre <= actual_vel: # Si se esta sobre la velocidad
1213                 ↪ de rotacion -> decreciente
1214                 self.angle = (self.angle + max(min(ROTVEL * t * direction * max(
1215                     cos((actual_vel * 90) / (1.3 * self.agarre * (self.maxvel - 2 * self.
1216                     ↪ maxrotvel))),
1217                     0.4), self.maxangvel), - self.maxangvel)) % 360
1218             else:
1219                 return
1220             self.desacelerate(self.roce, t, False)
1221             self.velx = actual_vel * ecos(self.angle)
1222             self.vely = actual_vel * esin(self.angle)
1223             self.lastdirangle = direction
1224
1225             # Se modifican las texturas
1226             self.imagenRotada = pygame.transform.rotate(self.texture, self.angle)
1227             self.imagenRotadaSize = self.imagenRotada.get_size()
1228             self.shadow_texture_rotada = pygame.transform.rotate(
1229                 self.shadow_texture, self.angle)
1230             self.shadow_texture_size = self.shadow_texture_rotada.get_size()
1231
1232             # Modifica la posicion de las ruedas
1233             #
1234             # | R3   R2 |
1235             # B   CE   F
1236             # | _R4_____R1_|_
1237             #
1238             w1 = self.width / 2
1239             h1 = self.height / 2
1240             # center_image = self.imagenRotada.get_rect()
1241             self.pos_center = (self.drawablepos[0] + w1, self.drawablepos[1] + h1)
1242             # largo = math.sqrt(((self.imagenRotadaSize[0] / 2) * 0.75) ** 2 + ((self.
1243             ↪ imagenRotadaSize[1] / 2) * 0.7) ** 2)
1244             pos_rueda1 = (
1245                 self.pos_center[0] + 2 * w1 * 0.75,
1246                 self.pos_center[1] + h1 / 2 * 0.8)
```

```

1244     pos_rueda2 = (
1245         self .pos_center[0] + 2 * w1 * 0.75,
1246         self .pos_center[1] - h1 / 2 * 0.8)
1247     pos_rueda3 = (
1248         self .pos_center[0] - 2 * w1 * 0.75,
1249         self .pos_center[1] - h1 / 2 * 0.8)
1250     pos_rueda4 = (
1251         self .pos_center[0] - 2 * w1 * 0.75,
1252         self .pos_center[1] + h1 / 2 * 0.8)
1253     self .pos_rueda1 = rotate_point(pos_rueda1, self .pos_center,
1254                                     360 - self .angle)
1255     self .pos_rueda2 = rotate_point(pos_rueda2, self .pos_center,
1256                                     360 - self .angle)
1257     self .pos_rueda3 = rotate_point(pos_rueda3, self .pos_center,
1258                                     360 - self .angle)
1259     self .pos_rueda4 = rotate_point(pos_rueda4, self .pos_center,
1260                                     360 - self .angle)

```

Una vez el vehículo acelera, frena y gira es necesario agregar un método **update()** que corresponde a una función que se ejecuta en todo tiempo, ésta actualiza la posición del jugador y comprueba su posición en la pista (de ahí que requiere conocer el objeto *Track*).

Código 4: player.py - Actualización del vehículo.

```

1376 def update(self, t):
1377     """
1378     Actualiza la posición usando el tiempo y retorna un mensaje para que view la
1379     → represente en pantalla.
1380     :param t: Tiempo t
1381     :return: Mensaje de estado
1382     """
1383     self .posx += self .velx * t
1384     self .posy += self .vely * t
1385     self .desacelerate( self .roce, t, False)
1386     self .lapTime += t
1387     # Comprobación de la posición del jugador en la pista
... 
```

Nótese que en cada actualización la nueva posición corresponde a la suma entre la posición actual y la ponderación entre la velocidad y el tiempo, a la que posteriormente se aplica un roce para que el vehículo se detenga si el jugador deja de acelerar.

Para determinar si el vehículo está en la pista o no se deben comprobar colisiones, el mecanismo es sencillo: conocidas las cuatro posiciones de las ruedas (las que se actualizan después de girar el vehículo) se comprueba el color del píxel en cada una de dichas posiciones; si dicho color está dentro del rango válido para las pistas (que corresponde al color gris que varía entre 88 y 91) entonces el vehículo está en el asfalto, en caso contrario

está fuera del terreno y debe desacelerarse. Para comprobar el color de los píxeles se usa la función de pygame `get_at()`. Si se quiere, para caso de *debugging*, mostrar en pantalla cada rueda es posible cambiando el parámetro de configuración SHOWRUEDAS dentro del archivo `view.ini` en la carpeta config/ a TRUE.

Para saber si el vehículo sigue correctamente la pista se realiza lo siguiente: se recorre completamente la pista (almacenada en un arreglo, en el objeto *Track*) y se comprueba si la posición del jugador está dentro de los márgenes de cada imagen, si lo está entonces se retorna el índice de dicha “posición” en la pista; y dado que son cerradas, si en un cierto instante de tiempo t el jugador está en la posición i de la pista, en un tiempo $t + 1$ el jugador puede estar en i (lo que indica que no se ha movido) o en $i + 1$, si esto no ocurre quiere decir que el jugador se ha saltado una porción del circuito, lo que es penalizado dentro del juego.

Una vez se tienen todas las variables el objeto vehículo (*Player*) provee de un método para dibujar la textura en una cierta superficie, esta textura siempre se dibujará al medio de la pantalla, rotada θ grados. Adicionalmente se dibujará una textura de sombra (la cual es igual en forma a la textura original, salvo que es de color negro y posee una opacidad de 56 %) escalada en un cierto coeficiente almacenado en una constante SHADOW_PERCENTAGE. Dicha sombra se dibuja debajo de la textura.

4.1.3. Desarrollo de clases Track y World

Para la creación de las pistas (**Track**) se necesita de una variable que almacene las decoraciones (en una lista llamada *decorations*), otra que almacene las pistas (*track*) y sus posiciones (*track_coords*), y adicionalmente se necesita de una variable que almacene el fondo de cada pista (*background*) y el tamaño de la pista (*mapLimits*). El objeto *Track* solo maneja dichas listas, todos sus métodos son tanto “setters” como definir el fondo, los objetivos (los tiempos de vuelta a batir), el título de la pista, agregar decoraciones, el jugador, o las marcas (que se producen al frenar) o métodos “getters” (el que retorna las marcas, los límites, el título, las decoraciones, el fondo, las pistas, etc.).

Por último se tiene al objeto **World**, la cual tiene como métodos el cargar un mapa (con el método *loadMap*) la que toma un índice (correspondiente a cada pista) y agrega, convenientemente, cada elemento de la pista a *Track* utilizando una textura adecuada (dentro de todas las texturas para las pistas posibles, ubicadas en la carpeta resources/images/-track) en una cierta posición usando el método *.addTrack()* de *Track*, se agregan también las decoraciones, títulos, entre otros. Dicho mapa se almacenará en la variable *actualMap* la cual se retornará mediante el método “getter” *.getActualMap()* el cual será llamado intensamente por *Controller* y *View*.

Código 5: world.py - Ejemplo de creación de las pistas.

```
224 def load_map(self, index=None):
225     """
```

```
226     Crea un mapa.  
227     :param index: Indice de la pista a cargar  
228     :return: void  
229     """  
230     ...  
231     # Pista 1 - El origen  
232     if index == 1:  
233         try:  
234             # Se definen los límites del mapas  
235             self.actualMap.set_map_limits(-2500, -2200, 3200, 3200)  
236             # Se definen las vueltas máximas  
237             self.actualMap.set_laps(3)  
238             # Se definen los objetivos del mapa  
239             self.actualMap.set_objetives(  
240                 [(17.5, 19.5, 23.5), (17.0, 19.2, 22.5),  
241                 (15.5, 17.1, 20.0)])  
242             # Se define el fondo del mundo  
243             self.actualMap.set_background(  
244                 self.load_image("grass", alpha=False))  
245             # Se agrega al jugador  
246             self.actualMap.add_car(  
247                 int(self.userConfig.getValue("TYPECAR")),  
248                 self.userConfig.getValue("TEXTURE"),  
249                 True, 0, True,  
250                 self.actualMap.get_track_logic(),  
251                 self.sounds, self.soundsChannel,  
252                 self.checksum, self.scoreConfig,  
253                 self.playerName,  
254                 self.actualMap.get_track_title(),  
255                 self.gameConfig, self.browser,  
256                 rotate=-270,  
257                 verbose=self.verbose)  
258             # Se agregan decoraciones  
259             self.actualMap.add_decoration(  
260                 self.load_image("tree0", alpha=True), (1650, 0))  
261             self.actualMap.add_decoration(  
262                 self.load_image("tree0", alpha=True), (1400, -250))  
263             self.actualMap.add_decoration(  
264                 self.load_image("tree0", alpha=True), (1440, 300))  
265             self.actualMap.add_decoration(  
266                 self.load_image("tree0", alpha=True), (800, 100))
```

En cuanto a recursos todos éstos se cargan una sola vez, utilizando una *lazy function* perteneciente a *World*. Las imágenes se almacenan en un *hashMap* llamado *images* y los sonidos en una lista llamada *sounds*. Luego al crear, por ejemplo, un vehículo, se pasan los valores por referencia de dichas variables a sus respectivos constructores.

4.1.4. Desarrollo del controlador

Para el controlador (**Controller**) se dispone, en el constructor, de la referencia al objeto *World* creada en el *main class*; el principal método de *Controller* es *event_loop* el cual recoge los eventos de la entrada estándar del teclado (único medio de comunicación usado). Cada tecla tiene asociada un cierto evento sobre *World*, en particular sobre el vehículo almacenado en *actualMap*. Para evitar repetir código se guardara directamente la referencia al objeto vehículo en la variable de intancia *player* la que usa el método “getter” de *Track*, *getPlayer()*, y dado que la pista pertenece exclusivamente al mapa actual entonces *player=world.getActualMap().getPlayer()*. Una vez se tienen todas las referencias importantes se realiza lo siguiente para cada tecla pulsada:

Tabla 2: Eventos controlador del automóvil.

Tecla	Objeto a modificar	Método utilizado
Flecha arriba / W	player	player.accelerate(time)
Flecha abajo / S	player	player.desaccelerate(player.getDesacel(), time)
Flecha izquierda / A	player	player.rotate(1,time)
Flecha derecha / D	player	player.rotate(-1,time)
Backspace	player	player.returnToTrack()

El controlador además de modificar el jugador permite modificar el estado del menú del juego. El funcionamiento de este es complejo y funciona de forma complementaria a la ejecución del programa. A grandes rasgos existen dos estados de este menú: si no se ha cargado una partida entonces el menú cargado es el menú principal; y si se ha cargado una partida este menú es el de pausa. Para ambos casos el controlador (que posee referencia al objeto **Menu**) realiza lo siguiente:

Tabla 3: Controles del menú.

Tecla	Objeto a modificar	Método utilizado
Flecha arriba	menu	menu.down()
Flecha abajo	menu	menu.up()
Enter	menu	menu.select()
Flecha izquierda	menu	menu.left()
Flecha derecha	menu	menu.right()
Esc	menu	menu.reset(1)

Código 6: controller.py - Función que recibe todos los input del teclado.

```
96 def event_loop(self):
97     """
98     Función que verifica los eventos de entrada
99     :return: void
100    """
101    time = float(
102        self.clock.get_time() / 1000.0 # tiempo que tomo el frame en generarse
103    # Se obtienen los eventos base
104    for event in pygame.event.get():
105        # Si se cierra la ventana (con evento QUIT o ALT-F4)
106        try:
107            if event.type == QUIT or (event.type == KEYDOWN and event.key ==
108                K_F4 and (key[K_LALT] or key[K_RALT])):
109                utils.destroy_process()
110            except:
111                utils.destroy_process()
112            # Si se presiona una tecla
113            if event.type == KEYDOWN:
114                # Se activa el menu de pausa
115                if event.key == K_ESCAPE:
116                    # Cerrar menu
117                    if self.inmenu:
118                        self.inmenu = False
119                        if self.player is not None:
120                            if not self.player.finished_lap():
121                                self.player.sound_unpause()
122                    # Abrir menu
123                else:
124                    if self.player is not None:
125                        if not self.player.finished_lap():
126                            self.player.sound_pause()
127                            self.inmenu = True
128            # Si se esta jugando
129            if self.player is not None and not self.inmenu:
130                if not self.player.finished_lap():
131                    # Limpiar la vuelta
132                    # elif event.key == K_F10:
133                    #     self.player.clear()
134                    # Captura de pantalla
135                    if event.key == K_F3:
136                        try:
137                            fileimg = 'Screenshot_' + str(abs(
138                                hash(utils.generate_random6())))
139                                + '.png'
140                            surfimg = pygame_to_pil_img(
```

```
139         pygame.display.get_surface())
140         surfimg.save(DIR_SAVES + fileimg)
141         if self.verbose:
142             print(self.lang.get(58, fileimg))
143     except:
144         if self.verbose:
145             print(self.lang.get(59))
146     ...
```

Nótese que para cualquier tipo de estado el controlador sólo accede a los objetos mediante sus métodos, este no accede directamente a las propiedades de cada uno de ellos. Se tiene además que dado que el único objeto que opera sobre los modelos es **Controller** y los modelos requieren de un tiempo de renderizado para las funciones físicas es *Controller* el que debe calcular dicho tiempo (y pasar como argumento) por ello es que antes de ver cada evento se obtiene dicho tiempo utilizando el objeto **Clock** de pygame.

4.1.5. Desarrollo de la clase View

Para la vista esta debe obtener el objeto **Menu** (idéntico al de *Controller*), el objeto **World** (el cual siempre accederá a todos sus campos mediante los “getters”), adicionalmente recibirá como argumento el controlador para acceder a los estados que ha inducido el usuario y que son independientes de *World* (ie. pueden existir sin una instancia directa).

La vista, luego de tener toda la información, posee cuatro estados diferentes en los cuales debe dibujar de forma distinta:

1. Se ha cargado una pista y no está en pausa: Dibuja la pista y la información de la vuelta y del vehículo
2. Se ha cargado una pista y está en pausa: Dibuja la pista y el menú de pausa
3. Se ha terminado la pista y está en la ventana resultados: Independiente del estado de pausa se dibuja la ventana de resultados
4. No se ha cargado la pista: Dibuja el menú principal

Si bien el funcionamiento de *View* es complejo en general lo que hace es recorrer cada objeto del juego y llamar al método *.draw()* de cada uno (tanto de *Object*, de *Player*, o de *Menu*) por lo tanto el cómo se dibuja en pantalla es delegado a cada objeto en particular; *View* sólo organiza este proceso.

Como objetos globales se tiene a **Window**, éste permite la creación de la ventana, y maneja tanto lo que es ancho como alto (mediante getters *getWidth* y *getHeight*), es intensamente usado por *View* (para manejar las coordenadas de cada elemento a dibujar en pantalla), por *Controller* (para alterar el estado de la ventana si es que

se elige dicha opción), es usado además por *Menu* (dado que hay funciones que operan sobre Window, y el método `.draw()` de los menús necesita saber las coordenadas de la pantalla) y también es usado por *Player* (dado que el vehículo siempre se dibuja al medio de la ventana).

4.1.6. Estructura de archivos

Para preservar el orden de los archivos, manejar los recursos (permitir una fácil carga de ellos), y manejar ciertos elementos base como el idioma, las configuraciones o la ubicación dinámica de los directorios se dividió la aplicación en módulos. Esto permite definir una cierta estructura separando por paquetes cada responsabilidad del software.

En la carpeta `bin/` pertenecen los archivos destinados exclusivamente a facilitar la comunicación con el sistema operativo, disponiendo funciones para manejar directorios (como `path`, `utils`, `bindir`, `hashdir`, `noStdOut`), herramientas para manejar recursos (`langs`, `configLoader`, `utils`). Adicionalmente dentro de `bin/` se deben incluir todos los módulos externos a python que se necesiten. Estos se ubican en la carpeta `external/` (en el caso de este juego se requiere de `wconio` y la librería de imágenes de Python, `PIL`, o `Python imaging library`).

Dentro de `config/` deben ir almacenados todos los archivos de configuración usados por el juego. Cada archivo debe mantener una estructura similar, y todos son leídos y cargados por `configLoader`; mantener configuraciones independientes del código es fundamental para organizar grandes proyectos, sólo basta modificar un parámetro para obtener ciertos resultados en vez de entrar a revisar el código. Adicionalmente permite guardar las preferencias del usuario.

Dentro de `data/` deben ir todos los recursos lógicos del programa, ya sea, logs, partidas guardadas, resultados, cache, etc. Para los recursos binarios como fuentes, sonidos o imágenes se destina una carpeta distinta denominada `resources/` la cual posee un script llamado `__scan__.py` el cual analiza recursivamente cada archivo dentro de cada directorio predefinido y almacena las direcciones dinámicas de cada uno de ellos en distintos `__init__.py`, ello produce que cada carpeta como `fonts`, `icons` o `images` es un nuevo módulo el cual a la hora de importar provee de métodos para obtener los archivos dado un cierto índice, el cual puede ser producto de un hash sobre el nombre del archivo, o el nombre del archivo mismo.

Finalmente se tiene el directorio `lib/` el cual contiene todos los archivos lógicos propios del juego.

4.1.7. Prototipo final

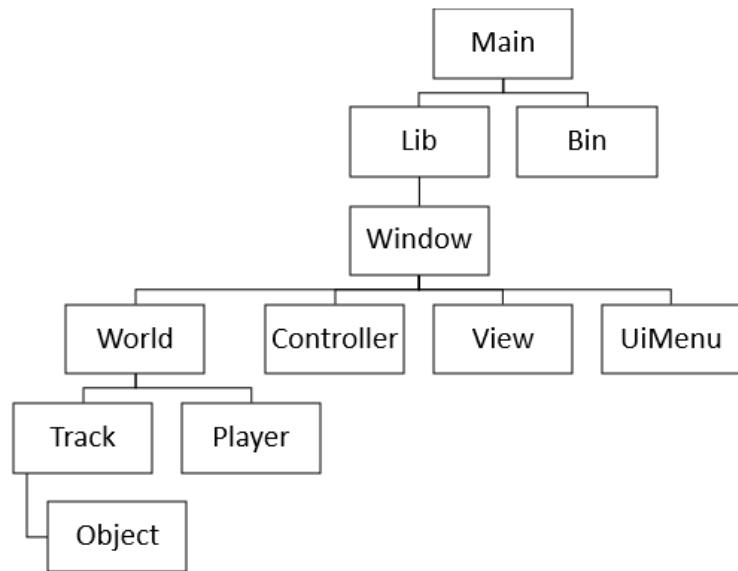


Figura 4: Estructura final sistema de objetos e interacción.

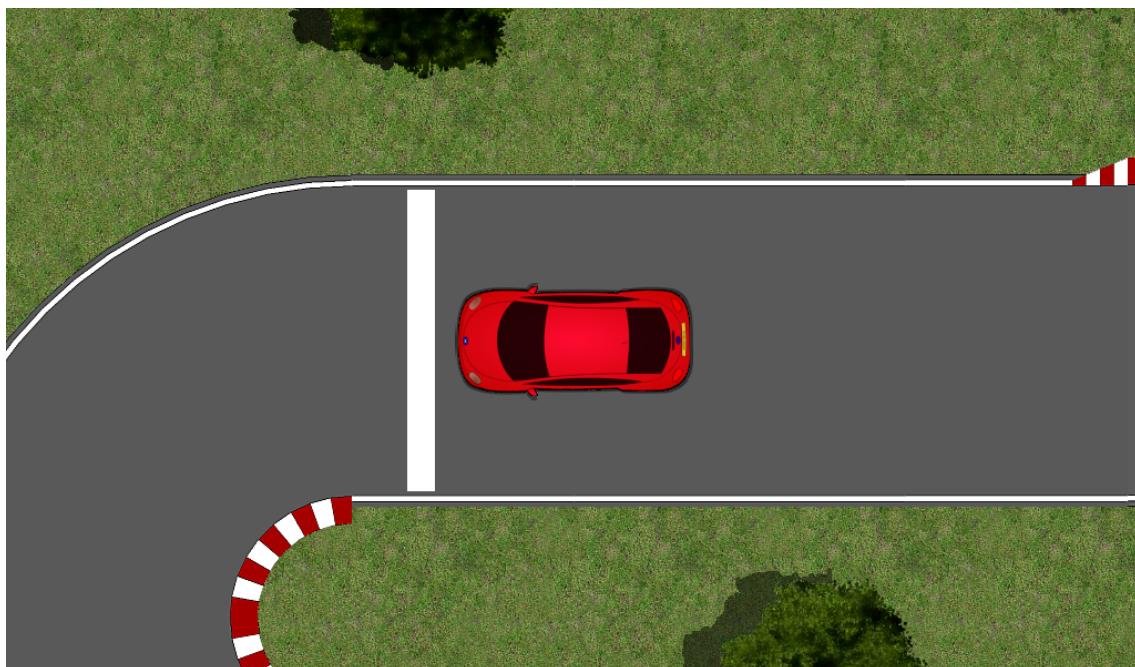


Figura 5: Prototipo de la aplicación.

4.2. Desarrollo de las extensiones

Una vez se tiene el prototipo base se añadieron nuevas extensiones, tales como una gui avanzada, las revoluciones del motor, los sonidos por cada cambio, efectos de frenado, fantasma de mejor vuelta y un scoreboard online. Estos fueron agregándose a medida que el desarrollo del producto avanzaba.

4.2.1. Sistema de revoluciones y marchas del motor

La primera extensión importante es la de los cambios y la revolución del motor. Para realizar lo anterior sin usar físicas involucradas se optó por manejar una lista de velocidades máximas asociadas a cada cambio. Dicha lista se almacena en una variable de instancia de la clase *Player* denominada *velcambios*. La idea base es que, si el jugador acelera o frena, y la velocidad instantánea del vehículo se mantiene dentro de un determinado rango, entonces la marcha del motor no cambia, en cambio, si al acelerar la velocidad es superior al límite del rango entonces la marcha aumenta. Este cambio es automático, sin embargo se puede añadir el cambio manual como una *feature* adicional.

Dado que cada cambio posee una distinta velocidad máxima es intuitivo que cada cambio posea una aceleración distinta (y es lo que pasa en realidad), por lo que la respuesta del vehículo varía según la marcha en la que está, haciendo más difícil y desafiante la conducción.

Para añadir más realismo al juego cada tipo de vehículo tiene un número de marchas máximo distinto, mientras el vehículo sea más lento éste tiene menos cambios; para el primer tipo (el clásico) el número de cambios es 4, para el moderno es 5 y para el súper es 6.

En teoría, cada marcha exige de manera distinta al motor, por lo que para reproducir este fenómeno se introdujo un sonido distinto de aceleración o frenado según el cambio. Estos sonidos fueron creados a partir de un sonido de motor de una moto (dada su frecuencia y pitch característicos). Mientras mayor sea el cambio menor será la tonalidad del sonido, para el primer cambio se tendrá una tonalidad más aguda (por ejemplo, #A) y para el sexto cambio se tendrá un tono #F; tanto amplitudes como longitudes de ondas fueron modificadas para dar el efecto deseado; todos estos cambios fueron creados usando el programa *Audacity*.

Código 7: player.py - Método responsable de los cambios en el vehículo.

```
684 def get_cambio(self, v=None):
685     """
686     Retorna la marcha del vehículo.
687     :param v: Indica si la velocidad es en kph o la interna
688     :return: Integer entre 0 1 inf+
689     """
690
691     if v is None:
692         vel = self.get_vel_kph()
693     else:
694         vel = v
695     if vel == 0:
696         new_cambio = 0
697     else:
698         cambio = 1
699         for velc in self.velcambios:
700             if vel >= velc:
701                 cambio += 1
702             else:
703                 break
704     new_cambio = min(cambio, self.maxcambio - 1)
705
706     # Se cambia el sonido si se modificó el cambio
707     if new_cambio != self.lastcambio:
708         # Se detiene el sonido anterior
709         self.soundGearChannel.stop()
710         # Si se acelera
711         if self.tiempoAcelerando > 0:
712             if self.sound_state:
713                 self.soundGearChannel.play(self.soundGear[new_cambio + 1],
714                                         -1)
715             self.soundGearPlaying = 1
716         # Si se desacelera
717         else:
718             if self.sound_state:
719                 self.soundGearChannel.play(self.soundGearR[new_cambio + 1],
720                                         -1)
721             self.soundGearPlaying = -1
722         self.lastcambio = new_cambio
723     # Si no ha pasado el cambio
724     else:
725         # Si se está acelerando pero se tiene el sonido de bajar marcha se cambia
726         if self.tiempoAcelerando > 0 and self.soundGearPlaying == -1:
727             self.soundGearChannel.stop()
728             if self.sound_state:
```

```
728         self .soundGearChannel.play(
729             self .soundGear[self.lastcambio + 1], -1)
730         self .soundGearPlaying = 1
731     elif self .tiempoAcelerando == 0 and self.soundGearPlaying == 1:
732         self .soundGearChannel.stop()
733     if self .sound_state:
734         self .soundGearChannel.play(
735             self .soundGearR[self.lastcambio + 1], -1)
736         self .soundGearPlaying = -1
737     if self .automatic:
738         return new_cambio
739     else :
740         return self .cambio
```

Para cada cambio adicionalmente se tiene una distinta revolución de motor, las revoluciones funcionan de la siguiente manera: mientras la velocidad del vehículo crezca y no cambie de marcha entonces la revolución del motor aumenta, en cambio, si la velocidad disminuye las revoluciones disminuyen. Adicionalmente para cada cambio se tendrá un 15 % de uso de motor base, ello evita que se tenga una revolución de 0 % si es que la velocidad del vehículo es igual a la velocidad base del ultimo cambio (lo que no es correcto).

Aunque esta forma de calcular las revoluciones no es físicamente correcta permite un efecto muy profesional a la hora de dibujarlo en pantalla. Para reflejar las revoluciones se creó un objeto **revolGraph** el cual dibuja un triangulo rectángulo de altura y anchura determinados, instanciado por *View* que toma un cierto valor *x*: si *x* es 0 % entonces el triangulo no se dibuja, y si es igual a un 100 % entonces se dibuja el triangulo completo. Además dicho triangulo posee un gradiente de colores el cual a mayor altura el color tenderá mas a amarillo, en cambio a menor altura dibujada será mas rojo. Las tonalidades también son tomadas por argumento.

Dado que se posee toda la información del vehículo esta se representa en pantalla en el objeto *View*, para ello en la esquina inferior izquierda se dibuja un sprite rectangular de color gris (para añadir contraste), encima se dibuja la velocidad en KM/H del vehículo (para obtener dicha velocidad se realizó una transformación lineal de la velocidad interna, multiplicando por una constante obtenida empíricamente), en la esquina inferior derecha se dibuja el triángulo de las revoluciones del motor (llamando al método *.draw()* de *revolGraph*), en la esquina superior izquierda se dibujar los tiempos de vuelta definidos en *World* y en la esquina superior derecha se dibuja el tiempo de vuelta actual y las vueltas completadas. Todo lo anterior se intentó desarrollar de la forma más pulcra y estéticamente agradable posible.

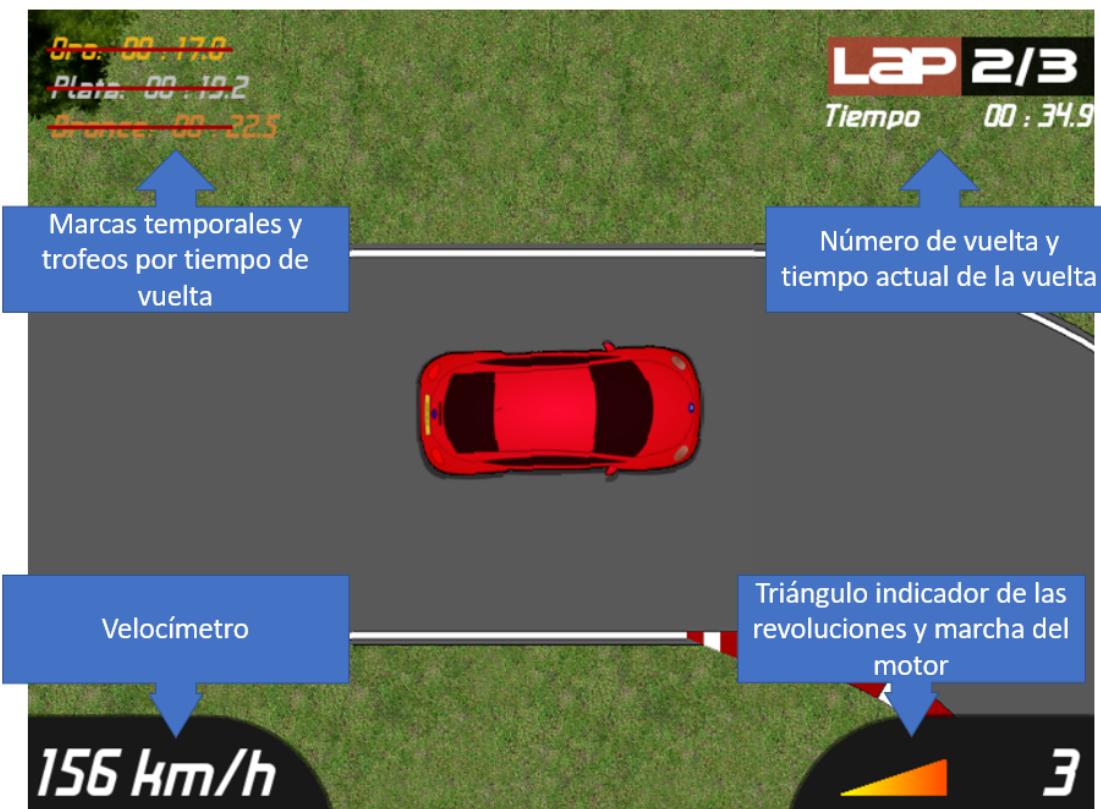


Figura 6: Componentes de la interfaz gráfica del juego, velocímetro, número de vuelta y contador de revoluciones.

4.2.2. Efectos de frenado y fantasma de mejor vuelta

Adicional a los cambios se incluyeron efectos de frenado y fuera de pista: Si el vehículo frena se reproduce un sonido en forma de *loop* (dicho sonido fue obtenido del internet), y adicionalmente realiza una marca en el suelo a partir de la posición de las dos ruedas traseras (posiciones conocidas al rotar el vehículo). Si el vehículo se sale de la pista en las cuatro ruedas se dibujan líneas de colores de tonalidades cafés (los cuales se eligen de forma aleatoria), y, al igual que el caso del frenado, se reproduce un sonido de "tierra" en forma de *loop*.

Para dar una sensación de competitividad local se añadió la extensión de mostrar el fantasma del jugador correspondiente a la mejor vuelta, para obtener esto para cada vuelta se agregan las posiciones y el ángulo de rotación del jugador en cada instante de tiempo t , luego, al finalizar la vuelta si el tiempo de dicha vuelta es menor que el tiempo de vuelta mínimo entonces dicha colección de información (guardada en una lista llamada `ghost_actual`) se despliega en pantalla al llamar a `draw()`.



Figura 7: Marcas de frenado y fantasma de mejor vuelta.

4.2.3. Panel de visualización de resultados y scoreboard en línea

Para la ventana de resultados se dibujan las estadísticas del circuito (los que se actualizan en `.update()` de *Player*), además se dibujan los objetivos (en verde o rojo dependiendo de cómo el jugador los complete), y el *scoreboard* de dicha pista con la posición del jugador con respecto al resto de jugadores en el mundo. En el anexo se presenta en más detalle las principales características de la base de datos y la comunicación con el servidor.

Para el desarrollo del *scoreboard* se tiene una tabla en una base de datos alojada en un servidor, el cual posee como campos el nombre del usuario (`username`), un hash del nombre de la pista (un identificador único), la fecha del registro y el puntaje mismo, el cual se calculó con la siguiente fórmula:

$$\text{puntaje} = 100000 \cdot \left(\frac{0.3}{tiempoFuera+1} + \frac{0.4}{\sqrt{\frac{\sum_{i=1}^{vueltas} t_{vuelta}(i)}{\min(\cup_{i=1}^{vueltas} t_{vuelta}(i))} + 1 - vueltas}} + \frac{0.3}{\min(\cup_{i=1}^{vueltas} t_{vuelta}(i))^{1.5}} \right) \quad (10)$$

De la fórmula anterior *tiempoFuera* indica la cantidad de tiempo en el que el jugador ha pisado la tierra, $t_{vuelta}(i)$ indica el tiempo de la vuelta i -ésima y *vueltas* indica la cantidad de vueltas del circuito. De la formula se obtiene que el puntaje máximo esperado es 100000, y mientras más el jugador se salga de la pista menos puntaje obtendrá; además si cada vuelta es parecida al resto entonces obtendrá más puntaje. Adicionalmente se da puntaje

por el tiempo de vuelta más rápido ($\min(\cup_{i=1}^{vuelta} t_{vuelta}(i))$).

Una vez que el jugador termina la vuelta se calcula el puntaje y este se sube al servidor mediante una consulta \$GET, como parámetro extra se envía el hash de los archivos del juego (utilizando la función `path_checksum`, la cual busca recursivamente archivos dentro de un directorio base, para el cual se crea una cadena md5 correspondiente a cada archivo binario), esto sirve para evitar trampas: si el usuario cambia sólo una línea de `main.py`, o de cualquier clase de `bin/` o de `lib/` entonces su puntaje no se subirá a la red y se retornará el mensaje "juego modificado, marcadores desactivados". Dichos puntajes una vez subidos estos se retornan en formato html, lo cual se lee de forma plana. Para ordenar los puntajes de forma decreciente se utiliza SQL, en especial ORDER BY.



Figura 8: Panel de resultados y scoreboard de mejor vuelta.

4.2.4. Sistema de menús

Se programaron dos tipos de menús, un menú textual (utilizado en “Ayuda” como “Acerca De”). Y un menú de sólo cambios elegibles (Menu).



Figura 9: Menú principal del juego.

Utilizando la clase menú se desarrolló, en forma de extensión, un menú de configuraciones en donde se incluyen las siguientes opciones:

- Cambiar de idioma (todos creados usando la herramienta del sistema), por defecto vienen dos idiomas, español e inglés. Crear idiomas es muy fácil, y añadirlos al juego también (basta con pegar el archivo de idioma en la carpeta de recursos y añadir la entrada en el menú).
- Cambiar el tamaño de la ventana, a formato 16:9 (widescreen) o 4:3, todos los tamaños son calculados por la clase **Window**.
- Cambiar a formato fullscreen / windowed (funciones provistas por **Window**).
- Mostrar o ocultar el fantasma (funciones operables sobre **Window**).
- Mostrar o ocultar los fps del juego en el título (funciones provistas por **Window**).
- Activar / desactivar los sonidos (funciones provistas por **View**, **UIMenu** y **Player**).

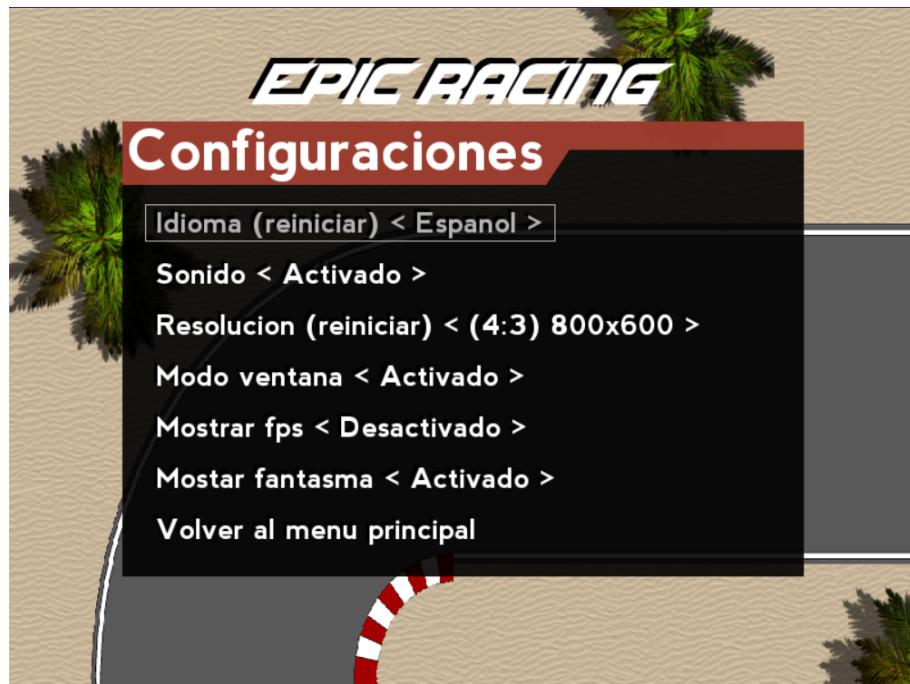


Figura 10: Menú de opciones.

Esto fue desarrollado dado que es importante ofrecer al cliente la opción de poder modificar los parámetros del producto en función de su hardware (configuraciones relacionadas con la vista), de sus preferencias personales (por ejemplo, lo del fantasma, o los fps, el sonido) o de su cultura (por ejemplo, el idioma).

Para el menú de juego se incluye una opción para cambiar el tipo de vehículo (de entre tres opciones predefinidas y extensibles), para cada tipo de vehículo se ofrecen distintos colores (los cuales pueden responder a las preferencias de cada jugador) y existe la opción de cambiar la pista a jugar (que es lo más importante).

Por defecto el juego viene con cuatro pistas elegibles, estas son muy extensibles, es fácil añadir nuevas pistas, pero toma tiempo desarrollarlas.

Todas las extensiones añadidas tienen como objetivo mejorar la experiencia del usuario a la hora de jugar, si alguna característica impidiese lo anterior siempre es mejor evitarla. El producto también fue desarrollado teniendo en mente la extensión, pudiendo agregar más texturas, diferentes fondos de pista, diversos colores de vehículo, diversos tipos de vehículo, etc.

Finalmente se obtuvo el siguiente esquema del proyecto:

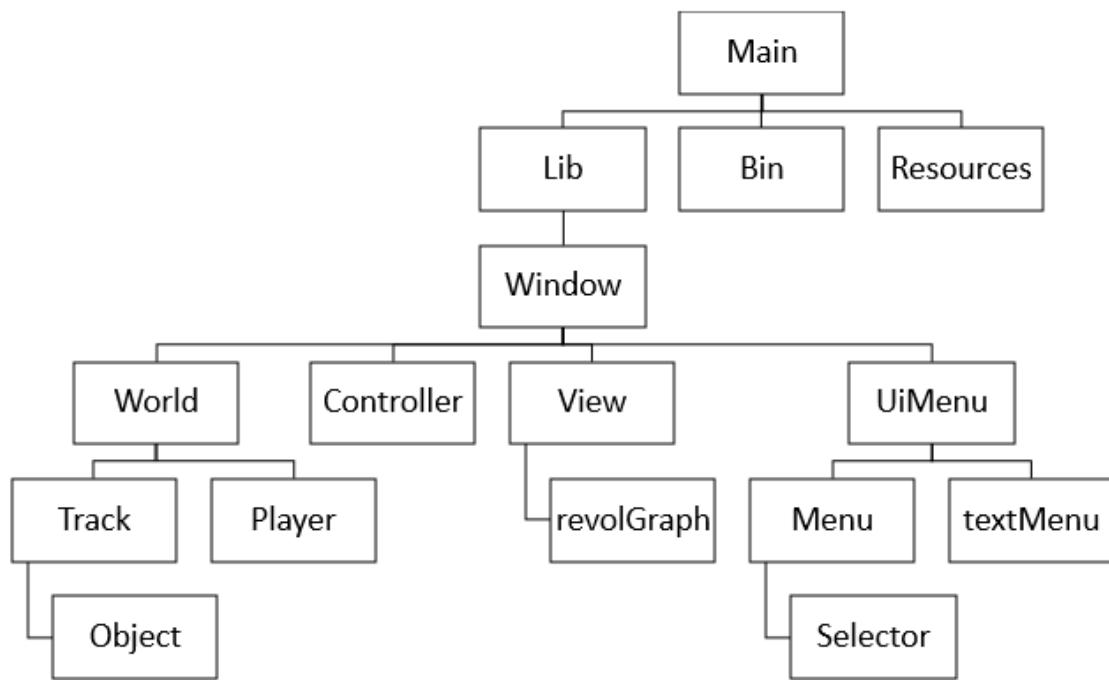


Figura 11: Esquema final de la aplicación.

5. Conclusiones

Como primera conclusión después de haber desarrollado *Epic Racing* es que desarrollar juegos que se cataloguen como un buen producto demanda mucho tiempo y una gran capacidad de organización, así como también una gran creatividad a la hora de plantear en código lo que se tiene en mente, aun así, no es tarea difícil.

Tal vez la parte más difícil del desarrollo es saber qué hacer, el universo de posibilidades siempre es infinito, pero hay que considerar los factores que, en general, afectan a todas esas posibilidades:

1. El tiempo.
2. Los recursos que se disponen, tanto humano como monetario.
3. El hardware.
4. El conocimiento que se tenga en dicha área.

Una vez se tienen las ideas en mente es hora de organizarlas, definiendo convenientemente las responsabilidades de cada objeto y manteniendo siempre un diseño de software que mejor permita el buen desarrollo del producto (por ejemplo MVC).

Luego viene la parte lógica del problema, la cual debe ir complementándose con la parte gráfica, la cual es una tarea sumamente importante, saber escoger tanto sonidos, como fuentes, o imágenes no es tarea fácil, puede demandar mucho tiempo y recursos. Es vital saber qué agregar y qué no, a lo largo del desarrollo ocurren muchas ideas como "qué tal si agrego *x* cosa, no creo que sea difícil"; si se mantuvo una buena organización del código no debiese ser exigente en dificultad, pero es exigente en tiempo si se tiene como propósito desarrollar algo bueno. Por lo mismo muchas características que quise agregar antes del plazo de la tarea no pudieron ser completados, como un editor de mapas, o la opción de multijugador (con fantasmas).

Referencias

- [1] Ejemplo tesis. Universidad de Cuenca.
<https://dspace.ucuenca.edu.ec/bitstream/123456789/2315/1/tps616.pdf>.
- [2] Antonio Checa Godoy. *Hacia una industria española del videojuego*. Universidad de Sevilla. http://revistacomunicacion.org/pdf/n7/articulos/a12_Hacia_una_industria_espanola_del_videojuego.pdf
- [3] ABC.es. *La industria del videojuego: entre las grandes superproducciones y el auge de lo «indie»* https://www.abc.es/tecnologia/videojuegos/abci-industria-videojuego-entre-grandes-superproducciones-y-auge-indie-201808240057_noticia.html
- [4] Code Project. Simple Example of MVC (Model View Controller) Design Pattern for Abstraction.
<https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design>
- [5] Martin Fowler, 2006. *The evolution of MVC and other UI architectures*.
<https://martinfowler.com/eaaDev/uiArchs.html>
- [6] Python Programming Language.
<https://www.python.org>
- [7] PyGame: A Primer on Game Programming in Python.
<https://realpython.com/pygame-a-primer/>
- [8] Overleaf, Editor de Latex Online <https://site.ieee.org/ecuador/2016/02/17/overleaf-otra-excelente-herramienta-gratis-con-tu-membresia-ieee/>

Anexo A. Bases de datos

Para almacenar el puntaje de cada jugador se creó una base de datos alojada en un servidor remoto gratuito¹. Esta aplicación web consiste en un *cgi* en *php*, y una tabla en SQL con los siguientes campos:

Tabla A.1: Campos de la tabla de la base de datos.

Campo	Tipo	Comentario
player	varchar(65)	Nombre del jugador
fecha	varchar(60)	Fecha del ingreso
pista	varchar(32)	Identificador de la pista
puntaje	int(11)	Puntaje del jugador en la partida
tiempo	float	Tiempo del jugador en la partida (seg)
qkey	varchar(15)	Hash de los datos del jugador
modelo	int(11)	Modelo del vehículo (clásico, moderno, super)

NOTA IMPORTANTE: Otro requisito para la realización del proyecto era la creación de cinco tablas relacionales en una base de datos. Dada la estructura de este juego, no es necesaria la creación de más de una tabla ya que la única información que se almacena es el hash de datos del jugador, tipo de coche y el nombre del jugador como se explica a continuación.

Un jugador al terminar la partida se genera una comunicación por GET con el script de puntajes del servidor, enviando el hash del identificador de la pista, el hash de los archivos del código fuente del juego (calculados con md5), el tipo de coche (enviado como un número entero entre 1 y 3), el puntaje obtenido con la fórmula (10) y el nombre del jugador como un string. Los hashes se incluyeron para evitar la vandalización de los puntajes, cualquier intento de cambiar el código fuente ejecutable en Python hace que la validación desde la base de datos falle.

El php obtiene toda la información desde `$_GET` (líneas 58-65), entre las líneas 67 y 83 se realizan diferentes validaciones de los datos, imprimiendo mensajes de error para cada uno de los fallos, los cuales son procesados por la aplicación (líneas 133-159), el puntaje del jugador se almacena construyendo la consulta sql (líneas 105-141) y se obtiene desde la base de datos a partir del código de la pista y el tipo de vehículo (líneas 173-195).

Cabe comentar que este cgi no es muy robusto en cuanto a la seguridad, potenciales mejoras pueden ser la conexión y ensamblaje de la consulta SQL con *prepared statements* para evitar ataques de SQL-Injection, o bien realizar una consulta a través de POST con https ya que GET es potencialmente vulnerable. Por último cabe destacar que validar que el nombre del jugador no sea ofensivo para la comunidad (racista, homofóbico, o que incite a cualquier tipo de odio) no es un problema menor, actualmente se usó una *blacklist* de

¹ https://epic-racing.000webhostapp.com/scoreboard_epic.php

614 nombres inválidos, pero es fácilmente atacable usando un patrón distinto, por lo que se recomienda o bien tener una blacklist más grande, el uso de librerías más sofisticadas para el reconocimiento de patrones o bien el poder añadir una opción para denunciar un nombre en específico mediante la misma aplicación.

Código A.1: Código CGI en php para la actualización y descarga de puntajes

```
1 # Función que genera un string aleatorio
2 function generateRandomString($length = 10)
3 {
4     $characters = '0123456789
5 → abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ';
5     $charactersLength = strlen($characters);
6     $randomString = '';
7     for ($i = 0; $i < $length; $i++) {
8         $randomString .= $characters[rand(0, $charactersLength - 1)];
9     }
10    return $randomString;
11 }
12
13 # Configuraciones del servidor
14 date_default_timezone_set("Europe/Madrid");
15 $SERVER_NAME = "localhost";
16 $SERVER_USERNAME = "...";
17 $SERVER_PASSWORD = "...";
18 $SERVER_DATABASE = "epicracing";
19 $SERVER_TABLE = "scoreboard_epic";
20
21 #Constantes del programa
22 $NEWLINE = "<br>";
23 $ERROR_BADPARAMETERS = "ERROR_BADPARAMETERS" . $NEWLINE;
24 $ERROR_FAKEHASH = "ERROR_FAKEHASH" . $NEWLINE;
25 $ERROR_FAKESCORE = "ERROR_FAKESCORE" . $NEWLINE;
26 $ERROR_FAKETIME = "ERROR_FAKETIME" . $NEWLINE;
27 $ERROR_FAKETRACK = "ERROR_FAKETRACK" . $NEWLINE;
28 $ERROR_NOCONNECTION = "ERROR_NOCONNECTION_DB" . $NEWLINE;
29 $MAX_SCORE = 100000;
30 $MAX_SCORE_INDEX = 11;
31 $MIN_SCORE = 1;
32 $NULL_SCOREBOARD = "NO_SCORES" . $NEWLINE;
33 $CORRECT_QUERRY = "OK" . $NEWLINE;
34
35 #Configuraciones globales
36 $_TEST = true;
37
38 #Configuraciones de hash
```

```
39 $SERVER_BINHASH = "";
40 $SERVER_LIBHASH = "";
41 $SERVER_MAINASH = "";
42
43 #Pistas validas
44 $CARS = array(1, 2, 3);
45 $TRACKS = array("B0E54CA67BC76B7FA4E03AF0FD435A0D", "32
    ↪ EF03C546D59DB09DA59E7EE4D38618", "6A396B771F6B07ED807984C9804C5B7B",
    ↪ "6422B538330980F3B78912FE4C83F19F");
46
47 #Se obtiene la accion a realizar
48 if ( isset($_GET["action"])) {
49 $action = $_GET["action"];
50
51 # Si es insertar y obtener la posición
52 if ($action == "insert") {
53 #Se comprueba la validez de los parámetros
54 $valid_parameters = isset($_GET["h1"], $_GET["h2"], $_GET["h3"], $_GET["t"], $_GET[
    ↪ "p"], $_GET["s"], $_GET["mlp"], $_GET["m"]);
55 if ($valid_parameters) {
56 $client_binhash = $_GET["h3"];
57 $client_libhash = $_GET["h1"];
58 $client_mainhash = $_GET["h2"];
59 $client_track = $_GET["t"];
60 $client_playername = $_GET["p"];
61 $client_score = $_GET["s"];
62 $client_minlapttime = $_GET["mlp"];
63 $client_model = $_GET["m"];
64
65 # Se comprueba el hash
66 if ($_TEST or ($client_libhash == $SERVER_LIBHASH and $client_mainhash ==
    ↪ $SERVER_MAINASH and $client_binhash == $SERVER_BINHASH)) {
67 # Se comprueba que la pista exista
68 if (in_array($client_track, $TRACKS)) {
69 # Se comprueba que el puntaje sea correcto
70 if (is_numeric($client_score)) {
71 $client_score = intval($client_score);
72 #Si el puntaje esta dentro del maximo
73 if ($MIN_SCORE <= $client_score and $client_score <= $MAX_SCORE) {
74 # Se comprueba que el tiempo minimo sea un numero
75 if (is_numeric($client_minlapttime)) {
76 $client_minlapttime = floatval($client_minlapttime);
77 # Se comprueba que el tiempo minimo sea mayor a cero
78 if ($client_minlapttime > 0) {
        # Se comprueba que el tipo de auto sea entero y entre 1,3
        if (is_numeric($client_model) and in_array($client_model, $CARS)) {
```

```
81 $client_model = intval($client_model);
82
83 # Se crea una conexión al servidor
84 $conn = new mysqli($SERVER_NAME, $SERVER_USERNAME,
85 → $SERVER_PASSWORD, $SERVER_DATABASE);
86
87 # Comprueba el estado de la conexión
88 if ($conn->connect_error) {
89     print($ERROR_NOCONNECTION);
90 } else {
91
92     # Se obtiene la fecha
93     $consult_time = date("Y-m-d H:i:s");
94     $consult_key = generateRandomString();
95
96     # Consulta sql
97     $sql = "INSERT INTO `{$SERVER_DATABASE}`.`{$SERVER_TABLE}`(`player`, `fecha`, `pista`, `puntaje`, `tiempo`, `qkey`,
98 → `modelo`) VALUES ('{$client_playername}', '{$consult_time}', '{$client_track}', '{$client_score}', '{$client_minlaptime}',
99 → '{$consult_key}', {$client_model});";
100
101     # Si la consulta se realizo correctamente
102     if ($conn->query($sql) === TRUE) {
103
104         # Se obtiene la lista
105         $sql = "SELECT player, puntaje, fecha, qkey FROM {$SERVER_TABLE}"
106 → WHERE pista='{$client_track}' AND
107             modelo={$client_model} ORDER BY puntaje DESC";
108         $result = $conn->query($sql);
109
110         # Se recorren los datos si es que existen
111         if ($result->num_rows > 0) {
112             echo($CORRECT_QUERRY);
113
114             # Se imprimen los datos con respecto al jugador
115             $index = 1;
116             $showed = false;
117             $printed = 0;
118             while ($row = $result->fetch_assoc()) {
119
120                 if ($row["player"] == $client_playername and $row["qkey"] ==
→ $consult_key) {
121                     print("<color>red</color><index>" . $index . "</index><player>" .
→ $row["player"] . "</player><score>" . $row["puntaje"] .
122                         "</score>" . "<fecha>" . $row["fecha"] . "</fecha>" .
```

```
        ↵ $NEWLINE);
121            $showed = true;
122        } else if ($printed < $MAX_SCORE_INDEX) {
123            if (($printed == ($MAX_SCORE_INDEX - 1) and !$showed) or (
124                $index == ($MAX_SCORE_INDEX) and !$showed)) {
125                    print("NULL" . $NEWLINE);
126                } else {
127                    print("<color>white</color><index>" . $index . "</index><player
128                    >" . $row["player"] . "</player><score>" . $row["puntaje"] .
129                        "</score>" . "<fecha>" . $row["fecha"] . "</fecha>" .
130                        "</fecha>" . $NEWLINE);
131                }
132            }
133        } else {
134            print($NULL_SCOREBOARD);
135        }
136    } else {
137        print($ERROR_NOCONNECTION);
138    }
139    $conn->close();
140}
141} else {
142    print($ERROR_BADPARAMETERS);
143}
144} else {
145    print ($ERROR_FAKEETIME);
146}
147} else {
148    print ($ERROR_FAKEETIME);
149}} else {
150    print ($ERROR_FAKESCORE);
151}} else {
152    print ($ERROR_FAKESCORE);
153}} else {
154    print($ERROR_FAKETRACK);
155}} else {
156    print($ERROR_FAKEHASH);
157}} else {
158    print($ERROR_BADPARAMETERS);
159}
160
161 #Si la accion es obtener el scoreboard de una cierta pista
162 } else if ($action == "get") {
```

```
163 # Se comprueba la validez de los parámetros
164 if ( isset($_GET["t"], $_GET["m"])) {
165 $client_track = $_GET["t"];
166 $client_model = $_GET["m"];
167 #Si la pista es válida
168 if (in_array($client_track, $TRACKS)) {
169 #Si el modelo de coche es válido
170 if (is_numeric($client_model) and in_array($client_model, $CARS)) {
171 $client_model = intval($client_model);
172
173 # Se crea una conexión al servidor
174 $conn = @new mysqli($SERVER_NAME, $SERVER_USERNAME,
175                      $SERVER_PASSWORD, $SERVER_DATABASE);
176
177 # Comprueba el estado de la conexión
178 if ($conn->connect_error) {
179 print($ERROR_NOCONNECTION);
180 } else {
181
182 # Consulta sql
183 $sql = "SELECT player, puntaje, fecha FROM {$SERVER_TABLE} WHERE pista='
184                      {$client_track}' AND modelo={$client_model} ORDER BY puntaje DESC";
185 $result = $conn->query($sql);
186
187 # Se recorren los datos si es que existen
188 if ($result->num_rows > 0) {
189 echo($CORRECT_QUERRY);
190
191 # Se imprimen los datos
192 $index = 1;
193 while ($row = $result->fetch_assoc()) {
194     print("<color>white</color><index>" . $index . "</index><player>" . $row["player"] . "
195                      </player><score>" . $row["puntaje"] .
196                      "</score>" . "<fecha>" . $row["fecha"] . "</fecha>" . $NEWLINE);
197     $index++;
198 }
199
200 } else {
201     print($NULL_SCOREBOARD);}}
202 } else {
203     print($ERROR_BADPARAMETERS);
204 } else {
205     print($ERROR_FAKETRACK);
206 } else {
207     print($ERROR_BADPARAMETERS);
208 } else {
```

```
206 print($ERROR_BADPARAMETERS);
207 } } else {
208     print($ERROR_BADPARAMETERS);
209 }
```

Anexo B. Diagrama de Gantt

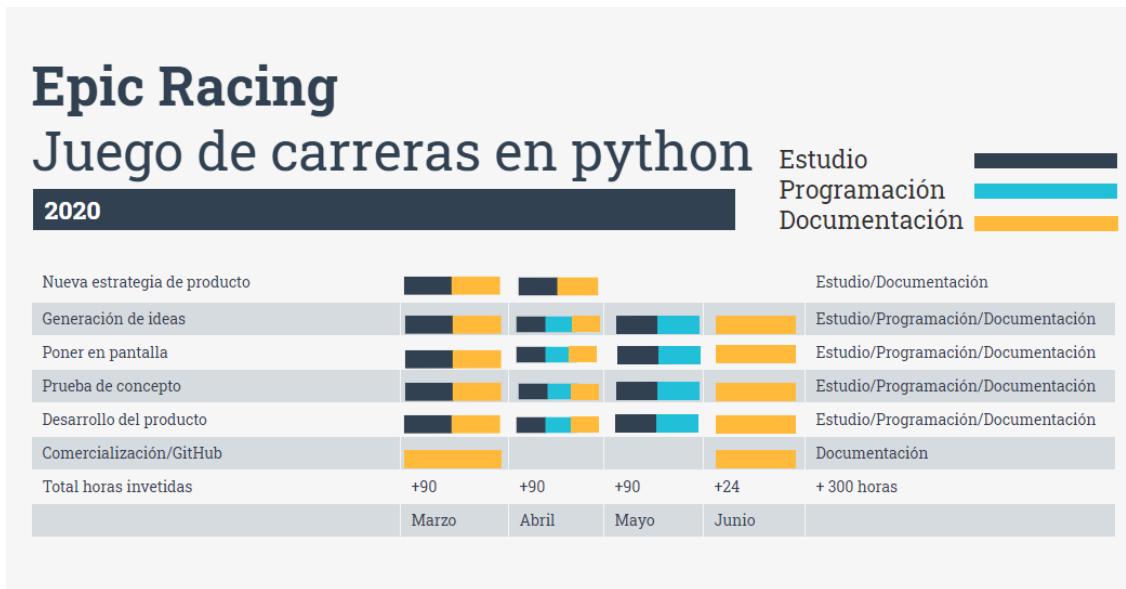


Figura B.1: Diagrama de Gantt

Cabe mencionar que se está trabajando en una nueva versión para que a partir de ella se pueda crear un ejecutable del juego para que el usuario no tenga que descargar toda la carpeta del proyecto, ni instalar python en su ordenador y tampoco usar la terminal, ya que para algunas personas puede llegar a ser algo tedioso.

Anexo C. Manual de usuario

Para poder ejecutar el juego es necesario tener instalado el intérprete de python, entonces, el primer paso sería descargar e instalar Python 2.x

[Link de descarga y tutorial de instalación.](#)

Una vez instalado python, abrimos el símbolo de sistema (CMD) desde el buscador de Windows y nos movemos a la carpeta del juego donde se encuentra el archivo main.py con el comando **cd**. Para hacer esto se escribe el comando y de seguido la ruta del archivo.
[Tutorial comando CD en Windows.](#)

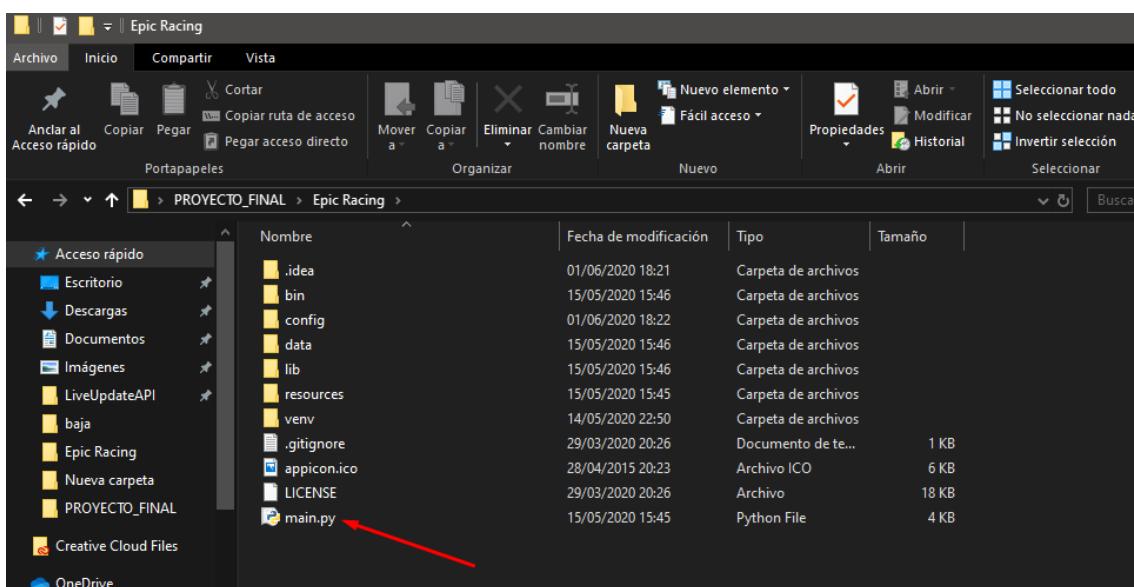


Figura C.1: Ubicación archivo main.py necesario para ejecutar el juego

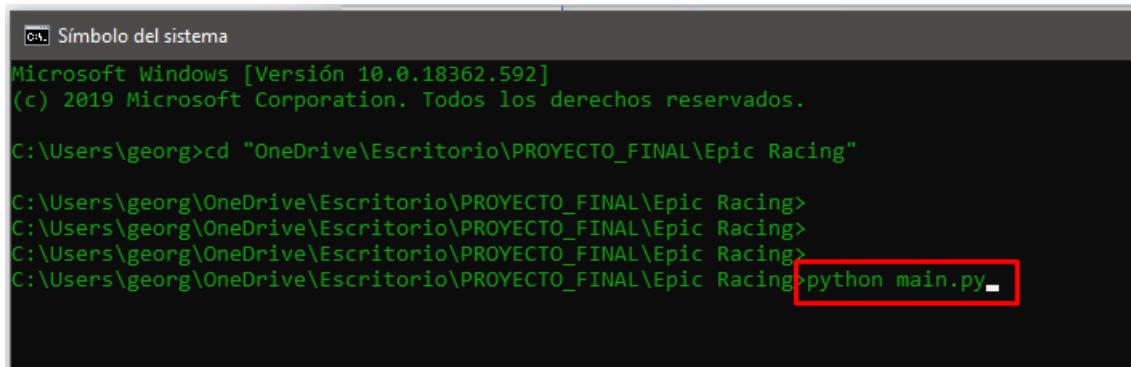
Ejemplo para moverse a la carpeta del juego con el comando CD:

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.18362.592]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\georg>cd "OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing"
C:\Users\georg\OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing>
```

Figura C.2: Moverse por carpetas con el comando CD

Lo siguiente será ejecutar el juego con el comando **python main.py**. Ejemplo para ejecutar el juego:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.18362.592]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Users\georg>cd "OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing"
C:\Users\georg\OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing>
C:\Users\georg\OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing>
C:\Users\georg\OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing>
C:\Users\georg\OneDrive\Escritorio\PROYECTO_FINAL\Epic Racing>python main.py
```

Figura C.3: Ejecutar el juego

Cuando se ejecuta por primera vez el juego, aparece un formulario para crear un perfil de usuario. El nombre debe contener entre cuatro(4) y diez(10) caracteres. Se escribe el nombre y se pulsa Enter (NO utilizar teclado numérico).

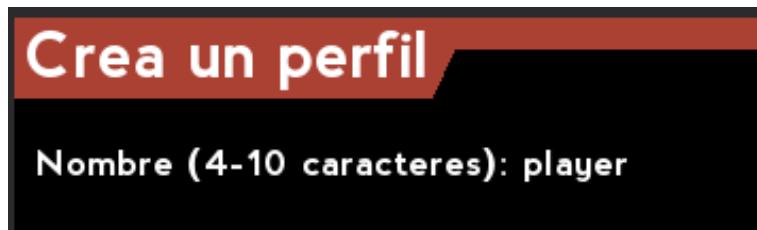


Figura C.4: Formulario para crear perfil de usuario

Una vez guardado el perfil hay que volver a ejecutar el juego con el comando **python main.py** en la terminal y estando en la carpeta donde se encuentra el fichero **main.py**.

Después de crear el perfil de usuario, el juego cargará siempre con ese nombre. Para cambiar el nombre de perfil es necesario modificar el código, se explica el procedimiento más adelante.



Figura C.5: Menú principal

En la opción **Configuraciones** se puede cambiar de idioma (español, inglés). Activar / desactivar los sonidos. Cambiar el tamaño de la ventana, a formato 16:9 (panorámico) o 4:3. Cambiar a formato fullscreen / windowed (pantalla completa). Mostrar o ocultar los fps del juego en el título. Volver al menú principal.



Figura C.6: Opción menú Configuraciones

Para moverse por el menú se utilizan las flechas del teclado (arriba, abajo, izquierda, derecha). **Recordar no utilizar el teclado numérico.**

En la opción **Ayuda** están las instrucciones para jugar.



Figura C.7: Opción menú Ayuda

- Presiona W / Flecha arriba para acelerar.
- Presiona S / Barra espaciadora / Flecha abajo para frenar.
- Presiona A / Flecha izquierda para girar hacia la izquierda.
- Presiona D / Flecha derecha para girar hacia la derecha.
- Si estas jugando pulsa F3 para capturar la pantalla, las capturas se guardan en el directorio /data/saves.
- Si te sales de la pista presiona Backspace para devolver el coche a la última posición válida.
- Intenta batir los objetivos en tiempo de cada circuito, salirse de la pista implica una penalización.
- Puedes volver al menú principal pulsando Enter / Esc.

Después de configurar el juego y leer la ayuda, solo falta jugar. Se vuelve a menú principal con la tecla Esc y se escoge la opción **Jugar**, aparecerá otro menú configurable donde se podrá escoger empezar la carrera, cambiar la pista, escoger el tipo de coche y su color.

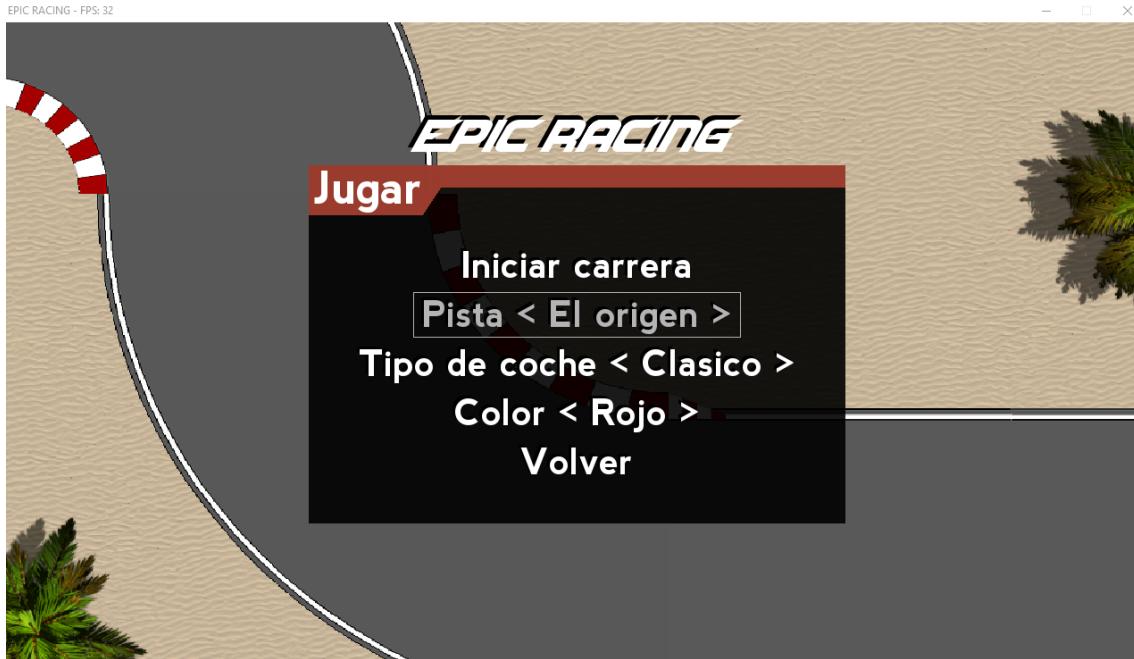


Figura C.8: Menú para iniciar carrera

Para empezar a jugar escogemos la opción **Iniciar carrera**.

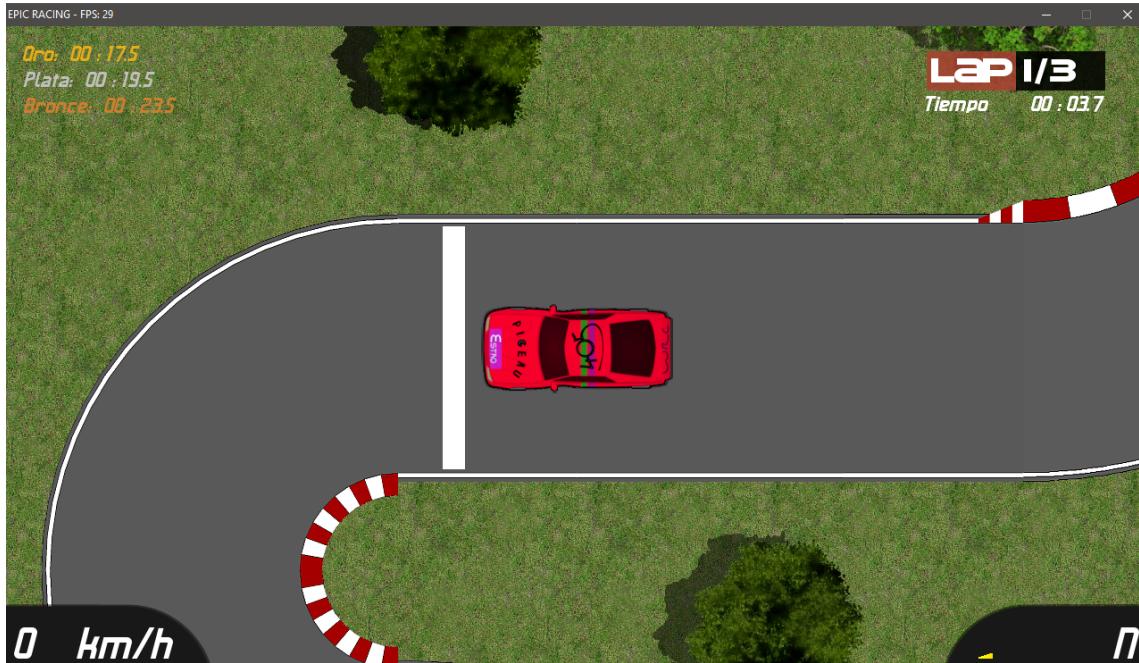


Figura C.9: Ventana juego iniciado

Después de terminar la partida aparece una ventana con el con los resultados de la partida y el scoreboard acumulado. El puntaje que aparece en rojo es el de la partida actual y el puntaje que aparece en blanco es de partidas anteriores.



Figura C.10: Ventana scoreboard partida

Anexo D. Como se ha generado este documento

Para variar, se ha elegido redactar este documento en un editor diferente de Word. Se trata de Overleaf un editor online de LaTex.

Overleaf es un servicio de LateX colaborativo en línea con el cual se puede realizar trabajos, escribir artículos técnicos, realizar posters y slides utilizando LateX en la nube. Como parte de tu membresía, IEEE brinda acceso completo a todas las herramientas de Overleaf. La herramienta además se encuentra integrada con Collaboratec, el nuevo hub de IEEE para colaborar en proyectos, investigación, ingeniería, bibliografía y mucho más.

En el repositorio de Gihub se encuentra un enlace a GoogleDrive con los ficheros necesarios para la realización de este documento.

Fin de trabajo