Máster en Inteligencia Artificial,
Reconocimiento de Formas e Imagen Digital

Trabajo Final de Máster

Videojuego de construcción y conducción sobre "Scalextric"

Autor:
Imanol Fraile de la Cruz

Tutor académico:
Roberto Vivó Hernando

Curso académico 2014/2015

Resumen

Los videojuegos están experimentando un continuo crecimiento en el mundo laboral. Cada vez son más las personas que se lanzan a desarrollar videojuegos junto a sus amigos para comercializarlos en las diferentes plataformas del mercado.

Las tecnologías de desarrollo de videojuegos avanzan a un ritmo vertiginoso y hoy en día cualquier persona con conocimientos básicos de programación puede iniciarse en este mundo y crear su propio juego.

Sin ir más lejos, en este proyecto se describen todas y cada una de las fases por las que se ha pasado para crear el videojuego WebScalextric VS. Gracias al estándar WebGL para gráficos en navegadores web y a la biblioteca gráfica Three.js, se ha implementado un videojuego de carreras sobre raíles que no necesita la instalación de ningún tipo de plugin para funcionar. Con WebGL, el juego hace uso de la tarjeta gráfica del cliente para procesar todos los objetos que componen la escena.

El propósito de WebScalextric VS es juntar a dos personas delante de un ordenador para que puedan pasar un buen rato compitiendo tanto en los circuitos prediseñados como en los que hayan construido en el editor de circuitos.

Palabras clave

WEBGL: especificación para gráficos 3D en navegadores web

THREEJS: biblioteca para gráficos 3D en la web

JAVASCRIPT: lenguaje de desarrollo web en la parte cliente

BLENDER: programa de modelado y animación en 3D

Keywords

WEBGL: 3D graphics specification for web browsers

THREEJS: library for 3D web graphics

JAVASCRIPT: client-side development web language

BLENDER: 3D modeling and animation program

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

Una de las salidas profesionales en mayor crecimiento para el experto en imagen digital es la programación de videojuegos. En esta línea, los juegos con alto contenido gráfico e interacción con ejecución sobre un navegador aprovechando las características de la tarjeta cliente están siendo una novedad, ofreciendo oportunidades de empleo o iniciativas emprendedoras dentro del mercado laboral. Actualmente existen numerosos ejemplos de juegos en web, donde no es necesaria la instalación de ninguna aplicación ni plugin, como *Adobe ShockWave*, *Adobe Flash* o el más reciente *Unity Web Player*, y que gozan de amplia popularidad.

El presente trabajo se enmarca dentro del grupo de proyectos orientados hacia la profesión informática, en general, y hacia la de diseño y programación de videojuegos web en particular. El mercado, altamente competitivo, demanda conocimientos demostrables sobre el dominio de tecnologías modernas, por lo que la construcción de demostradores totalmente funcionales es clave a la hora de convencer a un posible empleador. Por otro lado, el producto desarrollado aquí, puede servir a su vez como semilla de emprendimiento de iniciativas particulares en el campo de los videojuegos web multiplataforma.

El presente proyecto consiste en el desarrollo de un videojuego con dos partes diferenciadas; la edición interactiva de un circuito de carreras de coches en base a la composición de piezas prediseñadas como en el conocido juego "Scalextric", y la simulación de la conducción por el circuito.

Construir un videojuego desde cero para poder jugarlo desde un navegador web es la principal motivación para llevar a cabo este proyecto. Además, el hecho de elaborar un producto que finalmente puedan probar otros usuarios introduciendo simplemente una dirección web en su navegador es sumamente reconfortante. Para que todo esto sea posible se requieren conocimientos sobre WebGL y la biblioteca Three.js.

1.2. Objetivos del proyecto

El objetivo general del proyecto consiste en el desarrollo de un videojuego de carreras basado en el conocido juego de vehículos sobre raíles llamado "Scalextric". El videojuego debe poder jugarse directamente en un navegador web gracias al estándar WebGL y debe contar con un editor de circuitos y un modo de competición para uno y dos jugadores.

Los objetivos particulares que se quieren conseguir se dividen en cuatro grandes fases, las cuales se citan a continuación:

Fase de diseño gráfico:

- Construir una página web sencilla que presente el producto final.
- Diseñar la estética de la interfaz de usuario del videojuego.

Fase de modelado:

- Modelar las piezas básicas del circuito: tramo recto, tramo recto doble y tramo curvo de 90º.
- Modelar los vehículos del juego: furia roja y centella azul.
- Modelar los escenarios de los circuitos: llanura plana, glaciar fresco y los circuitos personalizados.
- Importar los modelos a la escena con Three.js.

Fase de implementación:

- Implementar los menús de selección.
- Elaborar un editor interactivo para la construcción de circuitos 3D cerrados.
- Facilitar la exportación e importación de circuitos.
- Desarrollar el simulador de conducción con sus 2 modos de juego: contra el ordenador y contra otro humano.
- Implementar diferentes vistas de cámara para el simulador.
- Establecer la iluminación y las sombras de la escena.
- Añadir elementos de realimentación para el usuario (número de vueltas, posición actual, cartel de carga, ayudas, etc.).

Fase de sonido:

- Ambientar los menús y los circuitos con diferentes temas instrumentales y efectos de sonido.

1.3. Estructura de la memoria

En el presente capítulo se ha introducido el contexto y la motivación existente que ha originado la elaboración de este proyecto. A continuación se ha citado el objetivo general del mismo y sus objetivos particulares.

En el Capítulo 2 se documentará al lector sobre las características más relevantes de la tecnología empleada para el desarrollo del videojuego WebScalextric VS. Se hablará de WebGL y Three.js, además de la metodología que se ha seguido a la hora de implementar el código del juego.

En el Capítulo 3 se hablará en detalle del desarrollo de cada uno de los objetivos particulares presentados en este mismo capítulo. Se ha decidido mostrar fragmentos de código junto a gran parte de las descripciones de las tareas para facilitar la comprensión de las mismas. No obstante, el lector que no esté interesado en los detalles de implementación podrá comprender de igual forma el objetivo de cada una de las tareas propuestas.

Finalmente, en el Capítulo 4 se concluirá la memoria destacando los objetivos alcanzados y algunas de las muchas ideas que existen para un trabajo futuro con el videojuego.

Capítulo 2

Metodología del proyecto

2.1. Tecnología empleada

Antes de comenzar a programar una sola línea de código, es conveniente conocer las posibilidades y limitaciones de las tecnologías que se van a utilizar. En este apartado se detalla la especificación WebGL y la biblioteca gráfica Three.js.

2.1.1. WebGL

WebGL es una especificación estándar desarrollada para visualizar gráficos en 3D en navegadores web sin la necesidad de plugins como *Adobe Flash* o *Unity Web Player*. Funciona sobre cualquier plataforma que soporte OpenGL 2.0 u OpenGL ES (*Embedded Systems*) 2.0.

Las ventajas que diferencian a WebGL de otras tecnologías de visualización gráfica son las siguientes:

- **Rendimiento**. WebGL es sorprendentemente rápido y como utiliza aceleración por hardware, resulta muy adecuado para la programación de videojuegos y visualizaciones complejas. No es la única tecnología que se aprovecha de la aceleración por hardware, pero WebGL está diseñado con el rendimiento en mente y debería funcionar mejor que otras opciones en gran parte de los casos.

- **Sencillez**. Puede llevar a cabo de forma simple tareas como los efectos de iluminación y reflexión sobre un material, cuyo alcance sería inviable o muchísimo más complicado con tecnologías como por ejemplo SVG (*Scalable Vector Graphics*).

- **Shaders**. Las aplicaciones WebGL pueden utilizar shaders. Un shader es un pequeño programa pensado generalmente para producir efectos visuales complejos. El lenguaje para OpenGL en el que están escritos estos shaders es el GLSL (*OpenGL Shading Language*).

En cuanto a algunas razones por las que WebGL ha ganado tanta popularidad en los últimos años, se encuentran las que siguen a continuación:

- Está basado en OpenGL, un *framework* bien probado para el desarrollo de aplicaciones 3D.
- Permite la integración del DOM (*Document Object Model*), ya que WebGL se muestra mediante el elemento 'canvas' y puede ser combinado con otras tecnologías web.
- Es capaz de ejecutar aplicaciones en diversos navegadores y plataformas.
- Es una especificación abierta, por lo que no está controlada por un solo dueño o empresa y cualquier persona puede colaborar en su desarrollo.
- Presenta una facilidad de aprendizaje para comenzar con el mundo de la programación en 3D.

A pesar de todas estas ventajas y características, WebGL no es del todo versátil. Para desarrollar aplicaciones 2D simples existen más bibliotecas en SVG. Además, SVG es comprendido por los motores de búsqueda y se integra mejor con los controladores de eventos del DOM. WebGL necesita bibliotecas adicionales para intentar rellenar este hueco. Por otro lado, WebGL no es adecuado para máquinas con bajas especificaciones técnicas o navegadores anticuados.

Por lo que respecta al tema de la seguridad, han existido controversias durante los últimos años sobre la viabilidad de WebGL [1]. Microsoft señaló que uno de los grandes problemas de seguridad que tenía WebGL era que como se conectaba directamente a la GPU (*Graphics Processing Unit*) del sistema, esto se podría aprovechar por un usuario malintencionado para ejecutar código malicioso a través de las imágenes de origen cruzado empleadas como texturas.

2.1.2. Three.js

Como WebGL es una tecnología diseñada para trabajar directamente con la GPU, es difícil de codificar en comparación con otros estándares web más accesibles. Este es el principal motivo por el que han aparecido numerosas bibliotecas de JavaScript para resolver este problema como Three.js, C3DL, CopperLicht, Curve3D, CubicVR, EnergizeGL, GammaJS, GLGE, GTW, JS3D, Kuda, O3D, OSG.JS, etc.

La biblioteca de interés para este proyecto es Three.js, pues es la más popular en términos de número de usuarios y es la que se ha estudiado en la asignatura de 'Gráficos por computador' de este máster. Es ligera y tiene un bajo nivel de complejidad en comparación con la especificación WebGL original.

Three.js fue creada y liberada en GitHub por el español Ricardo Cabello en abril de 2010, conocido por su seudónimo de Mr. Doob. El código se desarrolló primeramente en ActionScript y fue traducido más tarde a JavaScript por su independencia de plataforma.

Para hacer uso de esta biblioteca no hay más que localizarla desde una página web, bien sea una copia local o remota. En el caso particular de este proyecto se incluye como se puede ver a continuación:

```
<script
src = "./recursos/lib/three.js"> <!-- Biblioteca gráfica -->
</script>
```

Una vez importada en la página web de esta manera, ya se puede hacer uso de toda su funcionalidad escribiendo THREE seguido de un punto y el nombre de una de sus funciones junto a sus argumentos (si los tiene). Por ejemplo, con la siguiente línea de código se instancia la escena sobre la que se añaden los objetos gráficos:

```
var scene = new THREE.Scene();
```

Almacenar una textura en una variable de JavaScript es tan sencillo como pasar por argumento la ubicación de la misma a la función *loadTexture* de *ImageUtils* de Three.js:

```
var textura = new THREE.ImageUtils.loadTexture("./textura.png");
```

Uno de los puntos flacos de la biblioteca es su documentación [2]. Actualmente está incompleta y la existente está muy poco elaborada. Dispone de un pequeño tutorial muy ameno para aprender a crear la primera escena con todos los elementos básicos donde se muestra un cubo dando vueltas (*Figura 1*).

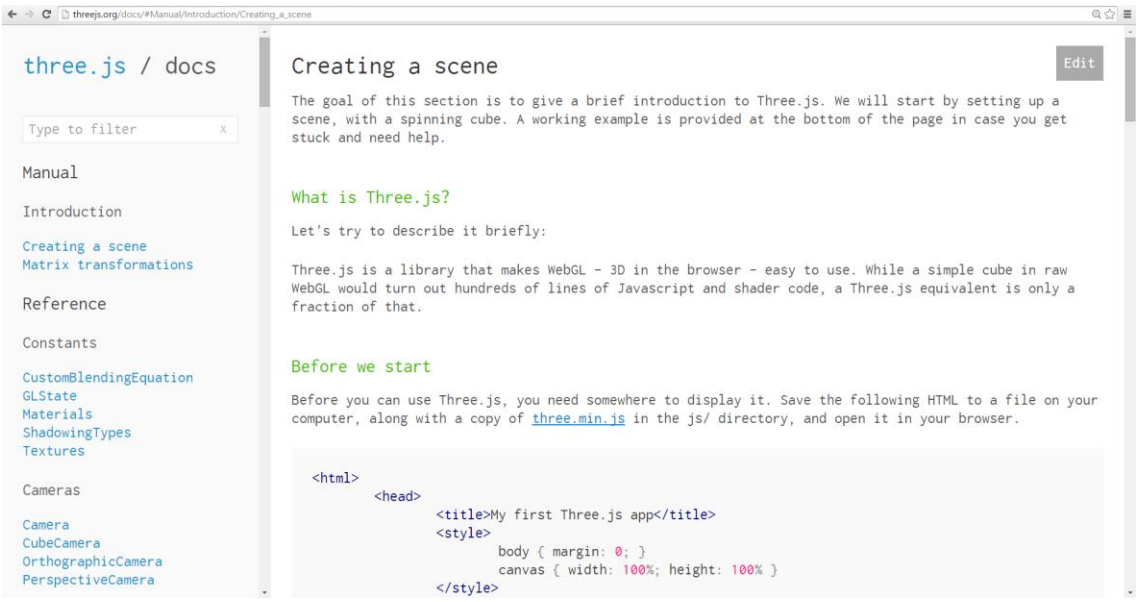


Figura 1.- Fragmento del tutorial introductorio de la documentación de Three.js.

Sin embargo, la documentación se queda corta en ejemplos a medida que aumentan los deseos de enriquecer una aplicación 3D. Afortunadamente, el hecho de que Three.js sea tan popular ha facilitado la existencia de tutoriales y ejemplos muy detallados por parte de muchas personas en la web.

2.2. Metodología de trabajo

La metodología seguida para la implementación del código de este proyecto está compuesta por tres fases bien diferenciadas.

La primera consiste en la contextualización del problema, pues sin saber qué resultado final se quiere obtener, difícilmente se podrán alcanzar los objetivos propuestos.

La segunda fase contiene cualquier actividad relacionada con la búsqueda de información. Internet [3] ha sido la fuente principal tanto teórica como de ejemplos prácticos.

Una vez recopilada la información necesaria, la tercera fase consiste en plasmarla en el código. Generalmente se comprueba que los ejemplos proporcionados por los manuales y tutoriales funcionan correctamente y a continuación se adaptan a las necesidades del proyecto.

La metodología no forma un ciclo secuencial, pues las fases segunda y tercera se realizan como mínimo una vez por cada objetivo del proyecto.

El entorno de trabajo personal ha sido en todo momento Windows. Tanto la página web como el videojuego se han desarrollado en un entorno local gracias al servidor Wamp y al editor de texto con soporte para lenguajes de programación Notepad++. Los resultados han sido probados en los tres navegadores más utilizados del mercado: Internet Explorer, Mozilla Firefox y Google Chrome.

El modelado de los objetos gráficos 3D más complejos como los vehículos, los árboles y las piezas del circuito se han llevado a cabo con Blender debido a la gran cantidad de opciones que otorga para ello. El resto de elementos como los botones de los menús o las paredes y el suelo de la habitación del escenario final, se han creado directamente con la biblioteca Three.js por tratarse de simples objetos en 2D.

Una vez ejecutado y probado el proyecto con el servidor local del ordenador propio, se ha utilizado el programa Filezilla para alojarlo en un servidor público bajo el dominio de www.webscalextric.com.

Al tratarse de un videojuego, el código desarrollado es el elemento más importante de este proyecto, por lo que se ha replicado periódicamente en varios

ordenadores y servidores que además están en distintos lugares físicos para evitar una posible catástrofe.

Los ficheros que componen la totalidad del proyecto se han organizado siguiendo la jerarquía que se puede ver en la *Figura 2* para un mayor orden y facilidad de búsqueda.

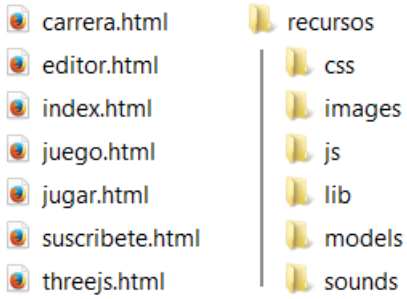


Figura 2.- Jerarquía de directorios del presente proyecto.

Capítulo 3

Desarrollo del proyecto

3.1. Fase de diseño gráfico

Esta primera fase del proyecto concierne cualquier elemento relacionado con el aspecto visual de la aplicación. Desde los colores seleccionados para los botones de los menús hasta la estética elegida para los árboles que decoran los circuitos, pasando por todos y cada uno de los elementos visuales del videojuego. El objetivo de esta fase es dotar de un aspecto uniforme y consistente tanto a la página web como al videojuego en sí para que no desentonen entre ellos.

Las tareas que se analizan en esta fase de diseño son la creación de la página web y el diseño de la estética de la interfaz de usuario.

Tarea 1. Construcción de la página web

Con los conocimientos sobre HTML (*HyperText Markup Language*) y CSS (*Cascading Style Sheets*) de los que dispongo, me he aventurado en la creación de una página web sencilla. Cabe destacar que esta tarea no estaba contemplada en los objetivos del proyecto. Sin embargo, he considerado que la creación de un videojuego para la web debería disponer de su propia página web previa como todo producto que se precie.

El objetivo de esta página es el de realizar una introducción al videojuego para incitar al usuario final a probarlo e informarse sobre la tecnología empleada durante su fase de desarrollo.

El aspecto de la página de inicio www.webscalextric.com es el que muestra la *Figura 3*. La web cuenta con 3 apartados más. En el apartado 'Jugar' se explican los controles del juego y se incluye un botón que lleva a su página del menú principal, llamada 'juego.html'. A continuación, el apartado 'Three.js' muestra información sobre esta biblioteca y sus posibilidades. Finalmente, en el apartado 'Suscríbete' se proporciona un formulario para suscribir al usuario al boletín de noticias sobre las futuras actualizaciones del videojuego.

Figura 3.- Aspecto de la sección de inicio de la web www.webscalextric.com.

Tarea 2. Diseñar la estética de la interfaz de usuario

En esta tarea se pretende conseguir una interfaz de usuario sencilla, intuitiva y uniforme para que el usuario navegue por ella tanto a la hora de seleccionar el modo de juego como al crear un circuito personalizado.

La *Figura 4* muestra una captura funcional de cada una de las diferentes pantallas de selección junto a una más del editor. Se ha escogido una gama de colores azules para los botones y letreros de información. La geometría de los botones cuya interacción da lugar al cambio de un menú por otro es rectangular. Los botones que sirven para volver al menú anterior se encuentran siempre situados en la esquina inferior derecha de la pantalla. Las letras que se muestran sobre los botones son todas de color blanco. Por su parte, las herramientas del editor de circuitos presentan un color negro para diferenciarse de los demás botones. En cuanto al fondo de los menús, está formado por una bandera de cuadros cuya tonalidad se ha adaptado a la gama de colores azules del resto de elementos.

La complejidad de esta fase recae en la inversión del número de horas que se han llevado a cabo hasta encontrar el diseño actual de colores, botones y elementos gráficos.

Figura 4.- Aspecto visual de los diferentes menús del juego.

3.2. Fase de modelado

La fase de modelado va estrechamente unida a la fase de diseño gráfico. Los objetos 3D diseñados deben seguir todos un mismo estilo gráfico para que el videojuego tenga coherencia visual. El principal propósito es modelar las piezas básicas del circuito con las dimensiones adecuadas para que encajen unas con otras perfectamente. En cuanto a los vehículos del juego, se permitía importar modelos ya construidos por terceros directamente para su uso. A pesar de ello, como en este máster hemos aprendido a defendernos con el programa Blender, he decidido modelar mis propios vehículos como se verá en la Tarea 4. Por último, para que la escena final no sea una simple carretera con dos vehículos, se ha decidido crear algunos elementos a escala para llenar los huecos vacíos.

Las tareas que se analizan en esta fase del proyecto son el modelado de las piezas del circuito, los vehículos, los escenarios de los mismos y la importación de modelos 3D con Three.js.

Tarea 3. Modelado de las piezas básicas del circuito

Blender es un programa informático multiplataforma dedicado especialmente al modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales. En este proyecto se ha utilizado para modelar y añadir texturas a los objetos 3D del videojuego que se analizan en esta fase del proyecto.

Para poder construir un circuito cerrado que no sea un círculo se necesitan como mínimo dos piezas: una recta y una curva. Como la creación de un videojuego desde cero abarca aspectos muy diversos, se ha decidido modelar tres piezas básicas por el momento. Éstas son un tramo recto, un tramo recto de doble longitud y un tramo curvo de 90 grados (*Figura 5*). El juego está pensado para dos jugadores, por lo que las texturas de las piezas tienen dibujados dos pares de raíles.

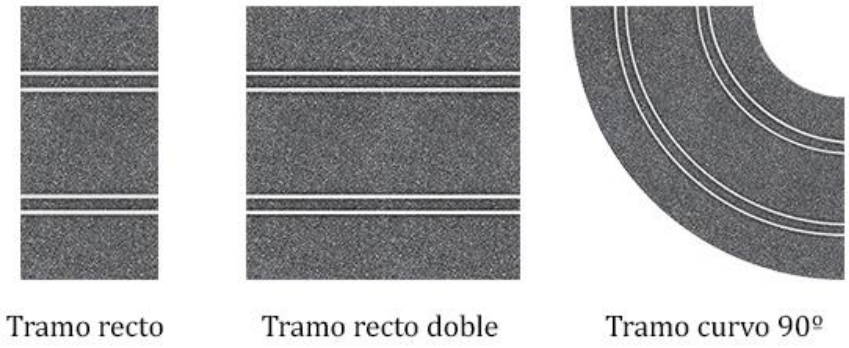


Figura 5.- Diseño principal de los 3 tramos con los que se puede construir un circuito.

En realidad las piezas no son planas, ya que presentan un pequeño volumen para que en el videojuego no parezca que estén pegadas al suelo. Es por ello que están formadas por 8 vértices en lugar de 4. Además, el tramo curvo de 90º es realmente una pieza rectangular como las otras dos, solo que su textura presenta una transparencia para eliminar las dos esquinas (*Figura 7*). De esta forma se reduce al máximo la complejidad de su geometría. El diseño de la geometría de las piezas ha pasado por dos etapas, la de mano alzada y la matemáticamente correcta.

En la primera etapa no se tuvieron en cuenta las coordenadas 3D de los vértices que componen cada pieza. Un ejemplo de coordenadas de un vértice en esta etapa podría ser (-1’5849, 2’1554, 1’3221). Las piezas se modelaron al alza con Blender y más tarde, a la hora de intentar ensamblar unas con otras, el circuito nunca cerraba a la perfección.

Este grave problema de diseño se solucionó en una segunda etapa. Se remodelaron las tres piezas de nuevo teniendo en cuenta las componentes XYZ de cada uno de sus 8 vértices, las cuales se simplificaron a por ejemplo (-1, 2, 1). De

esta manera, conocer el desplazamiento que debía haber entre la posición de dos piezas contiguas resultó trivial y matemáticamente perfecto. En la *Figura 6* se aprecia la geometría definitiva del tramo recto con las coordenadas locales de sus 4 vértices superiores. La coordenada local Z = 1 se corresponde con la coordenada global Z = 0'03955, pero como todas las piezas tienen la misma Z, se ha simplificado por 1 para su ilustración.

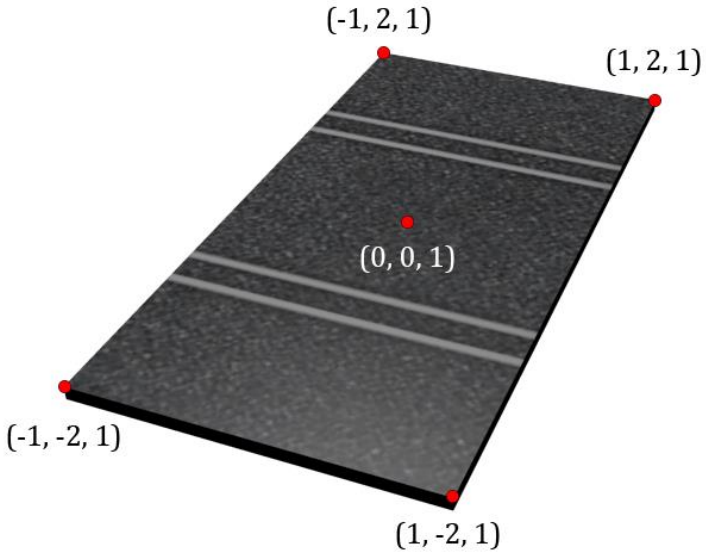


Figura 6.- Geometría y coordenadas XYZ de los 4 vértices superiores del modelo del tramo recto.

La conclusión de este apartado es que modelar objetos a ojo no es una buena idea cuando el objetivo es unirlos en una fase posterior.

Por lo que respecta a las texturas utilizadas para las piezas, que son exactamente las que se han mostrado en la *Figura 5*, se han creado con Photoshop CS6. Photoshop es un programa muy práctico para crear y manipular imágenes, entre otros fines. Uno de los aspectos más útiles de este programa es que dispone de un sistema de capas en las que poder situar diferentes elementos y manipularlos de manera independiente para acabar creando una imagen final (*Figura 7*). Además, las capas se pueden ocultar pulsando un simple botón o incluso aplicarles una transparencia leve para ver lo que queda por debajo o por encima de ellas.

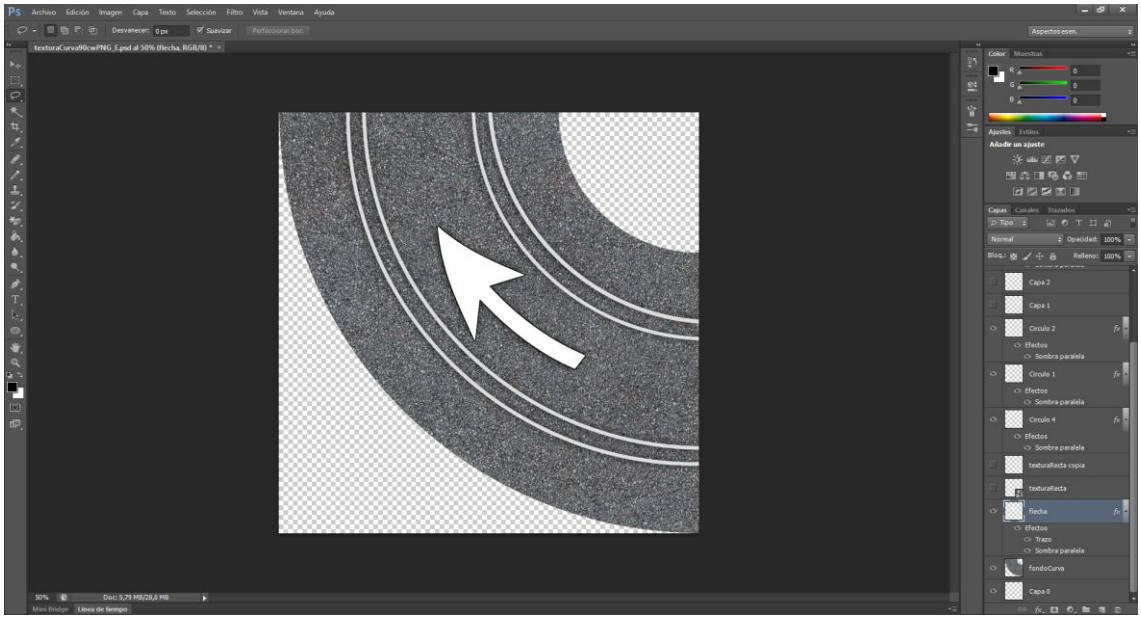


Figura 7.- Espacio de trabajo de Photoshop donde se muestran algunas de las capas (derecha) que componen la textura para la curva de 90 grados del editor de circuitos.

Tarea 4. Modelado de los vehículos del juego

Para modelar la geometría de los vehículos se ha partido de un videotutorial [4] en el que se explican los pasos a seguir para construir nuestro propio vehículo 3D con Blender en unos cuantos minutos. Como los coches pretenden ser de juguete, su geometría no es compleja.

El primer vehículo modelado se corresponde con el nombre de 'Furia roja' (*Figura 8*). Está formado por varios objetos 3D como el chasis, las ruedas o los retrovisores, todos ellos unidos para formar una sola malla definitiva. La parte de modelado ha sido relativamente sencilla. Sin embargo, la estructura de sus colores, materiales y texturas ha sufrido varios cambios durante el proyecto debido a la complicada forma de importar modelos que tiene la biblioteca Three.js y a la falta de experiencia en el tema. En la Tarea 6 se describe el procedimiento seguido para importar objetos 3D creados con Blender en la escena de Three.js y todas sus complicaciones.

Tras finalizar el modelado del vehículo, el siguiente paso consistió en establecer distintos materiales para cada una de sus partes. Por ejemplo, para el chasis se eligió un color rojo y una especularidad baja para que las luces se reflejaran levemente sobre él. Para las ruedas en cambio, como la goma no refleja la luz, la especularidad de su material es nula y el color negro. Desgraciadamente, todo el trabajo empleado en los materiales ha resultado en vano. Los motivos se explican al final de la Tarea 6.

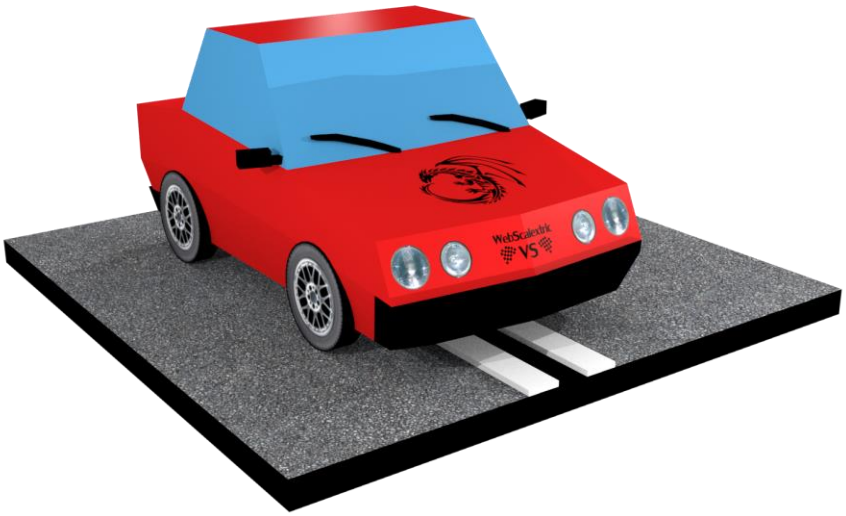


Figura 8.- Renderizado final realizado con Blender v2.74 del vehículo 'Furia roja' sobre un tramo de circuito.

Una vez establecidos los materiales y sus propiedades, llega el turno de los detalles finales que como siempre dan un aspecto mucho más realista a cualquier objeto: las texturas. La primera característica importante de una textura es que debe estar guardada en un formato que permita almacenar un canal alfa de transparencia, como por ejemplo PNG (*Portable Network Graphics*). Gracias a la transparencia de una textura se consigue que se vea el color rojo del chasis cuando se pone el vinilo del dragón, los faros o el logotipo del videojuego sobre la carrocería del vehículo.

Para ir pegando adecuadamente las diferentes texturas a lo largo del modelo 3D, Blender dispone de la herramienta 'Editor UV'. Cuando pulsamos sobre ella, la pantalla del programa se divide en dos, quedando a la izquierda la textura y a la derecha el resultado de su aplicación al modelo 3D (*Figura 9*). Al principio cuesta dominar la técnica, pues no es sencillo comprender cómo gestiona Blender las texturas sin haber leído antes sobre el tema. Una vez dominada, resulta bastante cómodo y gratificante situar en tiempo real las texturas sobre tus propios modelos en 3D.

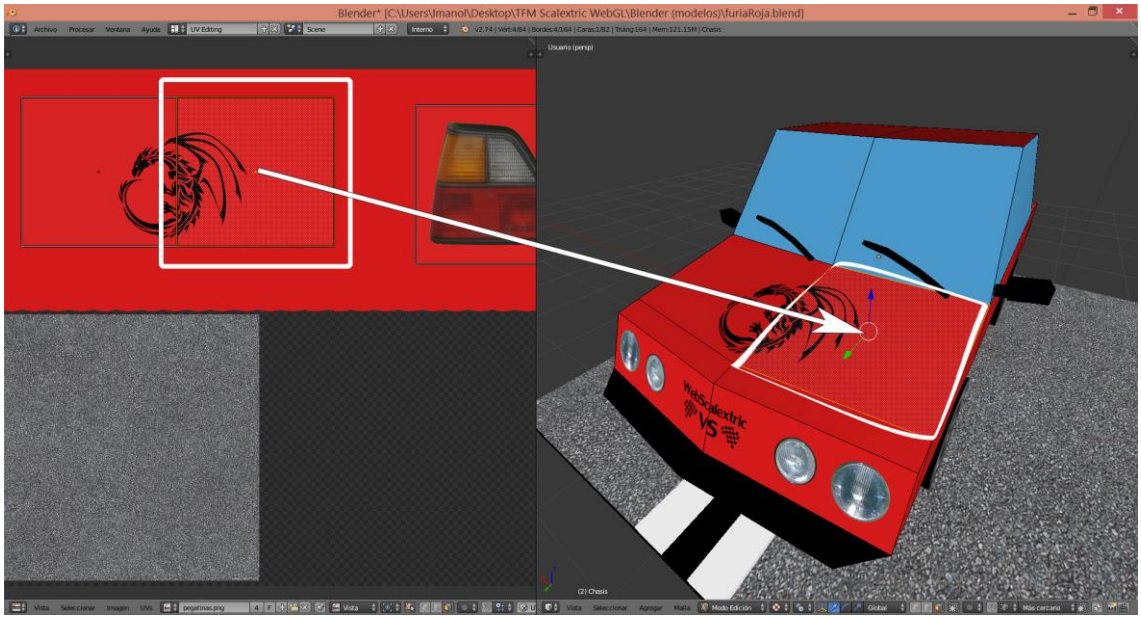


Figura 9.- Correspondencia de los vértices de la textura 2D (izquierda) con los vértices del modelo 3D (derecha) en el modo 'Editor UV' de Blender.

Con las texturas también han surgido problemas, esta vez conceptuales. Lo que se hizo al principio fue tener separadas en distintas imágenes PNG las texturas de cada uno de los objetos físicos del vehículo. Es decir, tenía una imagen PNG para la textura de las llantas, otra para los logotipos, faros y vinilo y otra más para el asfalto. Cuando me documenté sobre la forma de importar modelos a Three.js, descubrí que solamente se admitía una textura para todo un objeto 3D. Llegados a este punto, cabe aclarar que el proyecto cuenta con una versión de los vehículos con un asfalto para situarlos en el menú de selección de vehículos (*Figura 8*) y otra versión sin el asfalto para utilizarlos en las carreras.

La solución a este problema fue la creación de una sola imagen de textura (*Figura 10*) donde se colocaron todas las texturas individuales. Este cambio obligó a redistribuir las coordenadas de textura antiguas para adaptarse a la nueva imagen 2D.

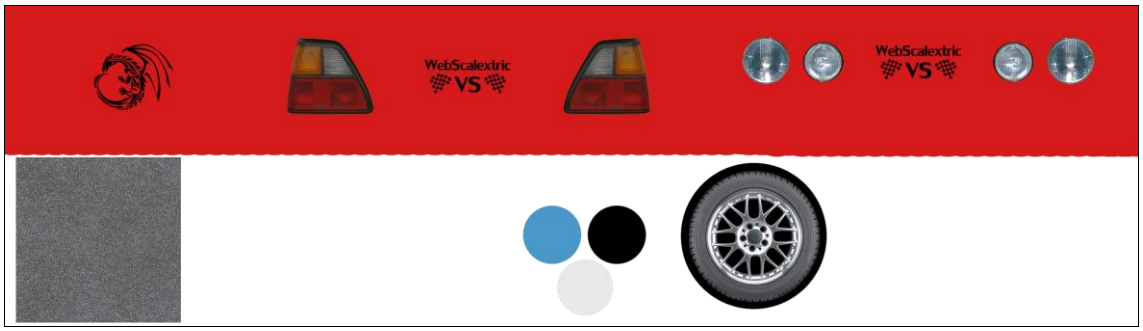


Figura 10.- Imagen en formato PNG que se utiliza como textura definitiva para el vehículo 'Furia roja'.

Para finalizar con esta tarea se ha modelado un segundo vehículo, cuyo nombre es 'Centella azul'. Para crearlo se ha partido de la geometría del primer vehículo y se le han añadido algunos cambios para que no fueran idénticos. Estos cambios son un alerón trasero y tres pequeñas crestas en su parte superior (*Figura 11*). La textura es prácticamente la misma, excepto por el color azul del chasis y el vinilo que lo diferencian del 'Furia roja'.

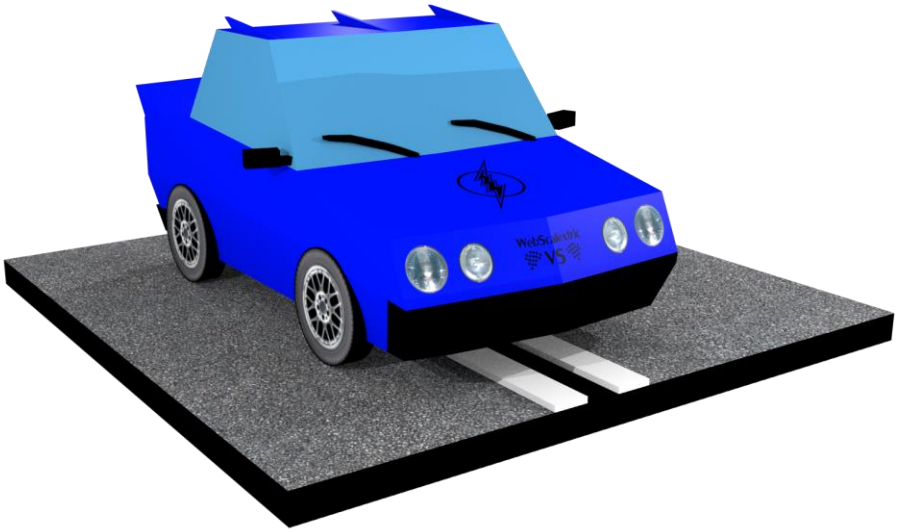


Figura 11.- Renderizado final realizado con Blender v2.74 del vehículo 'Centella azul' sobre un tramo de circuito.

Tarea 5. Modelado de los escenarios de los circuitos

Una vez que tenemos modelado el circuito y los vehículos, es el momento de dar vida a la escena que los contiene. El videojuego cuenta con 3 ambientaciones distintas, una para cada circuito seleccionado: 'Llanura plana', 'Glaciar fresco' y 'Circuito personalizado'.

El objetivo principal consiste en simular que los circuitos se encuentran sobre una moqueta o un suelo de una habitación cuadrada. Como el modelado de estos objetos está basado en planos 2D, su implementación se ha llevado a cabo con las geometrías básicas de las que dispone la biblioteca Three.js.

El suelo de las habitaciones está formado por un plano. Toda malla 2D o 3D (*mesh* en inglés) que se crea en Three.js necesita una geometría con unas dimensiones y un material con sus diversas características que la envuelva. A continuación se muestra la textura, la geometría y el material utilizados para dar vida a la moqueta verde sobre la que se sitúa el circuito 'Llanura plana':

```
// Si el circuito seleccionado es 'Llanura plana', se carga su textura
if (circuito == 'llanuraPlana') {
   // Textura del plano
   var texturaSuelo = new
     THREE.ImageUtils.loadTexture("./texturaSueloLlanura.png");
   // La textura se repite a lo largo del plano para ganar resolución
   texturaSuelo.wrapS = texturaSuelo.wrapT = THREE.RepeatWrapping;
   texturaSuelo.repeat.set(4, 4);
}

// Anchura y Altura del plano (130x130)
geoPlanoSuelo = new THREE.PlaneBufferGeometry(130, 130);

// Color ambiente, del material, textura y lado por el que se verá
materialPlano = new THREE.MeshBasicMaterial(
                                  { ambient: 0x000000,
                                      color: 0xFFFFFF,
                                        map: texturaSuelo,
                                       side: THREE.FrontSide }
                                       );

// Malla con la geometría y el material previamente definidos
planoSuelo = new THREE.Mesh(geoPlanoSuelo, materialPlano);

// Objeto 3D con todas las mallas que dan pie al decorado (suelo,
paredes, árboles...)
decorado = new THREE.Object3D();

// Se añade la malla al objeto 3D
decorado.add(planoSuelo);

// Se añade el objeto 3D a la escena del videojuego
scene.add(decorado);
```

El resultado de las anteriores líneas de código da lugar a la creación de un plano de 130 unidades de ancho por 130 de alto cubierto por una de sus caras (la visible) con la primera textura que aparece en la *Figura 12*.

Figura 12.- Texturas atribuidas al plano que forma el suelo o moqueta en función del circuito seleccionado.

Las paredes de las habitaciones se han creado de la misma forma, solo que con diferentes texturas y diferente geometría. En lugar de ser un plano, ahora se

trata de una caja (*BoxGeometry*), cuyas dimensiones son las mismas que las del plano (130x130x130), incluyendo ahora una tercera dimensión, la profundidad:

```
// Anchura, Altura y Profundidad de las paredes (130x130x130)
geoParedes = new THREE.BoxGeometry(130, 130, 130);
```

Al igual que ocurre con las texturas del suelo, se ha querido utilizar una textura diferente para las paredes de cada circuito en concordancia con el suelo. La *Figura 13* muestra las 3 texturas empleadas para "empapelar" las paredes de los diferentes circuitos.

Figura 13.- Texturas atribuidas a las caras que forman las paredes en función del circuito seleccionado.

Por último, se han añadido algunos objetos más para ambientar los circuitos como árboles o casas. La textura de los abetos es diferente en función del circuito seleccionado (nevados o sin nieve). A diferencia del resto de objetos modelados, los árboles han sido descargados de Internet [5] e importados con Blender. Lo único que se les ha manipulado ha sido sus texturas, pues venían separadas en varias imágenes PNG que se han reorganizado en una sola para poder importarlos correctamente a la escena con Three.js.

La granja del circuito 'Llanura plana' se ha descargado también de Internet [6]. El modelo 3D venía sin ningún tipo de textura, por lo que se ha creado desde cero a partir de la combinación de varias texturas de madera y otros materiales.

En la *Figura 14* podemos ver el aspecto final que presenta esta granja tras realizar un renderizado de la misma con Blender. A pesar de no partir desde cero por el hecho de disponer del modelado de la granja y su decorado, la fase de texturizado ha llevado varias horas, pues la cantidad de mallas que componen la granja es mucho más elevada que la de los árboles o los vehículos.

Figura 14.- Modelo 3D descargado gratuitamente de Internet para su uso en el proyecto.

Tarea 6. Importación de los modelos a la escena con Three.js

El propósito de esta tarea parece bien sencillo: hacer uso de los modelos 3D creados con Blender en la escena tridimensional que proporciona la biblioteca Three.js. Sin embargo, se han dado varios inconvenientes, los cuales se especifican al final de esta tarea.

Para empezar, hay que tener en mente los diferentes formatos que Blender permite darle a nuestros diseños 3D a la hora de exportarlos a un fichero externo: DAE, 3DS, FBX, BVH, PLY, OBJ, X3D y STL. El formato con el que se ha tenido éxito ha sido el OBJ (*Wavefront*).

Una vez exportado, el siguiente paso es abrir la web www.threejs.org/editor. Esta página contiene un editor en fase beta de objetos OBJ para importarlos, manipularlos y exportarlos en formato JSON (*JavaScript Object Notation*). No se va a entrar en los detalles de cada formato, pues las modificaciones aplicadas a los objetos 3D no han requerido acceder a sus atributos más internos. No obstante, es importante saber que el formato JSON organiza la información en forma de árbol jerárquico con pares nombre-valor, de forma que es muy sencillo acceder a cada atributo mediante su nombre y utilizando índices.

Tras importar el OBJ, el primer paso es escalarlo, pues generalmente se crea con unas dimensiones muy pequeñas en la escena del editor. A continuación, se selecciona la propiedad 'Map' de la pestaña 'Material' del objeto seleccionado y se carga de nuevo la imagen de textura que ya se utilizó en Blender. Lo que se

consigue con este paso es asociar las coordenadas de textura automáticamentePor último, se exporta el objeto en formato JSON con la opción 'File -> Export Object'.

La tercera y última fase consiste en importar el objeto JSON generado por el editor web de Three.js a nuestra escena. El siguiente código muestra cómo se ha importado el vehículo 'Furia roja' a nuestra escena:

```
// Se instancia el cargador de objetos JSON 'ObjectLoader' de Three.js
var loader = new THREE.ObjectLoader();

// Se carga el modelo 3D en formato JSON
loader.load("./models/furiaRoja.json", function (obj) {

   // Se carga su textura
   var tx = new THREE.ImageUtils.loadTexture("./tx/txFuriaRoja.png");
   tx.minFilter = tx.magFilter = THREE.LinearFilter;

   // Se le pega la textura al objeto 3D cargado
   obj.traverse(function (child) {
      if (child instanceof THREE.Mesh) {
         child.material.setValues ({map: tx});
      }
   });

   // Se le otorga un nombre como identificador
   obj.name = 'furiaRoja';

   // Se escala acorde al tamaño en la escena
   obj.scale.set(0.5,0.5,0.5);

   // Se añade a la escena de Three.js
   scene.add(obj);
});
```

La *Figura 15* resume de forma gráfica la interacción de objetos durante los diferentes pasos explicados para transportar un modelo 3D desde Blender hasta la escena de Three.js.

Figura 15.- Conversiones necesarias de un modelo 3D realizado con Blender para poder importarlo en la escena de Three.js.

Ahora llega el turno de los inconvenientes. Lo primero de todo es que el editor web de Three.js se encuentra en fase beta. Hay que ir con mucho cuidado porque no se puede deshacer un error. Si nos equivocamos en algo, tenemos que

volver a cargar el modelo OBJ en el editor y empezar de nuevo con las transformaciones que queramos hacerle antes de exportarlo al formato JSON.

Tal y como se ha comentado en la Tarea 4, los materiales que se crearon para los vehículos en Blender no se han podido utilizar en el editor beta de Three.js. En realidad, cuando se exporta con Blender un objeto 3D al formato OBJ, se generan dos ficheros: un fichero con extensión OBJ y otro con extensión MTL. El fichero con extensión MTL es el que contiene las características de los materiales del objeto. El problema del editor beta es que no soporta los ficheros MTL. Por lo tanto, la solución que se ha tomado para conseguir que los vehículos estén pintados como si de materiales se tratase ha sido envolverlos completamente con sus texturas PNG. Si observamos de nuevo la textura que se presentó en la *Figura 10*, podemos apreciar que contiene todos los colores del vehículo: rojo para el chasis, azul para las lunas, negro para los parachoques y gris para el tubo de escape.

3.3. Fase de implementación

Llegados a este punto del proyecto, ya tenemos modelados los objetos más importantes de la escena del videojuego: las piezas y los vehículos. El siguiente gran paso consiste en programar la mecánica y la interacción del juego. La fase de implementación está compuesta por tres programas JavaScript bien diferenciados: juego.js, editor.js y carrera.js.

Las tareas que se analizan en esta fase del proyecto son la implementación de los menús, el editor de circuitos y el simulador de conducción.

Tarea 7. Implementación de los menús de selección

En la mayoría de los casos, los videojuegos están formados por una interfaz de usuario cuyo propósito es navegar por ella para realizar una serie de selecciones antes de comenzar a jugar. El programa que contiene toda la interfaz de los menús del juego se llama juego.js.

Biblioteca Raycaster.js y eventos del ratón

Para implementar la interacción con los botones de los menús del juego ha sido necesario el uso de la biblioteca Raycaster.js, cuyo objetivo es detectar la posición de los objetos 3D de la escena (los botones) utilizando el sistema de coordenadas 2D de la pantalla (la escena). Su funcionamiento se explica en los siguientes párrafos.

En primer lugar, se necesita crear una variable que contenga un objeto de esta biblioteca, otra para las coordenadas XY del ratón, otra más donde almacenar el último objeto intersectado y un vector donde se almacenarán los objetos seleccionables de la escena:

```
var raycaster = new THREE.Raycaster();
var mouse = new THREE.Vector2();
var INTERSECTED;
var objects = [];
```

A continuación, hay que asegurarse de que todos los botones que queramos que se puedan seleccionar estén dentro del vector 'objects'. En la pantalla de inicio, los únicos botones que nos interesa añadir al vector son el de jugar y el de editar:

```
objects.push(botonJugar);
objects.push(botonEditar);
```

Los eventos del ratón que se van a monitorizar en el programa son la pulsación (*mousedown*) y liberación (*mouseup*) de cualquiera de sus botones y el movimiento del puntero por la pantalla de la escena (*mousemove*). La forma de hacerlo es añadiendo escuchadores de eventos al objeto 'window':

```
window.addEventListener('mousedown', onMouseDown, false);
window.addEventListener('mouseup', onMouseUp, false);
window.addEventListener('mousemove', onMouseMove, false);
```

El siguiente paso es crear las funciones que se ejecutarán en base a los eventos del ratón: *onMouseDown*, *onMouseUp* y *onMouseMove*.

La función ***onMouseDown*** crea un vector 3D con las componentes X e Y del puntero del ratón y el valor 0'5. El vector es proyectado inversamente con la cámara de la escena. A continuación, se crea un rayo (*raycaster*) cuyo origen es la posición 3D de la cámara y su dirección es el vector anterior al cual se le ha restado la posición de la cámara y se le ha normalizado. Por último, se crea un vector de objetos donde se almacenan todos los objetos intersectados con el rayo que hemos creado, ordenados de menor a mayor distancia respecto al origen del rayo. En el siguiente fragmento de código se puede apreciar cómo se utiliza el rayo y el vector de objetos intersectados para actuar en función del objeto más cercano al rayo:

```
function onMouseDown( event ) {

   var vector = new THREE.Vector3(mouse.x,
                                  mouse.y,
                                      0.5).unproject(camera);

   var raycaster = new THREE.Raycaster(camera.position,
      vector.sub(camera.position).normalize());

   var intersects = raycaster.intersectObjects(objects);

   // Si se ha pulsado algún botón:
   if (intersects.length > 0) {
      // Si el nombre del material del botón es 'botonJugar':
      if (intersects[0].object.material.name == 'botonJugar') {
         // Se elimina el menu1 de la escena
         scene.remove(menu1);
         // Se vacía el vector de objetos (botones) seleccionables
         objects = [];
```

```
            // Se introducen los siguientes objetos seleccionables
            objects.push(botonM1);
            objects.push(botonM2);
            objects.push(botonVolver1);
            // Se añade el menu2 a la escena
            scene.add(menu2);
        } // ...
    }
}
```

La función ***onMouseUp*** se encarga únicamente de devolver el estilo del cursor del ratón a su estado original (flecha):

```
function onMouseUp( event ) {
    // Modifica el estilo del cursor del contenedor del canvas
    container.style.cursor = 'auto'; // Estilo por defecto
}
```

La función ***onMouseMove*** se utiliza para cambiar el color del material de los botones cuando se pasa el cursor sobre ellos y para hacer aparecer o desaparecer elementos en la escena como el vehículo de la *Figura 16*. Esta función se ejecuta cada vez que cambia la posición del cursor sobre la pantalla, es decir, cuando movemos el ratón. El objetivo consiste en actualizar la dirección del rayo que intersecta la cámara con el cursor de la escena para calcular qué objetos quedan intersectados por este rayo.

Figura 16.- Cuando se pasa el cursor sobre el botón 'Furia roja', aparece en la escena el modelo 3D y sus estadísticas 'Velocidad' y 'Aceleración'.

Aquí entra en juego la variable 'INTERSECTED' que se ha definido al comienzo del programa. Esta variable se utiliza para almacenar el último objeto intersectado y devolverle su color original en caso de que se intersecte un objeto distinto a él en la siguiente llamada a la función *onMouseMove*. El siguiente fragmento de código muestra cómo se calculan las componentes X e Y del ratón y cómo influye la variable 'INTERSECTED' en el cambio de color de los botones:

```
function onMouseMove( event ) {

    // Se calculan las componentes X e Y del ratón (de -1 hasta +1)
    mouse.x = (event.clientX / window.innerWidth) * 2 - 1;
    mouse.y = - (event.clientY / window.innerHeight) * 2 + 1;

    // Se actualiza el rayo con las nuevas componentes del ratón
    raycaster.setFromCamera(mouse, camera);
```

```
    // Se calcula qué objetos intersecta el rayo
    var intersects = raycaster.intersectObjects(objects);

    // Si el rayo intersecta algún botón:
    if (intersects.length > 0) {
       // Si el botón intersectado anterior es diferente del actual:
       if (INTERSECTED != intersects[0].object) {
          // Se devuelve el color original grisáceo al
          // botón intersectado anterior, si lo había
          if (INTERSECTED) INTERSECTED.material.color.setHex(0xE4E4E4);
          if (intersects[0].object.material.name == 'cpu') {
             scene.remove(planoM2);
             scene.add(planoM1);
          } else {
             scene.remove(planoM1);
             // ...
          }
          // Como estoy encima de un botón, lo resalto de blanco
          intersects[0].object.material.color.setHex(0xFFFFFF);
          // El botón intersectado pasa a ser el actual
          INTERSECTED = intersects[0].object;
       }
       // Modifica el estilo del cursor por el de una mano
       container.style.cursor = 'pointer';

    } else { // Si el rayo no intersecta ningún botón:
       scene.remove(planoM1);
       // ...
       // Se devuelve el color original grisáceo al
       // botón intersectado anterior, si lo había
       if (INTERSECTED) INTERSECTED.material.color.setHex(0xE4E4E4);
       // Ya no hay ningún botón intersectado
       INTERSECTED = null;
       // Se devuelve el cursor a su estilo por defecto
       container.style.cursor = 'auto';
    }
}
```

La implementación de esta tarea no habría sido posible de no ser por el tutorial y los ejemplos que aparecen en la página web de Soledad Penadés [7]. La autora introduce al lector en la teoría básica sobre el trazado de rayos para seleccionar o incluso desplazar objetos por una escena 3D como la de Three.js.

Tarea 8. Implementación del editor interactivo de circuitos 3D

Tras programar la interfaz de los menús, el siguiente paso es crear un segundo programa que se encargue de gestionar un editor interactivo de circuitos en 3D. El nombre de este programa es editor.js. Con él, el usuario dispone de un panel con herramientas para ensamblar unas piezas con otras hasta crear un circuito cerrado. Además, podrá guardar el circuito en un fichero de texto.

La *Figura 17* muestra la interfaz del editor con un circuito a medio construir. Como se puede observar, está compuesta por varios botones con los que poder crear piezas (botones con fondo azul) y modificarlas (botones negros transparentes).

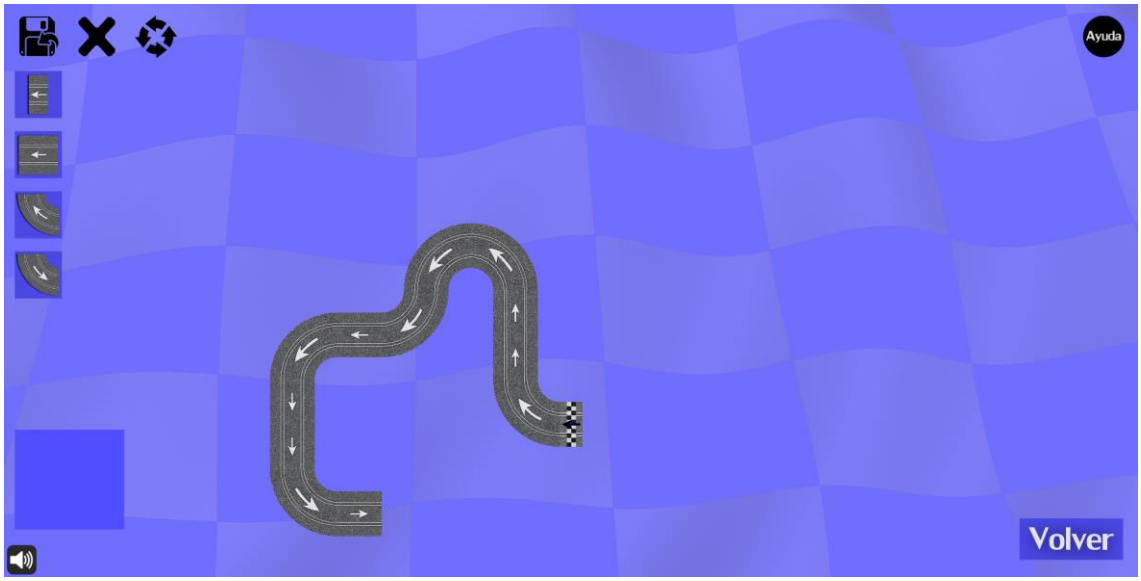


Figura 17.- Interfaz del editor interactivo de circuitos 3D.

La mecánica del editor es muy sencilla. Cuando se carga, lo único que aparece es la pieza de salida. Su posición se puede modificar si se desea haciendo clic sobre ella y desplazándola con el ratón. Para que no haya varias piezas sueltas por la escena, solamente se permite la existencia de una pieza sin colocar. Si se quiere ensamblar la siguiente pieza, ésta debe coincidir con la dirección y el sentido de la pieza anteriormente colocada. Por ejemplo, como el sentido de la pieza de salida es siempre hacia la izquierda, solamente podremos colocar una recta o una curva que sigan este mismo sentido.

Para colocar una nueva pieza, el primer paso es pulsar sobre uno de los botones superiores con dibujos de piezas. Tras hacer esto, la nueva pieza es cargada en el recuadro azul de la esquina inferior izquierda de la pantalla. El siguiente paso es hacer clic sobre ella y mantener pulsado el botón del ratón para arrastrarla hasta la última pieza que se colocó en el circuito. Cuando la nueva pieza está lo suficientemente cerca de la vieja, se colorea de verde para indicar al usuario que ya puede soltarla. No es necesario que el usuario las haga coincidir perfectamente. El programa se encarga de hacer esta tarea por él, pues no sería sencillo ni rápido que el usuario tuviera que ensamblar las piezas a la perfección.

El editor cuenta con un botón de ayuda donde se explican brevemente las instrucciones que aquí se han detallado. Las herramientas disponibles son las siguientes:

 *Guardar el circuito en el ordenador*

Este botón se podrá utilizar únicamente cuando el final de la última pieza colocada coincida con el principio de la pieza de salida. Su función es almacenar los

atributos de las piezas en un fichero de texto plano en el ordenador cliente para poder importarlo desde el menú de selección de circuito y jugar en él.

 *Eliminar la última pieza creada*

Al pulsar sobre este botón se elimina de la escena la última pieza creada, se encuentre colocada en el circuito o no. La única pieza que no elimina este botón es la de salida.

 *Rotar 90º la pieza que está por colocar*

Cada vez que se pulsa sobre este botón, se aplica una rotación de 90º en el sentido opuesto a las agujas del reloj sobre la última pieza que se ha creado pero que todavía no ha sido adherida al circuito.

Mostrar el panel de ayuda

Al pulsarlo aparece un nuevo botón con una imagen donde se explican brevemente las instrucciones del editor. Para cerrarlas no hay más que pulsar sobre este nuevo botón.

Al igual que sucede con el programa juego.js, el programa editor.js hace uso de los mismos eventos del ratón: ***mousedown***, ***mouseup*** y ***mousemove***. La diferencia es que para desplazar las piezas por la escena se necesitan nuevos elementos. El más importante es el plano invisible en 2D por el que se van a desplazar:

```
// Plano por el que se desplazan los objetos seleccionables
var plano = new THREE.Mesh(
   // Geometría del plano (2000x2000)
   new THREE.PlaneBufferGeometry(2000, 2000),
   // Material del plano
   new THREE.MeshBasicMaterial({opacity: 0.25, transparent: true})
);
// Se hace invisible
plano.visible = false;
// Se añade a la escena
scene.add(plano);
```

Además del plano y la variable 'INTERSECTED', se requiere una variable que almacene la pieza que está siendo desplazada. A esta variable se le ha llamado 'SELECTED':

```
var SELECTED;
```

Ahora, cuando se pasa el cursor sobre una pieza (*mousemove*), si la pieza no ha sido todavía seleccionada (SELECTED), la posición del plano se actualiza con la

posición de la pieza intersectada (*INTERSECTED*). A continuación, se modifica la dirección del plano para que apunte hacia la cámara. De esta forma, las piezas se desplazarán paralelamente a la vista del usuario (función *onMouseMove*):

```
plano.position.copy(INTERSECTED.position);
plano.lookAt(camera.position);
```

Cuando se pulsa sobre una pieza (*mousedown*), la variable 'SELECTED' toma su valor (función *onMouseDown*):

```
SELECTED = piezaCircuitoIntersectada[0].object;
```

Volviendo de nuevo a la función *onMouseMove*, si existe una pieza seleccionada por el ratón, se le aplica una translación a través del plano 2D correspondiente a su posición anterior menos el desplazamiento (*offset*):

```
if (SELECTED) {
   var piezaIntersectada = raycaster.intersectObject(plano);
   SELECTED.position.copy(piezaIntersectada[0].point.sub(offset));
   return;
}
```

Para finalizar con esta tarea, se procede a explicar qué ocurre cuando soltamos una pieza nueva cerca de la última colocada en el circuito. Como ya se ha comentado, la función encargada del evento (*mouseup*) se ha llamado *onMouseUp*. Ahora, cuando se suelta el botón del ratón con el que teníamos seleccionada una pieza, si se encuentra cerca de la última pieza colocada en el circuito, se colocará a continuación de ella.

Como cada pieza es de un tamaño diferente, resulta completamente necesario programar cada una de las posibles combinaciones de piezas, pues en función de la dirección y el sentido, se tendrá que aplicar una translación u otra para que encajen perfectamente. El siguiente fragmento de la función *onMouseUp* muestra un ejemplo de lo que ocurre si se quiere pegar una pieza a continuación de una recta cuyo sentido es hacia arriba (recta_1):

```
if (ultimaPiezaColocada.name == 'recta_1') {
   if (piezaParaColocar.name == 'recta_1') {
      piezaParaColocar.position.set(
      ultimaPiezaColocada.position.x-1,
      ultimaPiezaColocada.position.y,
      ultimaPiezaColocada.position.z);
   } else if (piezaParaColocar.name == 'rectaDoble_1') {
      piezaParaColocar.position.set( Translación distinta );
   } else if (piezaParaColocar.name == 'curva90_1') {
      piezaParaColocar.position.set( Translación distinta );
   } else if (piezaParaColocar.name == 'curva90_5') {
      piezaParaColocar.position.set( Translación distinta );
   } else {
     // Mostrar mensaje de incompatibilidad de piezas
   }

} // Y así para el resto de piezas
```

En la *Tabla 1* se muestran los nombres internos de todas las posibles piezas del circuito en función de su rotación. Es por ello que en el ejemplo anterior solamente se puede añadir a una ***recta_1*** las piezas ***recta_1***, ***rectaDoble_1***, ***curva90_1*** o ***curva90_5***.

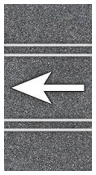
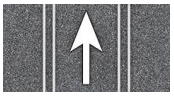
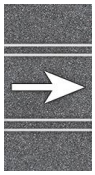
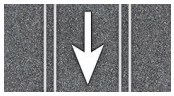
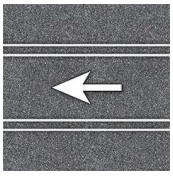
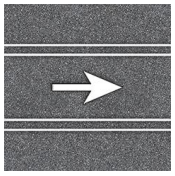
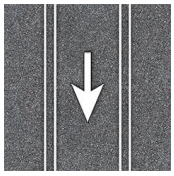
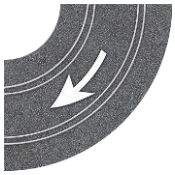
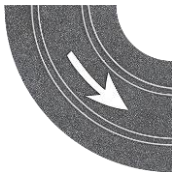
			
recta_1	recta_2	recta_3	recta_4
			
rectaDoble_1	rectaDoble_2	rectaDoble_3	rectaDoble_4
			
curva90_1	curva90_2	curva90_3	curva90_4
			
curva90_5	curva90_6	curva90_7	curva90_8

Tabla 1.- Distribución de las posibles piezas a colocar en el editor de circuitos.

Tarea 9. Exportación e importación de circuitos construidos

La existencia del editor de circuitos no tendría mucho sentido si después no pudiéramos llevarnos con nosotros las pistas que vamos construyendo. Para evitar que el usuario tenga que crear una pista cada vez que quiera jugar en un circuito personalizado, se ha implementado una mecánica de exportación e importación de circuitos en el videojuego.

Para exportar un circuito, la única condición que debe cumplirse es que la última pieza colocada coincida perfectamente con la pieza de salida. Si la condición no se cumple y se pulsa el botón de guardar, aparecerá una alerta indicando que antes debemos cerrar el circuito perfectamente. Como la rotación de la pieza de salida es invariable (***recta_1***), las únicas piezas con las que se puede cerrar el circuito son ***recta_1***, ***rectaDoble_1***, ***curva90_4*** o ***curva90_6***. La comprobación de la condición se lleva a cabo en la función *onMouseDown*:

```
// Si el nombre del material del botón intersectado es 'guardar'
if (botonIntersectado[0].object.material.name == 'guardar') {
   if (ultimaPiezaColocada.name == 'recta_1') {
      // Si la coordenada X del centro de la pieza colocada se
      // encuentra a una unidad de distancia positiva de la
      // coordenada X del centro de la pieza de meta/salida,
      // entonces se ha cerrado el circuito con esta última pieza
      if (ultimaPiezaColocada.position.x-1 == piezaMeta.position.x) {
         // Se llama a la función que guarda el circuito
         guardarCircuito();
      } else {
         // Se llama a la función que muestra un mensaje de alerta
         errorCircuitoNoCerrado();
      }
   } // Misma mecánica con las otras 3 posibles piezas:
     // 'rectaDoble_1', 'curva90_4' y 'curva90_6'
   else {
      // Si las condiciones no se cumplen, se llama a la función
      // que muestra un mensaje de alerta
      errorCircuitoNoCerrado();
   }
}
```

Como se puede observar, cuando el circuito queda cerrado, se llama a la función 'guardarCircuito'. Esta función recorre el vector de piezas que componen el circuito ('listaPiezasCircuito') para construir dinámicamente una cadena de texto ('cadenaPosPiezas') con el nombre, las posiciones XYZ y la rotación en Y que tiene cada una de las piezas:

```
function guardarCircuito() {

   // Cadena de texto con los nombres, las posiciones centrales
   // (componentes XYZ) y las rotaciones (en el eje Y) de las piezas
   cadenaPosPiezas = '';

   // Bucle que recorre cada pieza del circuito
   for (var i=0; i < listaPiezasCircuito.length; i++) {
      // Actualizamos la cadena de texto con los atributos
      // de cada pieza (nombre, posición y rotación)
      cadenaPosPiezas += listaPiezasCircuito[i].name + ' ' +
      listaPiezasCircuito[i].position.x + ' ' +
      listaPiezasCircuito[i].position.y + ' ' +
      listaPiezasCircuito[i].position.z + ' ' +
      listaPiezasCircuito[i].rotation.y + '\n';
   }

   // Biblioteca FileSaver.js
   var blob = new Blob([cadenaPosPiezas],
      {type: "text/plain;charset=utf-8;", });

   // Se guarda la cadena de texto en el fichero 'Circuito.txt'
   saveAs(blob, "Circuito.txt");
}
```

Una vez construida la cadena de texto, se almacena en un fichero llamado 'Circuito.txt', el cual se ubica en la carpeta predeterminada de descargas que tiene asignada el navegador web. De esta forma, el usuario podrá importar el circuito creado más tarde cuando seleccione el botón 'Circuito Personalizado' desde el

menú de selección de circuito. La biblioteca que permite exportar la cadena de texto en el ordenador del cliente se llama FileSaver.js. Al igual que el resto de bibliotecas, para añadirla al proyecto no hay más que situar su ubicación en la página HTML que contenga el programa que vaya a utilizarla (en este caso editor.js). El siguiente fragmento de código pertenece al fichero editor.html:

```
<!-- Biblioteca gráfica -->
<script src = "./recursos/lib/three.js"></script>

<!-- Control de la cámara -->
<script src = "./recursos/lib/OrbitControls.js"></script>

<!-- Interacción con el ratón -->
<script src = "./recursos/lib/Raycaster.js"></script>

<!-- Exportación de ficheros -->
<script src = "./recursos/lib/FileSaver.js"></script>

<!-- Programa que utiliza las bibliotecas anteriores -->
<script src = "./recursos/js/editor.js"></script>
```

En la *Figura 18* se puede observar la estructura que adopta la cadena de texto con los atributos de las piezas ('cadenaPosPiezas') cuando se abre el fichero 'Circuito.txt' con el editor de textos Notepad++. Cada línea de este fichero está compuesta por los 5 atributos que se necesitan de cada pieza para poder reconstruir el circuito a la hora de importar este fichero en el programa juego.js. Cabe destacar que entre un atributo y otro se ha colocado un espacio en blanco para poder manipularlos con facilidad.

```
1  recta_1 0 -3 6 0
2  rectaDoble_1 -1.5 -3 6 0
3  curva90_1 -4 -3 5.5 0
4  curva90_6 -5 -3 2.5 3.141592653589793
5  curva90_1 -8 -3 1.5 0
6  curva90_6 -9 -3 -1.5 3.141592653589793
7  curva90_1 -12 -3 -2.5 0
8  rectaDoble_2 -12.5 -3 -5 4.71238898038469
9  rectaDoble_2 -12.5 -3 -7 4.71238898038469
10 curva90_2 -12 -3 -9.5 4.71238898038469
11 rectaDoble_3 -9.5 -3 -10 3.141592653589793
12 rectaDoble_3 -7.5 -3 -10 3.141592653589793
13 rectaDoble_3 -5.5 -3 -10 3.141592653589793
14 curva90_3 -3 -3 -9.5 3.141592653589793
15 curva90_8 -2 -3 -6.5 0
16 curva90_7 1 -3 -6.5 1.5707963267948966
17 curva90_2 2 -3 -9.5 4.71238898038469
18 rectaDoble_3 4.5 -3 -10 3.141592653589793
19 curva90_3 7 -3 -9.5 3.141592653589793
20 curva90_8 8 -3 -6.5 0
21 curva90_3 11 -3 -5.5 3.141592653589793
22 curva90_8 12 -3 -2.5 0
23 curva90_3 15 -3 -1.5 3.141592653589793
24 rectaDoble_4 15.5 -3 1 1.5707963267948966
25 rectaDoble_4 15.5 -3 3 1.5707963267948966
26 curva90_4 15 -3 5.5 1.5707963267948966
27 rectaDoble_1 12.5 -3 6 0
28 rectaDoble_1 10.5 -3 6 0
29 rectaDoble_1 8.5 -3 6 0
30 rectaDoble_1 6.5 -3 6 0
31 rectaDoble_1 4.5 -3 6 0
32 rectaDoble_1 2.5 -3 6 0
33 recta_1 1 -3 6 0
```

Figura 18.- Fichero de texto con los atributos de las 33 piezas que componen el circuito 'Glaciar Fresco'.

Tal y como se ha comentado, el primer atributo es el nombre interno de la pieza. Los atributos segundo, tercero y cuarto se corresponden con las posiciones X, Y y Z de la pieza. El quinto y último atributo almacena la rotación en radianes del eje Y de la pieza respecto a su rotación original (0 radianes).

Por ejemplo, podemos decir que la pieza número 19 del circuito 'Glaciar Fresco' (*Figura 18*) se trata de una curva de 90º, cuyas componentes XYZ en la escena del videojuego deben ser (7, -3, -9.5) y su ángulo de rotación sobre el eje Y tiene que ser de 180º (3'141592... o simplemente π).

A continuación, tras la exportación del circuito, el siguiente objetivo era que el usuario pudiera importarlo cuando quisiera para utilizarlo en cualquier modo de juego. Desgraciadamente, el lenguaje JavaScript es asíncrono, lo que significa que no se puede garantizar el orden de ejecución de las cargas de modelos 3D que se hagan dentro de un mismo bucle. Por este motivo, la importación de circuitos se ha dejado aplazada como trabajo futuro para poder completar el resto de tareas del proyecto.

Tarea 10. Desarrollo del simulador de conducción y sus modos

En esta tarea se trata el aspecto más importante del videojuego: la implementación del simulador de carreras sobre raíles. El programa encargado de llevar a cabo esta tarea es carrera.js.

El primer problema que aparece es que el simulador deberá reaccionar de una forma u otra en función del modo de juego, el circuito y los vehículos seleccionados. Como ya se ha visto, esta serie de selecciones se realizan en el programa previo a carrera.js, que es juego.js. Para conseguir transportar esta información de un programa a otro, se ha utilizado la técnica de la 'cadena de consulta', más conocida como '*query string*' [8].

Esta técnica consiste en construir una cadena de texto (*query string*) con las variables del programa juego.js como por ejemplo el modo de juego o el circuito seleccionado para 'pegarla' a continuación de la URL que se encarga de ejecutar el programa carrera.js. La URL que pone en marcha el simulador de carreras es www.webscalextric.com/carrera.html. El formato de la *query string* es el siguiente:

```
var queryString =
   '?' + modoSeleccionado +
   '&' + vehiculoSeleccionadoJ1 +
   '&' + vehiculoSeleccionadoJ2 +
   '&' + circuitoSeleccionado + '';
```

Como se puede ver, la cadena comienza siempre por el signo '?', seguido de todas las variables separadas siempre por algún carácter (en este caso '&'). A continuación se muestra un ejemplo de URL junto a una *query string*:

www.webscalextric.com/carrera.html*?cpu&furiaRoja&centellaAzul&llanuraPlana*

Una vez que se ha construido una URL como la del ejemplo, el programa carrera.js recupera la información de la *query string* para almacenar cada parámetro en su variable correspondiente.

El siguiente objetivo es cargar todos y cada uno de los modelos 3D, texturas e imágenes necesarias que componen la escena del simulador. Si el circuito elegido es 'Llanura plana', se cargarán unas piezas y elementos del escenario diferentes a si se trata del circuito 'Glaciar fresco' o uno personalizado.

Para que el usuario no vea cómo la escena va cargando las texturas hasta que todo se ve correctamente, se ha implementado una función de carga que muestra un cartel de 'Cargando' mientras el número de objetos cargados no sea igual al número de objetos totales a cargar. Para llevar a cabo esta técnica se ha imitado uno de los ejemplos que se proporcionan en la web www.alteredqualia.com [9].

Básicamente, la idea es que cada función que se ejecuta para cargar una textura ('loadTexture'), avise de que ha acabado e incremente la variable que almacena el número de objetos cargados hasta el momento:

```
var textura =
   new THREE.ImageUtils.loadTexture("./tx.png", cargandoCallback());

// Función que escucha la finalización de la carga de cada textura
function cargandoCallback() {
   // Número de objetos cargados hasta el momento
   numObjetos += 1;
   // Si ya se ha cargado todo, se oculta el cartel de 'Cargando...':
   if (numObjetos == numObjetosTotales) {
      planoCargando.style.display = "none";
      // Se cargan los elementos del HUD, el botón 'Comenzar'
      // y la imagen con los controles de los vehículos
   }
}
```

Cuando el número de objetos cargados coincide con el total, se elimina el cartel de 'Cargando' y se coloca un botón de comienzo. Al pulsar sobre él, se inicia la función encargada de mostrar una cuenta atrás. Cuando el contador llega a 0, la carrera comienza y se activan los eventos del teclado para capturar la pulsación de las teclas de cada jugador gracias a la biblioteca KeyboardState.js.

En los siguientes párrafos se explicará cómo se mueven los vehículos por el circuito y cómo se ha implementado la inteligencia artificial del vehículo controlado por el ordenador, entre otros aspectos.

Simulador de conducción

Para que los vehículos se desplacen por la trayectoria que siguen sus raíles, el primer paso consiste en crear una polilínea para cada uno. Una polilínea es una

sucesión de puntos 3D cuyo último punto se enlaza con el primero. Para generarlas, se ha implementado una función llamada 'extraccionPolilineas'. La función se ejecuta tras cargarse los modelos de las piezas 3D del circuito.

El propósito es recorrer cada una de las piezas y generar los puntos 3D correspondientes a sus 2 polilíneas. En la *Figura 19* se han dibujado los puntos sobre la escena para que se aprecien cuáles son las trayectorias de ambas polilíneas. La roja es para el vehículo del jugador 1 y la azul para el del jugador 2.

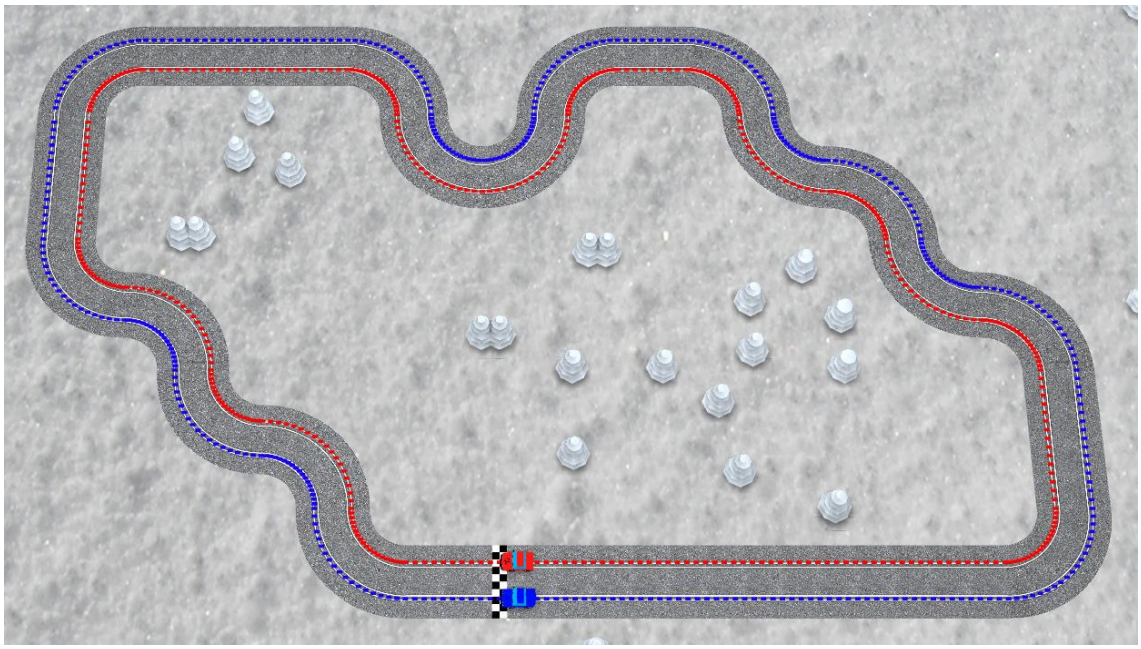


Figura 19.- Polilíneas de cada uno de los vehículos.

Una vez que se han construido los caminos por los que se van a desplazar los vehículos, llega el momento de diseñar el movimiento de los mismos.

El primer paso es calcular la velocidad de los vehículos, la cual se descompone en 2 variables: magnitud y dirección.

Tras el pistoletazo de salida, cuando el jugador pulsa la tecla para acelerar, se incrementa la velocidad de su vehículo con el valor de su aceleración particular, a menos que se haya alcanzado su velocidad punta. Cabe destacar que cada vehículo cuenta con un valor de aceleración y un valor de velocidad punta propio. En cambio, la velocidad de frenada es la misma para todos los vehículos, aunque se puede modificar en un futuro. De manera opuesta, cuando el jugador suelta la tecla de aceleración, se decrementa la velocidad con el valor de la velocidad de frenada, a menos que se haya alcanzado la velocidad mínima, que es 0.

A continuación, se debe calcular la dirección de la velocidad que debe adoptar el vehículo. Esta dirección viene dada por el vector unitario (U(AB)) entre el punto actual de la polilínea en la que se encuentra el vehículo (punto A) y el siguiente

punto al que debe llegar (punto B). La fórmula del vector unitario entre estos puntos es la siguiente:

$$U(AB) = \frac{AB}{|AB|}$$

donde los términos AB y |AB| se calculan como:

$$AB = (B.x - A.x\,,\ B.y - A.y,\ B.z - A.z)$$

$$|AB| = \sqrt{AB.x \cdot AB.x\ +\ AB.y \cdot AB.y\ +\ AB.z \cdot AB.z}$$

Una vez que tenemos la magnitud y la dirección de la velocidad calculada, el siguiente paso es obtener el vector desplazamiento que originarían estas variables aplicadas sobre el vehículo. El cálculo del vector desplazamiento (v) se realiza de la siguiente forma:

$$v = (U(AB).x \cdot espacio\,,\ U(AB).y,\ U(AB).z \cdot espacio)$$

donde el espacio se calcula como:

$$espacio = velocidad \cdot tiempo$$

Como se puede ver en la fórmula del vector, la componente 'Y' del mismo no se multiplica por el espacio recorrido. Esto es debido a que el vehículo se desplaza siempre por el plano XZ, ya que las piezas son todas planas y en ningún momento hay que elevarlo del suelo. La variable del tiempo tiene siempre el mismo valor, pues se corresponde con la velocidad de renderizado de la escena.

Por último, solamente queda actualizar la posición del vehículo en la escena. Para ello, se suma a su posición actual el vector desplazamiento calculado previamente:

$$posición = (vehículo.x + v.x\,,\ vehículo.y,\ vehículo.z + v.z)$$

El siguiente punto a tratar es la salida de los vehículos de su trayectoria. El reto que presenta el videojuego al jugador es intentar mantener la máxima velocidad posible sobre los raíles sin que el vehículo descarrile. Un vehículo descarrila cuando la distancia existente entre su centro geométrico y el siguiente punto de su polilínea es mayor que cierto umbral. Cuando el vehículo descarrila, su velocidad se anula y se recoloca automáticamente en el punto donde se salió de la trazada, teniendo el jugador que acelerar desde el reposo tal y como pasaría en la realidad.

Por último, para conseguir que los vehículos giren en las curvas se ha observado el signo que toma la componente 'Z' de los vectores que componen las direcciones de sus velocidades (vectores U(AB)). Si 'Z' es menor o igual a 0, la dirección de la velocidad se multiplica por -90º. Si 'Z' es mayor o igual a 0, se multiplica por 90º y se le suma una rotación de 180º para que cuadre con el sentido de la dirección.

Inteligencia Artificial (IA) del vehículo controlado por el ordenador

Si sumamos que prácticamente cualquier videojuego incluye algún componente que toma ciertas decisiones o imita el comportamiento humano y que una de las ramas más importantes de este máster cursado es la inteligencia artificial, se ha decidido implementar un modelo de conducción automático contra el que poder competir como jugador humano.

Este modelo está incluido en el modo de juego para un solo jugador llamado 'Contra el ordenador'. Siempre viene muy bien un modo de juego individual cuando queremos competir pero no tenemos a nadie cerca.

Para implementar la inteligencia artificial del vehículo controlado por el ordenador, se ha imitado el comportamiento de conducción que tomaría un humano.

Si la dirección de la velocidad del vehículo pertenece a la de una recta, se incrementa su velocidad hasta alcanzar la máxima. Como en las rectas el vehículo no se va a salir nunca por más velocidad que lleve, es lógico acelerar al máximo para recortar distancia al jugador humano.

En cambio, si el vehículo se encuentra en una curva, se le aplicará un decremento de su velocidad como si dejara de pulsar la tecla de aceleración. El decremento se aplica hasta que alcanza cierta velocidad superior a 0 para que no llegue a detenerse (velocidad mínima sin peligro).

Por último, se debe tener también en cuenta lo que ocurre cuando el vehículo se sale en una curva. Como al recolocarse en el circuito después de una salida su velocidad es 0, el vehículo debe acelerar desde el reposo hasta alcanzar la velocidad mínima establecida para circular por las curvas sin peligro.

Tarea 11. Elementos del HUD (*Heads-Up Display*)

Los elementos que componen el HUD de un videojuego son los rótulos, carteles, imágenes o cualquier contenido audiovisual que aporte información de interés al usuario sobre el estado del juego. Generalmente, estos elementos se encuentran estáticos en un plano distinto y por delante de la escena donde se desarrolla la acción.

Los elementos básicos que aparecen en un HUD de un videojuego de carreras suelen ser la posición actual en carrera, el número de vueltas dadas y las totales o el tiempo transcurrido desde el comienzo de la carrera.

La estrategia seguida para implementar el HUD consiste en la combinación de código HTML con hojas de estilo CSS y código JavaScript. La *Figura 20* muestra una captura del juego donde se puede apreciar que el jugador 1 va en segunda posición y que los dos vehículos se encuentran en la misma vuelta (vuelta 1 de 4). La tecla que debe pulsar el jugador 1 para acelerar es la 'A', mientras que la del jugador 2 (en este caso el ordenador) es la 'L'. Además, en la esquina inferior izquierda aparecen 3 botones interactivos cuya funcionalidad e implementación se detalla en las Tareas 12, 13 y 14.

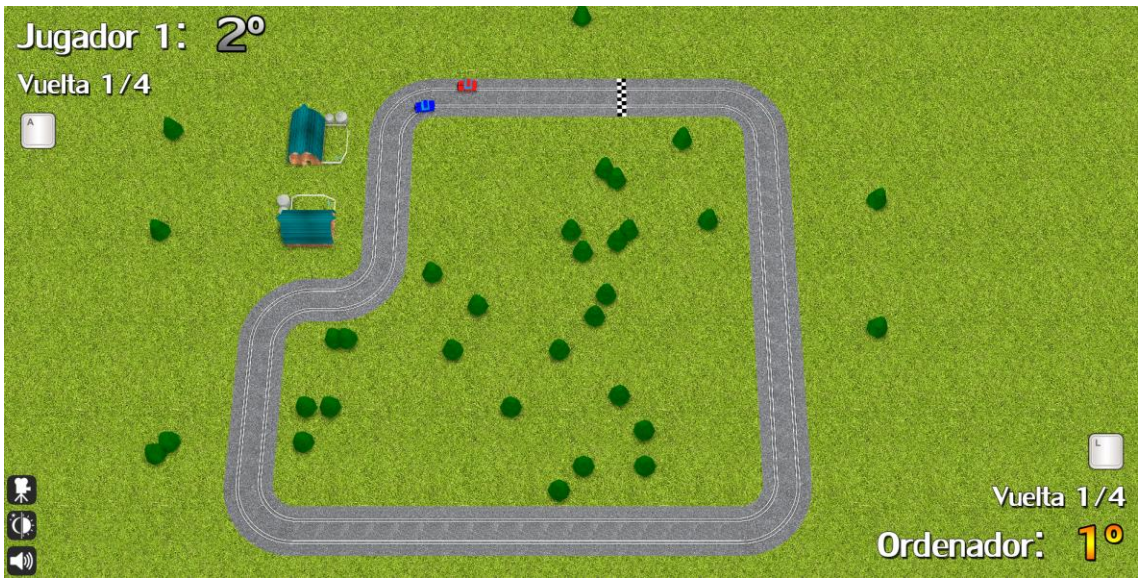


Figura 20.- Captura de pantalla del simulador donde se aprecian los elementos del HUD como la posición de los vehículos y los botones interactivos.

Los ficheros que intervienen en la implementación del HUD son los siguientes: **carrera.html**, **main.css** y **carrera.js**.

El fichero carrera.html se utiliza únicamente para crear secciones en la web con las que poder interactuar de manera individual a través del código de los siguientes dos ficheros. A estas secciones se les otorga un identificador (id='nombre') para poder hacer referencia a ellas individualmente.

carrera.html

```
<!-- Posiciones de los jugadores -->
<div id='posJ1'> </div>
<div id='posJ2'> </div>
<!-- Vueltas actuales -->
<div id='lapJ1'> </div>
<div id='lapJ2'> </div>
```

```
<!-- Teclas de aceleración -->
<div id='keyJ1'> </div>
<div id='keyJ2'> </div>
```

A continuación, situamos cada una de las secciones en un lugar concreto de la pantalla mediante el fichero main.css. Por ejemplo, las posiciones en carrera de cada jugador ('posJ1' y 'posJ2') se sitúan en las esquinas superior izquierda e inferior derecha, respectivamente.

main.css

```
/* La posición fixed hace que no se mueva ni al hacer scroll */

/* Con top y left a 0px se fuerza al elemento a situarse en la
esquina superior izquierda */
#posJ1 { position: fixed; top: 0px; left: 0px; }

/* Con bottom y right a 0px se fuerza al elemento a situarse en
la esquina inferior derecha */
#posJ2 { position: fixed; bottom: 0px; right: 0px; }

/* Algo más bajo que posJ1 */
#lapJ1 { position: fixed; top: 100px; left: 10px; }

/* Algo más alto que posJ2 */
#lapJ2 { position: fixed; bottom: 100px; right: 10px; }

/* Algo más bajo que lapJ1 */
#keyJ1 { position: fixed; top: 175px; left: 20px; }

/* Algo más alto que lapJ2 */
#keyJ2 { position: fixed; bottom: 175px; right: 20px; }
```

Finalmente se accede en tiempo real a los valores de estas secciones mediante el programa carrera.js. El objetivo es cambiar el código HTML que contiene la imagen de cada sección por la que sea oportuna en función de ciertas variables del juego.

Las variables que afectan en el cambio de las posiciones son 'espacioRecorridoJ1' y 'espacioRecorridoJ2'. Estas variables almacenan la cantidad de puntos alcanzados de las polilíneas del circuito de cada jugador. Si el jugador 1 ha alcanzado más puntos que el jugador 2, es porque ha recorrido más distancia, y por lo tanto, va en primera posición.

En cambio, las variables que influyen en el cambio del número de vueltas dadas son 'vueltaJ1' y 'vueltaJ2'. Cada vez que un vehículo pasa por el último punto de su polilínea, se actualiza su número de vueltas, lo que conlleva a la actualización de la imagen que se muestra en su HUD.

carrera.js

```
// Se almacena el valor de los elementos HTML en variables
var posJ1 = document.getElementById('posJ1');
var posJ2 = document.getElementById('posJ2');
```

```
var lapJ1 = document.getElementById('lapJ1');
var lapJ2 = document.getElementById('lapJ2');
var keyJ1 = document.getElementById('keyJ1');
var keyJ2 = document.getElementById('keyJ2');


// ...

// Esta función se ejecuta continuamente
function update() {

   // ...

   // Si el jugador 1 ha recorrido más espacio que
   // el 2 es porque va en cabeza:
   if (espacioRecorridoJ1 >= espacioRecorridoJ2) {

      // Se recupera el valor del elemento HTML 'posJ1'
      posJ1 = document.getElementById('posJ1');
      // Se le inserta la imagen correspondiente mediante código HTML
      posJ1.innerHTML = '<img src="./Jug1Primero.png">';

      // Lo mismo para la posición del jugador 2
      posJ2 = document.getElementById('posJ2');
      posJ2.innerHTML = '<img src="./Jug2Segundo.png">';

   } // De lo contrario, el jugador 2 es quien va en cabeza:
   else {
      posJ1 = document.getElementById('posJ1');
      posJ1.innerHTML = '<img src="./Jug1Segundo.png">';
      posJ2 = document.getElementById('posJ2');
      posJ2.innerHTML = '<img src="./Jug2Primero.png">';
   }

   // Si el jugador 1 va por su primera vuelta:
   if (vueltaJ1 == 1) {
      lapJ1 = document.getElementById('lapJ1');
      lapJ1.innerHTML = '<img src="./Vuelta1.png">';
   } // Si el jugador 1 va por su segunda vuelta:
   else if (vueltaJ1 == 2) {
      lapJ1 = document.getElementById('lapJ1');
      lapJ1.innerHTML = '<img src="./Vuelta2.png">';
   } // Tercera vuelta:
   else if (vueltaJ1 == 3) {
      lapJ1 = document.getElementById('lapJ1');
      lapJ1.innerHTML = '<img src="./Vuelta3.png">';
   } // Última vuelta:
   else if (vueltaJ1 == 4) {
      lapJ1 = document.getElementById('lapJ1');
      lapJ1.innerHTML = '<img src="./Vuelta4.png">';
   }

   // Si el jugador 2 va por su primera vuelta:
   if (vueltaJ2 == 1) {
      // ...
   }

   // ...

}
```

Tarea 12. Implementación de las cámaras del simulador

El objetivo de esta tarea consiste en situar un botón en el HUD del simulador con el que poder elegir entre una cámara u otra para visualizar la carrera. A continuación se detalla la implementación y el movimiento que sigue cada una de estas cámaras.

La primera de ellas es la cámara estática (*Figura 21*). Su misión es mostrar la totalidad del circuito desde arriba. Para conseguirlo, se sitúa 100 unidades sobre el suelo y 40 unidades hacia el observador para ver el circuito con cierto grado de perspectiva:

```
if (numCamara == 1) {
   camera.position.set(0, 100, 40);
}
```

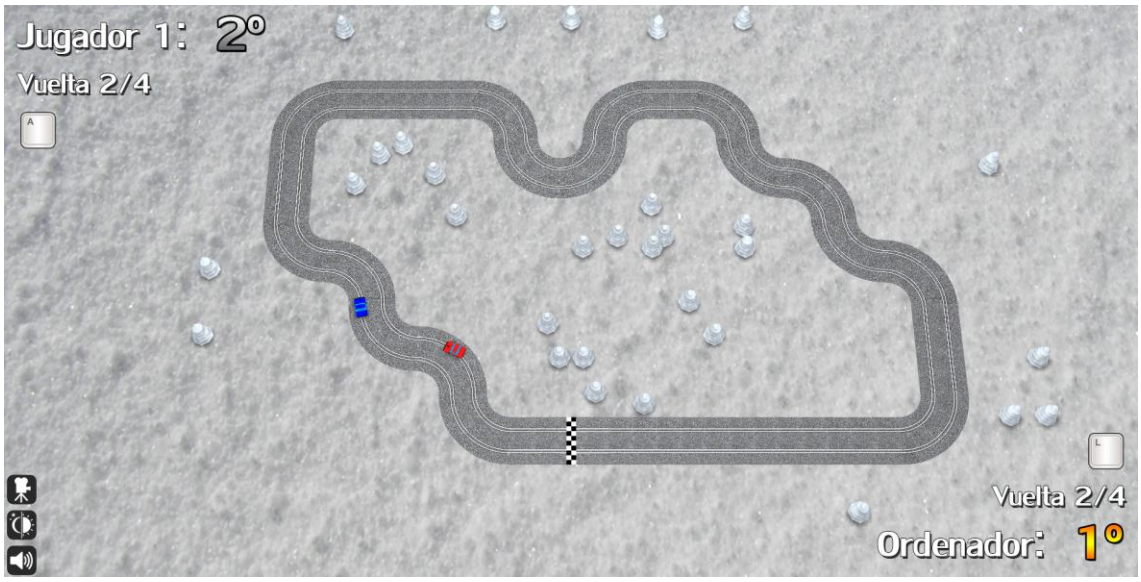


Figura 21.- Vista de la cámara estática.

Esta cámara es la única que comparten los 2 modos de juego ('Contra otro humano' y 'Contra el ordenador'). Las 2 cámaras siguientes son exclusivas del modo 'Contra el ordenador', pues a diferencia de la cámara estática, lo que hacen es seguir la trayectoria del vehículo controlado por el jugador humano.

La segunda cámara implementada (*Figura 22*) también está situada en una posición XYZ fija de la escena, pues se ha situado en el punto (-20, 20, 50). La diferencia que tiene con la primera cámara es que ahora sigue la trayectoria del vehículo. Para conseguirlo, se debe fijar un objetivo con la función 'lookAt':

```
if (numCamara == 2) {
   camera.position.set(-20, 20, 50);
   // La cámara mira en todo momento hacia la posición
   // del vehículo controlado por el jugador humano
   camera.lookAt(vehiculoJugador1.position);
}
```

Figura 22.- Vista de la cámara móvil.

La tercera y última cámara es la cámara cercana (*Figura 23*). A diferencia de las otras, su posición va variando en función de la posición del vehículo del jugador humano. Esta cámara está siempre situada a 10 unidades de distancia del vehículo y elevada a 4 del mismo. Al igual que la cámara móvil, su objetivo es siempre la posición del vehículo:

```
if (numCamara == 3) {
   // Camara situada 10 unidades a la derecha y 4 hacia arriba
   // de la posición central del vehículo del jugador humano
   camera.position.set(
     vehiculoJugador1.position.x+10,
     vehiculoJugador1.position.y+4,
     vehiculoJugador1.position.z);
   // La cámara mira en todo momento hacia la posición del vehículo
   camera.lookAt(vehiculoJugador1.position);
}
```

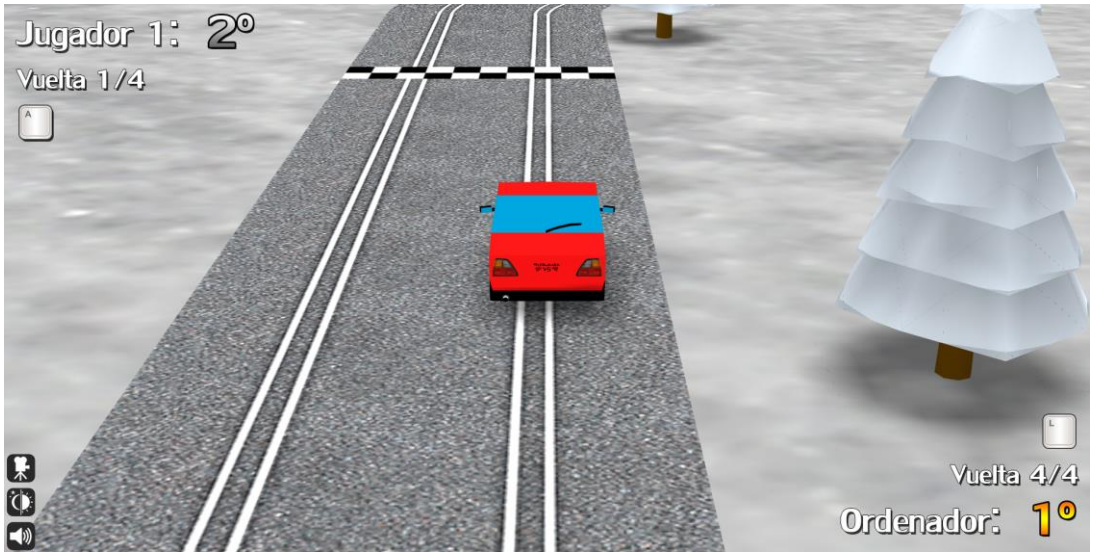


Figura 23.- Vista de la cámara cercana.

Tarea 13. Iluminación y sombreado de la escena

Para darle un toque de iluminación diferente a la escena, se ha creado otro botón para el HUD con el que poder alternar entre iluminación de día y de noche. Las dos iluminaciones cuentan con una luz ambiental de color blanco cuyo propósito es que la escena y todos sus elementos se vean con su color original:

```
// Luz ambiental
luzAmbiente = new THREE.AmbientLight(0xFFFFFF); // Color blanco
scene.add(luzAmbiente); // Se añade la luz a la escena
```

La única luz que produce sombras es la luz focal. La iluminación de día cuenta con una luz de este tipo situada en la parte superior de la habitación. Lo que se pretende imitar es la luz que produciría una lámpara situada en el techo. La configuración de esta luz es la siguiente:

```
// Luz focal que hace de lámpara en el techo (iluminación diurna)

// Color gris para no saturar (0x555555), intensidad máxima (1.0)
var luzFocalDia = new THREE.SpotLight(0x555555, 1.0);
// Posición de la luz
luzFocalDia.position.set(0, 110, 0);
// Ángulo de apertura de la luz
luzFocalDia.angle = Math.PI/8;

// Dimensiones de la pirámide que produce las sombras

// Ahora la luz produce sombras
luzFocalDia.castShadow = true;
// Distancia a partir de la cual produce sombras
luzFocalDia.shadowCameraNear = 1;
// Distancia a partir de la cual no va a producir sombras
luzFocalDia.shadowCameraFar = 120;
// Anchura de la base de la pirámide que produce sombras
luzFocalDia.shadowCameraFov = 60;

scene.add(luzFocalDia); // Se añade la luz a la escena
```

La iluminación de noche cuenta también con una luz focal propia. Sin embargo, lo que simula es la iluminación que entraría en una habitación completamente oscura a través de una ventana rectangular situada en una de sus 4 paredes:

```
// Luz focal que hace de ventana (iluminación nocturna)

// Color blanco (0xFFFFFF), intensidad máxima (1.0)
var luzFocalNoche = new THREE.SpotLight(0xFFFFFF, 1.0);
// Posición de la luz (ahora cuenta con inclinación)
luzFocalNoche.position.set(70, 70, 0);
luzFocalNoche.angle = Math.PI/8;

luzFocalNoche.castShadow = true;
luzFocalNoche.shadowCameraNear = 1;
luzFocalNoche.shadowCameraFar = 140;
luzFocalNoche.shadowCameraFov = 50;
scene.add(luzFocalNoche);
```

Una vez definidas todas las luces necesarias, llega el momento de activar la iluminación diurna o nocturna en función del valor de una variable. Cuando la variable 'numAmbiente' toma el valor 1, se activa la luz diurna. En cambio, cuando toma el valor 2, se activa la luz nocturna. El procedimiento que se sigue para activar cualquiera de las luces es el que se describe a continuación.

En primer lugar, se recorren todas y cada una de las mallas de la escena para oscurecer o intensificar el color de sus materiales. Esto es muy sencillo gracias a la función 'traverse'.

En Three.js, los objetos pueden proyectar su sombra y también recibir las de los demás objetos que la proyecten. A continuación, se comprueba si la malla que se está recorriendo se trata de una pieza del circuito o no. Para ello se observan las 3 primeras letras de su nombre (object.name.substring(0,3)). Si se trata de una recta ('rec') o una curva ('cur'), se les dice que no van a proyectar sombras (object.castShadow = false) pero sí van a recibirlas (object.receiveShadow = true). El motivo es para que no se vean sombras en el suelo de los picos de las curvas que se eliminaron con transparencia en la fase de modelado. El resto de mallas proyectan y reciben sombras, excepto las paredes y el suelo que solo las reciben:

```
// Tratamiento de la luz diurna
if (numAmbiente == 1) {

   // Se añade la luz focal correspondiente
   scene.remove(luzFocalNoche); // Se elimina la nocturna
   scene.add(luzFocalDia);      // Se añade la diurna

   // Recorrido del grafo de escena para sombras
   scene.traverse(function (object) {
      if (object instanceof THREE.Mesh) {
         // Máximo color (blanco)
         object.material.color.setHex(0xFFFFFF);

         // Si se trata de una pieza del circuito,
         // solo debe recibir sombras
         if (object.name.substring(0,3) == 'rec' ||
             object.name.substring(0,3) == 'cur') {
            object.castShadow = false;
            object.receiveShadow = true;
         } else {
            object.castShadow = true;
            object.receiveShadow = true;
         }
      }
   });
   paredes.castShadow = false; // Las paredes no proyectan sombras
   planoSuelo.castShadow = false; // El suelo no proyecta sombras
}
```

En la *Figura 24* se muestra la dirección de cada una de las luces focales implementadas en la misma escena tras pulsar el botón de iluminación en el HUD. La posición de la luz nocturna origina sombras más pronunciadas debido a su gran inclinación.

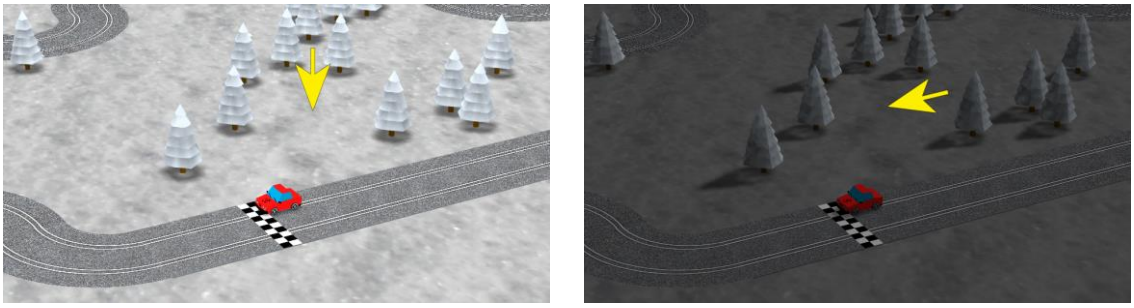


Figura 24.- Dirección de la luz focal en el modo de iluminación diurna (imagen izquierda) y nocturna (imagen derecha).

3.4. Fase de sonido

El sonido es un elemento sumamente importante en un videojuego. La última fase de este proyecto ha consistido en ambientar con diferentes melodías y efectos todas y cada una de las pantallas, desde el menú principal, pasando por el editor de circuitos, hasta los carteles de victoria y derrota que aparecen en el simulador [10].

Tarea 14. Ambientación sonora del videojuego

Para la implementación del sonido, se ha hecho uso de las propiedades de audio del DOM. Por ejemplo, para crear la melodía que suena de fondo en el menú principal, se deben crear dos elementos ('createElement'). El primero es de tipo 'audio' y el segundo de tipo 'fuente' (*source*). Al de tipo fuente hay que indicarle la ruta relativa donde se encuentra el fichero de audio a reproducir. A continuación, se añade esta fuente al audio ('appendChild') y ya se pueden utilizar el resto de métodos disponibles para manipular la melodía en cuestión.

```
var audioMenuPrincipal = document.createElement('audio');
var sourceMenuPrincipal = document.createElement('source');
sourceMenuPrincipal.src = './recursos/sounds/MenuPrincipal.mp3';
audioMenuPrincipal.appendChild(sourceMenuPrincipal);

audioMenuPrincipal.loop = true;
audioMenuPrincipal.volume = 0.8;
audioMenuPrincipal.play();
```

Con la propiedad 'loop', se fuerza al audio a repetirse cada vez que llegue a su final. Con 'volume' se puede establecer su volumen, donde 0 es el mínimo y 1 el máximo. Esta propiedad es muy útil cuando se implementa un botón para silenciar todos los audios y efectos que estén sonando. Por último, para reproducirlo se debe llamar al método 'play'. Si se quiere pausar, se llama al método 'pause', y si se quiere escuchar desde el principio se llama a 'load'.

Una vez que se han cargado todos los audios desde sus respectivas rutas relativas, es el momento de controlar cuándo deben sonar y cuándo no.

Para hacer esto, se ha creado un botón en HTML al igual que el del cambio de cámara y el del cambio de iluminación. Su posición es siempre la misma, independientemente de la pantalla en la que nos encontremos. Cuando se pulsa sobre él, se llama a la función 'cambiarAudio', cuya implementación varía en función del programa que se esté ejecutando (juego.js, editor.js o carrera.js).

Sin embargo, el funcionamiento es siempre el mismo: modificar el valor de la variable booleana 'audioOn' para que cada vez que se pulse el botón, se apague o se encienda el sonido.

```
// Función que apaga o enciende el sonido de los menús
function cambiarAudio() {
   if (audioOn) {
      audioMenuPrincipal.pause();
      audioMenuSelecciones.pause();
      audioHoverMenu.volume = 0;
      audioOn = false; // Si estaba encendido, se apaga
   } else {
      if (objects.length == 2) {
         audioMenuPrincipal.play();
      } else {
         audioMenuSelecciones.play();
      }
      audioHoverMenu.volume = 0.25;
      audioOn = true; // Si estaba apagado, se enciende
   }
}
```

Capítulo 4

Conclusiones

4.1. Objetivos alcanzados

La creación desde cero de este videojuego de carreras ha sido todo un reto personal. Los objetivos inicialmente propuestos parecían estar por encima de las posibilidades que otorgaban los tres meses de trabajo que había por delante. Sin embargo, la constancia y la dedicación depositada han sido la clave de la consecución de los objetivos que se habían propuesto inicialmente.

De entre las tareas que se han descrito en el presente documento, la construcción de la página web de presentación del videojuego no estaba contemplada en los objetivos iniciales. El motivo por el cual se ha decidido crear esta web es para darle un envoltorio al juego, como si de un papel de regalo se tratase. Con este sencillo papel se da a conocer el ámbito del juego y la tecnología con la que ha sido desarrollado, incitando al visitante a probarlo para salir de dudas acerca del funcionamiento del mismo.

Otro de los aspectos que se ha desarrollado sobre la marcha ha sido el botón del HUD que sirve para cambiar la iluminación de la escena entre diurna y nocturna. Esta función se ha creado para mostrar la capacidad de iluminación que otorga la biblioteca Three.js y para modificar el ambiente que crean los circuitos.

La última tarea que se ha llevado a cabo sin estar inicialmente prevista es la inclusión de audio en el videojuego. Como ya se ha comentado, un videojuego sin ninguna clase de sonido o incluso con una errónea selección de audios puede hacerlo mucho menos atractivo, a pesar de que su jugabilidad sea buena.

Por otro lado, la tarea de facilitar al usuario la exportación e importación de circuitos es la única que se ha quedado a medias. La fase de exportación se ha conseguido realizar con éxito, pero la importación no ha podido llevarse a cabo debido a problemas con el lenguaje JavaScript.

4.2. Trabajo futuro

El primer objetivo que se debería alcanzar es el de la posibilidad de importar cualquier circuito creado por un usuario. Gracias a esto, la cantidad de posibles circuitos sobre los que poder competir sería prácticamente infinita, dotando al juego de una mayor duración.

El siguiente objetivo sería el de aumentar la cantidad de contenido que ofrece el videojuego para que no resulte monótono a las pocas horas de jugarlo. Para empezar, se deberían diseñar como mínimo 16 circuitos prediseñados y 8 vehículos iniciales. A continuación, se establecería un modo de juego en el que se iría desbloqueando nuevo contenido a medida que se alcanzan ciertas victorias u objetivos.

Cuando se piensa en aumentar las posibilidades y el contenido de un videojuego, es difícil saber cuándo es suficiente. Otro de los modos típicos de un juego de carreras que podría implementarse es el modo contrarreloj, donde el jugador debe intentar completar cada circuito en el menor tiempo posible, compartiendo posteriormente sus mejores tiempos con el resto de jugadores mediante una clasificación online.

Por otro lado, se ha observado que el rendimiento del juego varía bastante en función del navegador web utilizado y las características técnicas de la tarjeta gráfica del ordenador donde se ejecuta. El navegador recomendado es Google Chrome, puesto que gestiona mucho mejor el estándar WebGL y la biblioteca Three.js. Para que el juego se ejecute con la misma fluidez tanto con una tarjeta gráfica de alta gama como con una de baja, se podría incluir un menú de configuración en el que poder modificar aspectos del videojuego como la calidad de las texturas y las sombras del escenario, la activación de sombras o la eliminación de los objetos decorativos para aumentar el rendimiento.

La ventaja del desarrollo de un producto es que siempre se puede seguir mejorando y puliendo para ofrecer al usuario final una experiencia enriquecedora. Estas han sido solo algunas de las muchas mejoras e ideas que se pueden desarrollar para aumentar las posibilidades de WebScalextric VS en un futuro.

Bibliografía

- Páginas web consultadas:

[1] WebGL y su vulnerabilidad (www.hipertextual.com/2011/06/webgl)

[2] Tutorial sobre la biblioteca Three.js (www.threejs.org/docs)

[3] Wikipedia, la enciclopedia libre (www.wikipedia.org)

[4] Modelado de vehículos simples (www.youtube.com/video)

[5] Web de modelos 3D gratuitos 1 (www.tf3dm.com)

[6] Web de modelos 3D gratuitos 2 (www.turbosquid.com)

[7] Object picking – Soledad Penadés (www.soledadpenades.com/threejs)

[8] Paso de parámetros entre diferentes páginas o programas (www.boutell.com/scriptpass.html)

[9] Ejemplo de cartel de carga (www.alteredqualia.com/deferred_skin.html)

[10] Sonidos gratuitos (www.freesound.org)