Escuela Técnica Superior de Ingeniería de Telecomunicación

Autor: Alejandro Fernández Arroyo

Implementación de situaciones de riesgo en un simulador de accidentes de tráfico mediante motor de videojuegos Unity

Director: Esteban Egea López

18 de septiembre de 2015

Universidad Politécnica de Cartagena

Autor	Alejandro Fernández Arroyo
E-mail del Autor	alexfa_3@hotmail.com
Director	Esteban Egea López
E-mail del Director	esteban.egea@upct.es
Codirector(es)	
Título del PFC	Implementación de situaciones de riesgo en un simulador de accidentes de tráfico mediante motor de videojuegos Unity
Descriptores	

Resumen

La memoria de este proyecto recoge el estudio realizado para desarrollar un simulador 3D de conducción en primera persona. En primer lugar se explicarán con detalle las herramientas utilizadas y cuál es su funcionamiento de forma general.

Más adelante, se expondrá la estructura que sigue el proyecto, tanto a nivel de ejecución como a nivel de clasificación en ficheros.

Lo siguiente será describir el funcionamiento íntegro de todos los elementos que forman la parte funcional del simulador.

Titulación	Ingeniería de Telecomunicaciones
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	

Contenidos

Tabla de imágenes

Introducción

Con el desarrollo de las Redes Vehiculares (VANET) se hace necesario el uso de simuladores híbridos de tráfico y comunicaciones. El diseño y desarrollo de aplicaciones para la prevención de colisiones y accidentes requiere de una simulación de accidentes de tráfico. Sin embargo, los simuladores actuales se han desarrollado para la evaluación del flujo de tráfico en condiciones de tráfico normales y utilizan modelos muy simples de conducción que no son adecuados para la simulación de accidentes.

Por el contrario, los motores de desarrollo de videojuegos son una herramienta de simulación de comportamientos físicos y modelados gráficos muy realista. Además, cada vez hay más motores que se ofrecen de manera gratuita o a bajo coste.

Por tanto, el aprovechamiento de estas ventajas para la realización de simuladores es una opción a tener muy en cuenta, ya que los resultados obtenidos son factibles y su coste asequible.

Es por ello que en este proyecto hemos decidido crear este simulador en Unity3D, una plataforma de desarrollo de videojuegos en 3D tanto para PC como para videoconsolas. Este entorno de desarrollo dispone de una gran versatilidad y flexibilidad a la hora de desarrollar cualquier aplicación o videojuego.

El simulador consta de un controlador de juegos formado por volante y pedales que otorga mayor realismo a la hora de llevar a cabo la simulación.

A continuación se describirá de forma detallada todo lo que este proyecto abarca, desde las tecnologías usadas y disponibles, hasta la descripción del simulador en su totalidad.

Resumen y objetivos

En definitiva, este proyecto comenzó a partir de la idea inicial de crear un simulador en tres dimensiones que permita extraer datos de colisiones en un accidente de tráfico en ciudad. Por tanto, esta aplicación ha tenido como objeto principal dicho cometido, cuidando además otros aspectos menos relevantes como la estética o la calidad de la experiencia del usuario.

El principal objetivo de este proyecto trata de experimentar la reacción de los conductores frente a una situación de posible accidente mediante una simulación de conducción en primera persona.

El usuario deberá conducir el vehículo por una ciudad aparentemente normal, viéndose en ocasiones obligado a tener que realizar maniobras de emergencia para evitar colisionar con otros vehículos de conducción temeraria. La aplicación generará este tipo de tráfico dependiendo de la ruta que siga el usuario por el escenario. Al pasar por ciertos puntos, un vehículo aparecerá en escena recreando un comportamiento no esperado dando lugar a una posible colisión.

Por otro lado, este simulador tiene la capacidad de almacenar los resultados obtenidos en cada simulación. Estos resultados recogerán medidas de velocidad, distancias, tiempos de frenada y detección de colisiones. Además, es posible realizar a través de *Matlab* un post-procesado de éstos, creando así una representación gráfica que describe la trayectoria y velocidad del usuario durante el experimento.

En conclusión, este proyecto abre la posibilidad a sus usuarios de experimentar y conocer su capacidad de reacción frente a posibles accidentes de tráfico, a bordo de un vehículo virtual.

1 Tecnologías utilizadas

En este primer apartado vamos a presentar y describir todas las herramientas que han permitido llevar a cabo este proyecto. Además haremos una pequeña comparación entre las utilizadas y algunas otras que podemos encontrar en el mercado.

1.1 Unity3D

Para poder entender con más claridad lo que este proyecto conlleva, es necesario hacer una pequeña introducción a esta temática que no todos conocen.

El motor de juego o motor gráfico es el elemento más importante, pues está compuesto por todas las instrucciones necesarias para la representación del videojuego o aplicación, así como por el sonido, la renderización en dos y tres dimensiones o la inteligencia artificial. En definitiva, un motor gráfico sirve de puente entre el usuario final y la aplicación, por lo que es necesario disponer de uno que cumpla las exigencias requeridas.

Unity es un motor gráfico 3D integrado creado por *Unity Technologies* con el que es posible crear multitud de aplicaciones y juegos en 3D para diferentes plataformas como PC, Android, Wii, Xbox o iOS.

Además posee un editor al que se le puede considerar como el eje central a la hora de construir una aplicación. Ofrece la posibilidad de crear un contenido de forma interactiva y visual, lo que hace que el desarrollo sea más rápido, fácil e intuitivo. Por otro lado, dichos contenidos han de tener un comportamiento que deben ser programados mediante *scripts*.

Teniendo en cuenta que un motor gráfico fluido que permita una fácil implementación es el requisito que perseguimos para la realización de este proyecto, hemos decidido hacer uso de esta herramienta, Unity3D.

Imagen 1. Interfaz de desarrollo integrado de Unity

En la imagen anterior se muestra la IDE –Integrated Development Environment- de Unity. A la derecha de la imagen se encuentra el Inspector. Una vez que seleccionemos un objeto en la escena, en esta pestaña se mostrarán todos los componentes que lo componen, así como scripts, colliders, cámaras, etc.

En la parte izquierda se encuentra el panel Hierarchy, que muestra todos los GameObjects u objetos de la escena ordenados jerárquicamente.

Por otro lado, en la parte inferior, encontramos los paneles Console y *Project* separados en dos pestañas. El panel Project se trata de un navegador que permite explorar todos los recursos que componen nuestro proyecto. Como vemos en la imagen, el directorio raíz es el llamado Assets. A partir de él heredan el resto de directorios que contienen los elementos disponibles para desarrollar nuestra aplicación.

A la hora de representar o renderizar un objeto en Unity, se realiza mediante *shaders*, que no son más que líneas de código en un lenguaje de sombreado que indican al hardware la forma de representar dicho objeto. Este lenguaje se compila de forma independiente, y se usa un lenguaje de alto nivel específico. Cabe destacar que durante la realización de este proyecto no ha sido necesario manejar este tipo de lenguaje de programación.

Por otro lado, para definir brevemente conceptos usuales en desarrollos de aplicaciones de este tipo, diremos que una textura es una imagen que sirve para cubrir la superficie de un objeto virtual, tanto en 2D como en 3D. Además, diremos que un material es básicamente un contenedor que combina texturas y *shaders* para aplicarlo a un modelo. En la siguiente imagen se ilustra el esquema descrito. [1]

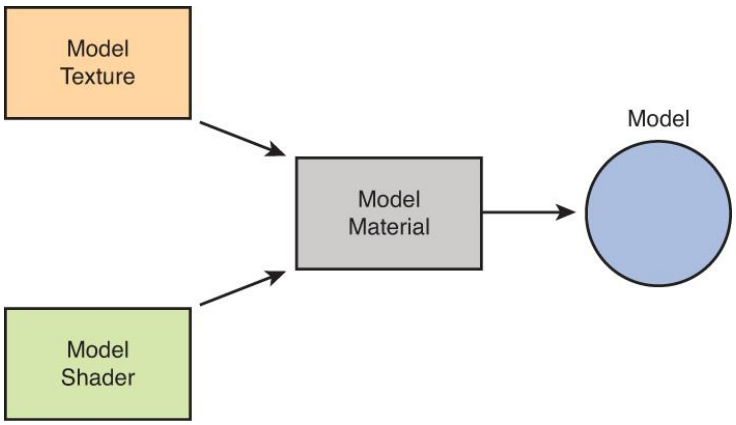

Imagen 2. Esquema de un material

Por último, el *scripting* en Unity se basa en *Mono*, la implementación de código abierto de .Net Framework. El motor que estamos introduciendo permite programar utilizando *UnityScript*, que es un lenguaje personalizado para este entorno inspirado en la sintaxis *ECMAScript*. También permite usar los lenguajes C# o Boo; éste último inspirado en *Python*.

1.1.1 FUNCIONAMIENTO GENERAL DE UNA APLICACIÓN EN UNITY3D

A la hora de crear un proyecto en Unity debemos tener en cuenta varias consideraciones previas en cuanto a estructura y jerarquía.

Un proyecto se puede componer de tantas escenas como sean necesarias, en la que cada una está formada por diversos objetos que en Unity se denominan *GameObject**s***. Cada escena será cargada independientemente y de forma secuencial según el flujo de ejecución de nuestra aplicación. También pueden contener otros *GameObjects* que estarán por debajo en la jerarquía; serán objetos hijos del primero y así sucesivamente.

Además, los GameObjects no realizan ninguna función de forma independiente, por lo que necesitan estar compuestos por elementos que realicen acciones; no son más que contenedores. Estos elementos son los denominados Components. Los Components son las piezas funcionales del proyecto, son los que se encargan del comportamiento de toda la aplicación.

Por defecto cada GameObject contiene un componente llamado Transform que es el que se encarga de la localización, rotación y escala relativa o absoluta del objeto en la escena, según el objeto sea "hijo" o "padre" respectivamente.

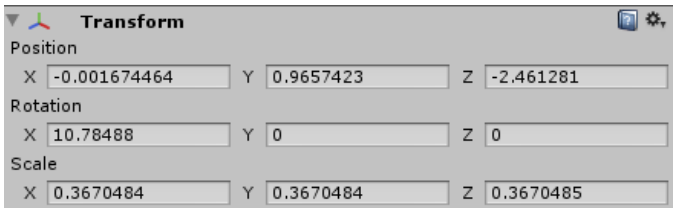

Imagen 3. Componente Transform

Por otro lado, en Unity pueden convivir scripts desarrollados en los diferentes lenguajes nombrados anteriormente siempre y cuando tengamos en cuenta el orden de compilación en el que se basa el motor.

Como regla general dentro de un proyecto se pueden crear tantos directorios como se desee, pero Unity reserva algunos nombres de carpetas para propósito general. Estos directorios tienen un efecto en el orden de compilación de los scripts.

Básicamente existen cuatro fases distintas en la compilación dependiendo del nombre de dicho directorio. Hay que tener en cuenta estas fases a la hora de hacer referencias a clases definidas en otros scripts. No se podrá acceder a todo aquello que aún no se haya compilado, es decir, a las clases cuyo directorio se encuentre en una fase posterior de compilación a la actual.

Lo mismo ocurre cuando se quiere acceder a una clase que está escrita en otro lenguaje al del script origen.

Las fases de compilación ordenadas cronológicamente son las siguientes:

- Fase 1: se compilan los scripts de los directorios Standard Assets, Pro Standard Assets y Plugins.

- Fase 2: los scripts de los directorios Standard Assets/Editor, Pro Standard Assets/Editor y Plugins/Editor.
- Fase 3: todos los demás scripts que no estén dentro del directorio Editor.
- Fase 4: el resto de scripts, por ejemplo, los que están dentro del directorio Editor.

Por otro lado, en cuanto a la sintaxis de UnityScript también existe una jerarquía que resuelve define el orden en el que se ejecutan las funciones de carácter específico dentro de un script o escena.

Cada vez que se cargue una escena, se ejecutarán todas las funciones llamadas Awake que se hayan implementado en dicha escena, siempre cumpliendo al mismo tiempo la jerarquía definida por los directorios descritos con anterioridad. La función Awake se ejecutará incluso antes de que los objetos sean instanciados en el entorno virtual.

La siguiente función que el motor se encarga de buscar y ejecutar en la escena es la denominada OnEnable. Esta función sólo se llamará en caso de que el objeto esté activo, y se hará justo antes de que el objeto se habilite.

La función Start será ejecutada inmediatamente después, sí y sólo sí, el objeto está activo. Dentro de esta función se definirán la mayor parte de variables necesarias para el script.

Tras la ejecución de estos tipos de funciones, Unity se encargará de iniciar la simulación de la aplicación o videojuego, por lo que se generará de forma visible el primer frame en la pantalla.

Después de estas funciones están las funciones llamadas OnApplicationPause que será ejecutada al final del frame en el que se detectó el "pause" de la aplicación. Como indica el nombre, sirve para realizar acciones a la hora de pausar el juego.

Más tarde se llamará a las funciones FixedUpdate. Si el frame ratio es bajo puede varias veces por frame, sino, puede incluso no ser llamada entre frame y frame. En esta función se debe colocar el código que se encarga de las físicas del juego.

Después, tras FixedUpdate, se ejecutarán todas las funciones llamadas Update. Ésta sí que se llamará una vez por frame. Dentro de esta función se podrán realizar todas las operaciones que precisen de actualización de variables, bucles, etc.

Por último, se procederá a la ejecución de las funciones *LateUpdate*. También se llamará una vez por frame, pero siempre después de *Update* por si nos interesa trabajar con algún dato tras ser actualizado, por ejemplo.

Existen además de las comentadas, otras muchas funciones ordenadas temporalmente después de *FixedUpdate* que afectan al renderizado de texturas, como puede ser *OnGUI*.

OnGUI ha sido utilizada en nuestro proyecto a la hora de realizar la interfaz gráfica de usuario de la que dispone. Es una función que se encarga de renderizar las texturas de la interfaz y capaz de procesar, varias veces por frame, eventos que provienen del ratón, teclado o cualquier otro método de entrada.

Como se observa, Unity dispone de una amplia cantidad de funciones que permiten interactuar con los objetos y sus propiedades, lo que permite tener un gran abanico de posibilidades a la hora de trabajar. [2]

1.1.2 VENTAJAS DE UNITY3D

Una de las principales ventajas de Unity es la capacidad de transportar una aplicación o juego a otras plataformas como Android, iOS o PS3 sin necesidad de programar de nuevo. Cabe destacar que para realizar algunas de estas conversiones hay que pagar una licencia.

Otra gran ventaja - unas de las principales - es la facilidad para crear una aplicación. Este *IDE* permite crear ambientes y entornos de forma visual y totalmente intuitiva. Permite realizar cambios de la simulación en tiempo real, es decir, es posible modificar el valor de variables, instanciar objetos o modificar partes del código de los scripts para observar su funcionamiento de forma instantánea sin necesidad de parar la simulación. Gracias a esto se puede conseguir el comportamiento deseado de nuestro juego en tiempo de ejecución, lo que hace que el desarrollo de la aplicación de principio a fin sea más rápido. Además, la programación mediante scripts es un método muy potente que ofrece innumerables posibilidades. [3]

La posibilidad de trabajar con distintos lenguajes de programación hace que Unity sea aún más atractivo para los programadores no tan experimentados.

Por otra parte, Unity dispone de una muy buena documentación en su web, donde se puede acudir en cualquier momento y en la que están detalladas todas y cada una de las funciones y variables personalizadas de su *IDE*.

Por otro lado, Unity es compatible con una gran cantidad de formatos de imágenes, modelos 3D, texturas o fuentes tipográficas. Se pueden importar una altísima variedad de modelos 3D desde aplicaciones de modelado como Blender, Cinema4D, Maya o 3DS-Max.

Unity3D dispone de una tienda oficial de recursos llamada *Asset-Store* en la que se encuentran gran cantidad de componentes como librerías, texturas o sonidos que se pueden importar directamente desde la interfaz de desarrollo.

En conclusión, Unity es un buen entorno de desarrollo integrado para la programación multiplataforma y para programadores no tan expertos. Para proyectos más complejos, está disponible la versión Unity3D Pro cuya licencia cuesta unos 1500 $ al ofrecer mejores prestaciones.

1.2 Desarrollador Monodevelop

Dentro del paquete de Unity podemos encontrar la herramienta Monodevelop, un entorno de desarrollo de los lenguajes de programación que soporta Unity, con una completa guía en la que se pueden encontrar todas las funciones, variables o propiedades que ofrece el motor. En esta guía se puede ver la implementación de todas las funciones para conocer internamente su funcionamiento, así como los parámetros de entrada y salida de cada una de ellas. En la imagen podemos ver un ejemplo de ello, mostrándonos la implementación de la función *DrawTexture* y los parámetros de entrada.

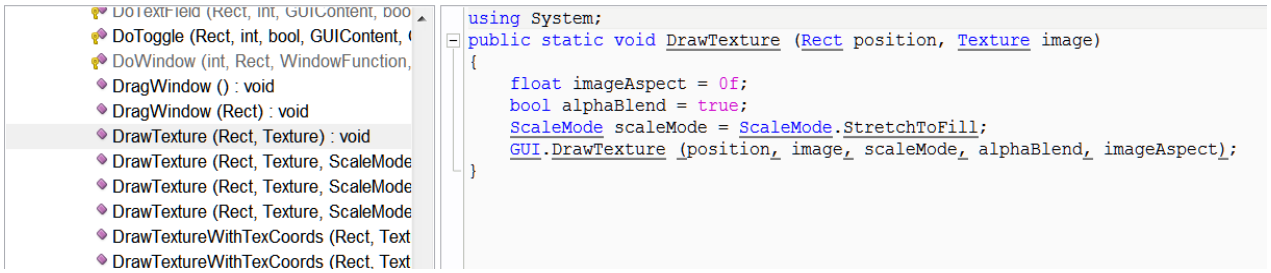

Imagen 4. Implementación de la función DrawTexture

Esto es posible hacerlo para cualquiera de los lenguajes aceptados por Unity. También se pueden ver las clases disponibles, con sus correspondientes variables y métodos. En la siguiente imagen podemos ver la clase *WheelCollider* y las variables y métodos que la componen.

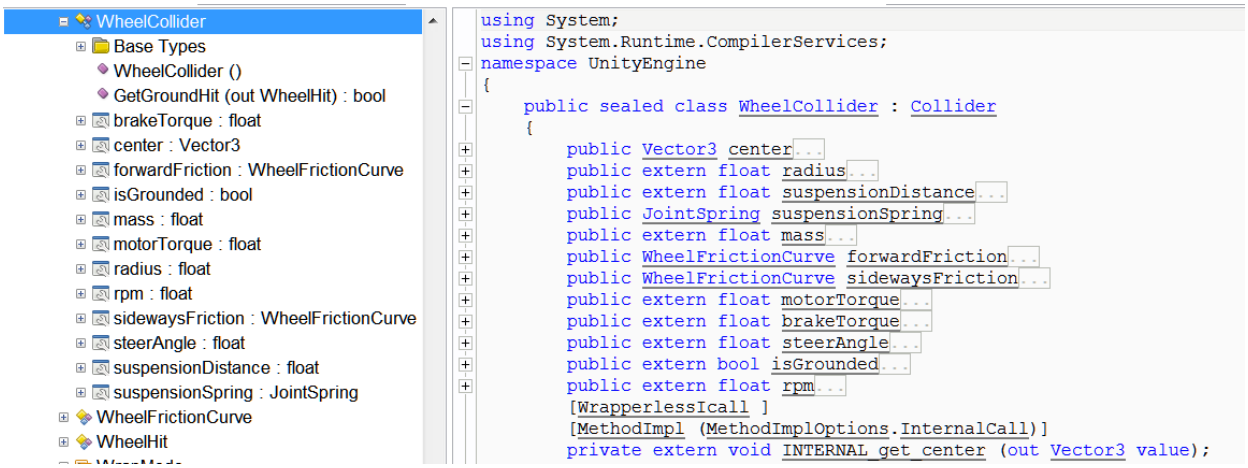

Imagen 5. Implementación de la clase WheelCollider

1.3 Herramientas alternativas

Dentro de las herramientas alternativas a la hora de desarrollar un simulador de tráfico en tres dimensiones podemos distinguir dos tipos: motores gráficos 3D similares a Unity, y simuladores de conducción cuyo propósito es exclusivamente la recreación de escenarios de conducción.

1.3.1 MOTORES GRÁFICOS DE APLICACIONES EN 3D

Para la creación de aplicaciones 3D existen muchos otros motores gráficos como Shiva 3D, o Torque 3D entre otros. Éstas son dos de las posibles alternativas para todo aquel programador sin mucha experiencia en el sector que desee crear juegos o aplicaciones 3D. A continuación describiremos brevemente algunas de estos motores gráficos.

1.3.1.1 SHIVA 3D

Shiva 3D se caracteriza por ser un motor que utiliza *LUA Script*, un tipo de *script* que tiene mejor rendimiento para juegos que requieran mayor calidad gráfica. Es un motor más rápido que el de Unity y además consume menos memoria. Por contrapartida, es un *IDE* más compleja y menos atractiva e interactiva que Unity, por lo que la curva de aprendizaje es más lenta con Shiva 3D. La licencia ilimitada de Shiva ronda los 1000$. [4]

Imagen 6. Editor del IDE de Shiva 3D

1.3.1.2 TORQUE 3D

Si hacemos la comparación con Torque 3D, diremos que es de código abierto y éste tiene mayores capacidades a la hora de renderizar que Unity, pues incluye mejores efectos de post procesado. Tiene herramientas más potentes como el editor de terrenos en tiempo real, editor de ríos, caminos y carreteras, etc. Torque utiliza el lenguaje *TorqueScript*, basado en C++. [5]

Torque se clasifica como un mejor motor gráfico en cuanto a calidad de resultados, pero muchos usuarios siguen prefiriendo Unity debido a lo ya comentado anteriormente: obtiene muy buenos resultados con un nivel de aprendizaje menos exigente.

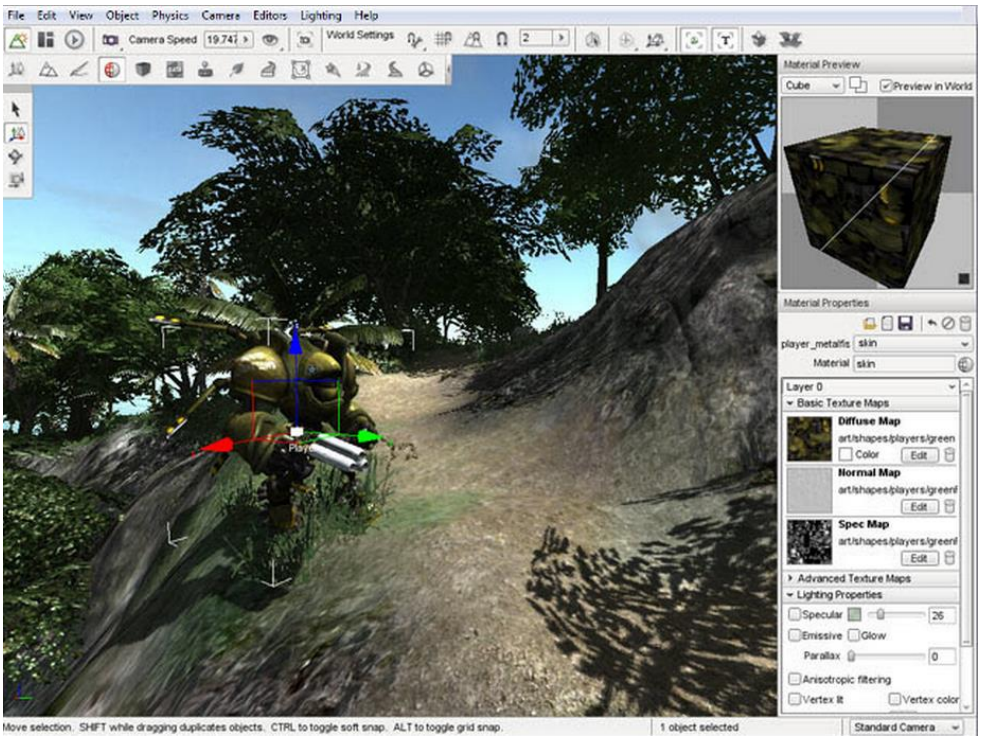

Imagen 7. Editor del IDE de Torque 3D

1.3.2 SIMULADORES DE CONDUCCIÓN

Una vez hemos comparado Unity con otros motores gráficos para desarrollar videojuegos en 3D podemos hacer una comparación con otros simuladores de vehículos, puesto que era otra opción a tener en cuenta al realizar este proyecto. Algunos de estos son: Rfactor, Car-X Technologies, VDrift o OpenDs.

1.3.2.1 RFACTOR

Rfactor es un complejo simulador de conducción online y *offline* que requiere de licencia para su utilización. Dispone de un motor de físicas avanzado con el que se pueden modificar aspectos de aerodinámica, neumáticos, etc. Dispone de una experiencia de transiciones día/noche a tiempo real, conducciones en diferentes climatologías y conducción nocturna.

También cabe la posibilidad de realizar cambios en las físicas del vehículo mediante un *SDK*. Sin embargo no dispone de la flexibilidad suficiente para realizar nuestro proyecto. [6]

1.3.2.2 CAR-X

Car-X es un motor de físicas para coches usado para aplicaciones y simuladores con sistemas de realidad virtual. Car-X funciona como un módulo externo que añade comportamientos reales al vehículo de tu sistema. Este motor se puede integrar con Unity3D, es decir, se pueden importar desde Unity los vehículos con las físicas desarrolladas por Car-X. Este sistema está orientado sobre todo a simulaciones de tráfico en ciudades, donde existen intersecciones, semáforos, etc. Está disponible para varias plataformas como PC, Mac, PS3 o Wii y sólo se puede disfrutar la versión demo de forma gratuita. Usa codificación C++ para PCs y Mac, y C# para el resto de plataformas. [7]

Imagen 8. Sistema Car-X integrado en Unity3D

1.3.2.3 VDRIFT

VDrift es una plataforma de simulación de conducción de código abierto basado principalmente en carreras de drift. El motor de físicas es nuevo pero está inspirado en el motor físico “Vamos”. Se distribuye con la licencia GNU 2 y 3 para Linux, Windows, Mac y FreeBSD. Dispone de muchas opciones entre ellas la de *Force Feedback*. [8]

Imagen 9. Simulación en el entorno VDrift

1.3.2.4 OPENDS

OpenDS es un software de simulación de conducción de código abierto. Está orientado a fines de investigación y es completamente desarrollado en Java, basado en el *framework* de *jMonkey Engine.* Está enfocado para todas las plataformas *OpenGL* 2 y 3 que dispongas de máquina virtual de Java.

Dispone de características interesantes como el cálculo de consumo del vehículo, la posibilidad de incluir tráfico autónomo o la simulación de diferentes meteorologías. [9]

Imagen 10. Simulador OpenDS.

2 Descripción de la aplicación

En primer lugar nos vamos a centrar en el sistema de archivos que se ha llevado a cabo en la aplicación, describiendo los componentes que lo forman y los directorios en los que se dividen.

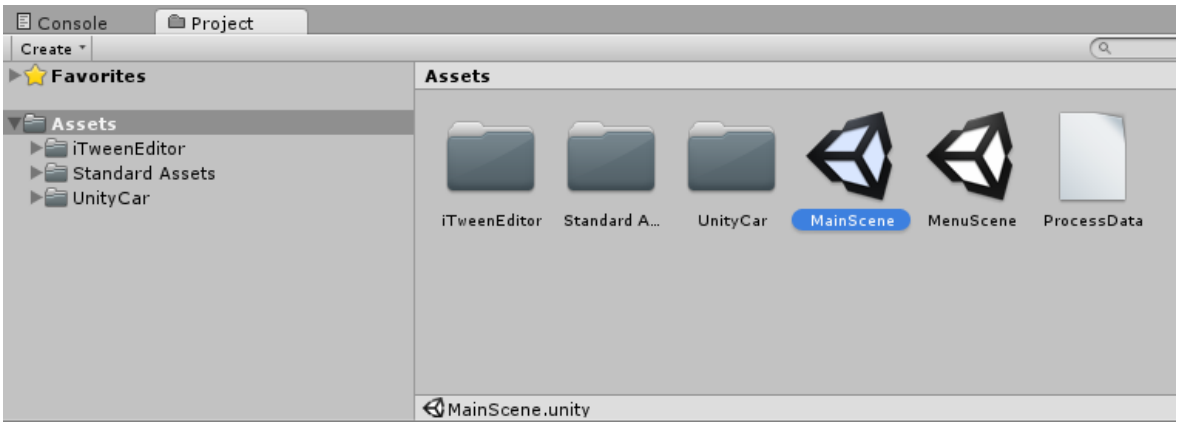

Imagen 11. Escenas en el explorador de Unity

En la anterior imagen se pueden apreciar en el sistema de ficheros las dos escenas que dan forma a este proyecto. En concreto, la escena del menú inicial es llamada MenuScene y la escena principal llevará el nombre de “MainScene”. Como se puede observar en la barra de información del panel de navegación estos archivos tienen extensión “.unity”.

Para describir con mayor claridad la aplicación desarrollada lo haremos detallando los componentes más relevantes que afectan a su correcto funcionamiento. Para ello dividiremos este apartado en dos partes claramente diferenciadas que corresponden a las dos escenas que componen el simulador: menú y escena principal.

2.1 Escena de menú

La escena de menú constituye la escena inicial, es decir, la escena que se carga inicialmente al ejecutar el simulador. Esta escena tiene como finalidad principal

separar temporalmente el inicio de la simulación del inicio de la ejecución de la aplicación.

2.1.1 CONFIGURACIÓN

Esta escena inicial, en cuanto a configuración, nos permite editar tanto la ruta donde exportar los datos tras la simulación, como el nombre del fichero que la contiene.

Por otro lado, esta escena inicial permite, a través de un botón, iniciar la simulación tras haber configurado los parámetros anteriormente nombrados. Cabe destacar que en caso de no editar los campos de configuración no provocará ningún problema a la hora de la ejecución de la aplicación. Esto es debido a que ambos campos contienen una configuración preestablecida. En concreto, la ruta donde se almacenarán los ficheros por defecto será C:\Users\usuario\Documents\, ruta que disponen todos los sistemas con sistema operativo Windows. Además el nombre predefinido del archivo de datos exportados es CrashInfo.txt. En la siguiente imagen se puede observar todo lo descrito anteriormente sobre el menú de inicio.

Imagen 12. Menú de inicio

2.1.2 ELEMENTOS DECORATIVOS

En cuanto al aspecto decorativo de esta escena, se han incluido dos componentes principales que otorgan un aspecto más dinámico al menú. En primer lugar, un vídeo demostrativo de la escena mediante una cámara; y, en segundo lugar, y al mismo tiempo que se genera dicho vídeo, una serie de vehículos dotados de inteligencia artificial circulan por la escena.

Es decir, por un lado se ha instanciado una cámara con unos comportamientos añadidos que dan como resultado un vídeo demostrativo de la ciudad que compone esta escena. Dicho vídeo se reproduce en segundo plano durante la ejecución del menú.

La cámara que se encarga de hacer el recorrido por la ciudad tiene añadido tres scripts. Uno de ellos contiene el código que permite representar en pantalla la interfaz gráfica del menú. Los otros dos scripts se encargan de darle el movimiento y la rotación correcta. Este comportamiento ha sido posible reproducirlo mediante una herramienta llamada “iTween”. En la siguiente imagen se representan los *scripts* que tiene la cámara añadidos, anteriormente comentados. Podemos ver dos componentes *iTweenPath* y otros dos *iTweenEvent* que trabajan conjuntamente.

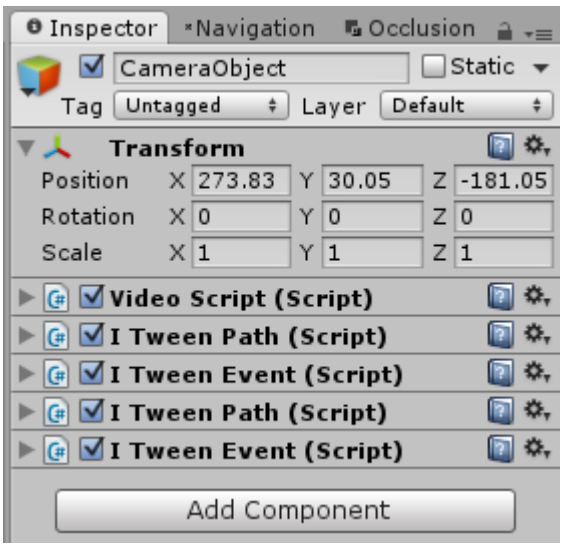

Imagen 13. Componentes de la cámara

Esta herramienta aporta una serie de funciones que nos han permitido crear rutas en tres dimensiones a lo largo de la escena. Además, se incluyen también múltiples

funciones que consiguen aportar dicho movimiento y rotación a objetos, en este caso, a la cámara.

Para lograr ese dinamismo a la cámara principal, ha sido necesario añadir dos *scripts* de dicha herramienta: *iTweenPath* y *iTweenEvent*.

La primera de ellas permite crear dicha ruta en el espacio tridimensional de Unity. Para ello, se añaden dos puntos o más en el lugar más conveniente para lograr la ruta deseada. *iTweenPath* dibuja en la escena dicho recorrido interpolando dichos puntos mediante sus posiciones y distancias que los separan.

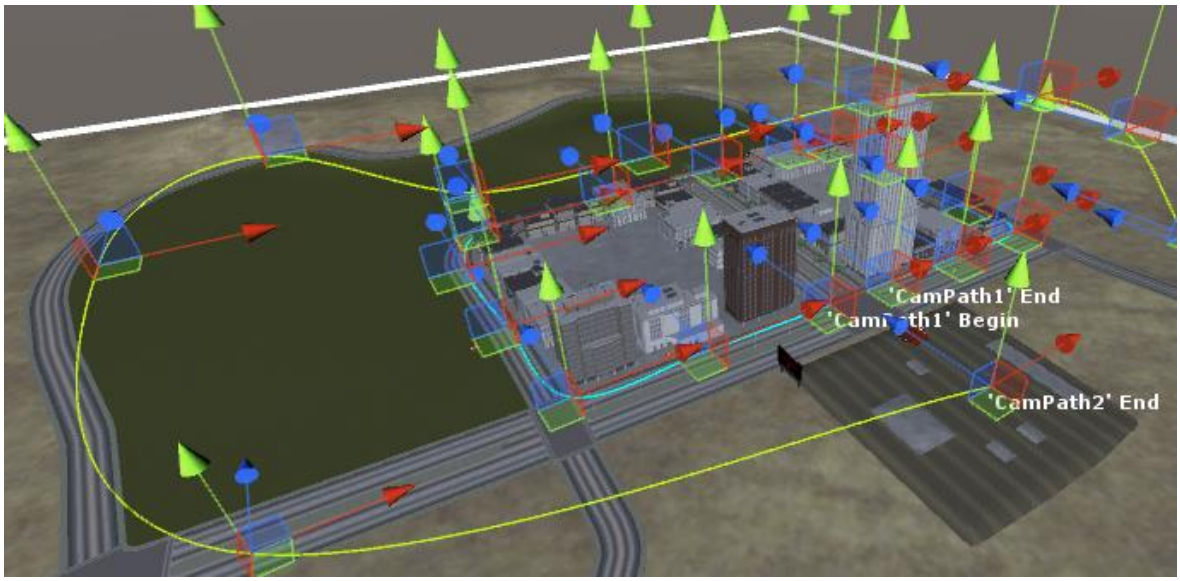

Imagen 14. Rutas iTween de la cámara

Se podrán crear tantas rutas como se desee para cada objeto. En concreto, nuestra cámara dispone de dos rutas bien diferenciadas que se ejecutan de forma secuencial tras iniciar la aplicación.

La primera de ellas –la ruta azul en la imagen- realiza un recorrido a ras del suelo a través de las carreteras de esta escena. De esta forma el usuario puede conocer un poco mejor la ciudad antes de comenzar la simulación.

La segunda ruta –la amarilla en la imagen- realiza un trayecto aéreo alrededor de la escena para poder tener una perspectiva más amplia de nuestro entorno de simulación.

2.2 Escena principal

Como ya hemos nombrado con anterioridad, esta es la escena con más peso en el desarrollo del proyecto. Esto está justificado ya que la mayor parte del tiempo de elaboración se ha centrado en ella, puesto que es la parte funcional que persigue nuestro objetivo.

En esta escena vamos a diferenciar cuatro elementos principales cuyos comportamientos añadidos dan lugar al núcleo de nuestra aplicación. Estos son: el vehículo principal, los contenedores de *waypoints*, los muros de detección y los vehículos con inteligencia artificial. Además se detallará finalmente la configuración del controlador de juegos que permitirá al usuario manejar el simulador.

A continuación se describen minuciosamente los anteriores elementos y posteriormente, en el Anexo A, se detallarán rigurosamente los scripts utilizados en los mismos.

2.2.1 VEHÍCULO PRINCIPAL

En este simulador la parte que puede ser manejada por el usuario en primera persona es un vehículo, al que llamaremos vehículo principal.

En primer lugar, este vehículo hace uso de una herramienta llamada “UnityCar Pro” que dota al automóvil de motricidad, es decir, añade las físicas para que éste circule. Para aclarar esto, vamos a explicar brevemente en que consiste UnityCar Pro.

UnityCar Pro es una herramienta no gratuita que podemos encontrar en el *Asset Store* de Unity. Se compone de una serie de directorios como vemos en la siguiente imagen:

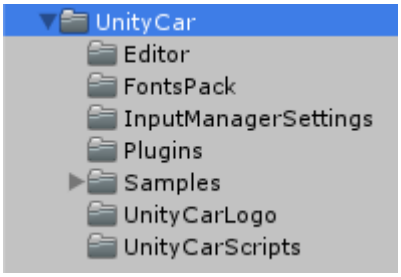

Imagen 15. Composición de UnityCar Pro

Entre otros, podemos destacar los directorios *Samples* y *UnityCarScript* que contienen los componentes imprescindibles para su utilización.

En primer lugar, *Samples* contiene todos los componentes referentes a los objetos que intervienen, como los *Prefabs* de los vehículos y escenas, los modelos en 3D, texturas, *shaders*, materiales, sonidos y archivos de configuración de los vehículos. En este directorio también podemos encontrar unas escenas prediseñadas entre las que se encuentra la ciudad sobre la que se ha desarrollado este simulador. Además encontramos una escena de prueba o test en la que podemos ejecutar esta herramienta sin necesidad de configurar ni implementar nada previamente.

Por otro lado, en el directorio *UnityCarScript* se encuentran todos los scripts que dotan a la herramienta de funcionalidad. Para hacernos una idea de la complejidad de esta herramienta basta con ver la gran cantidad de *scripts* que la componen.

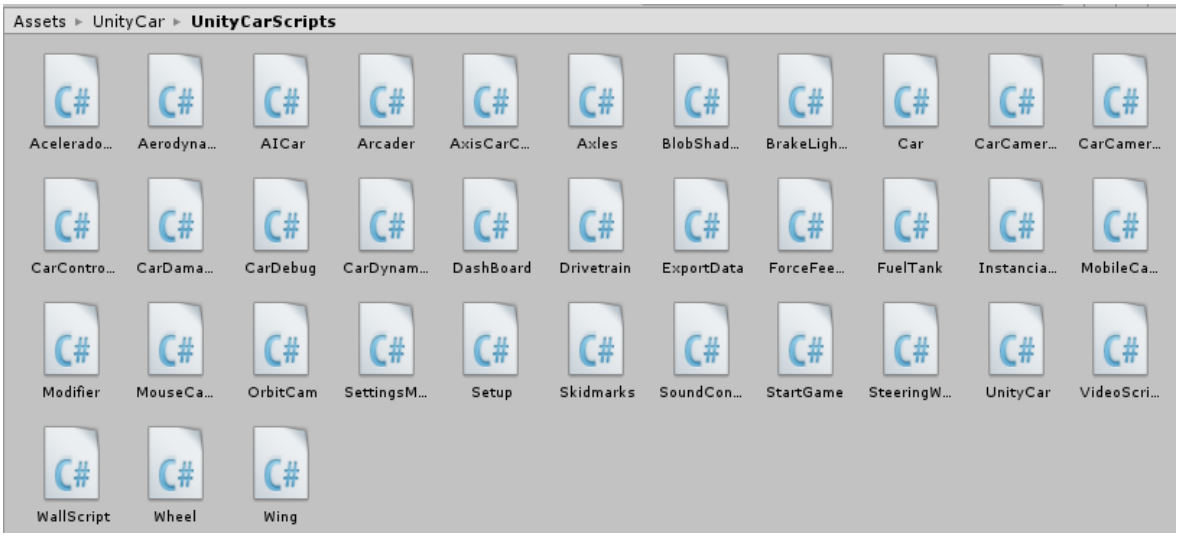

Imagen 16. Scripts de UnityCar Pro

UnityCar Pro es capaz de generar y aplicar todas las fuerzas que intervienen en un vehículo real a la hora de circular, desde la potencia del motor teniendo en cuenta las marchas y revoluciones de éste, hasta la inercia o el rozamiento del vehículo. Se trata de un complemento indispensable a la hora de la realización de este proyecto puesto que permite recrear a la perfección las físicas que intervienen en el proceso. Estos scripts se tratan con más detalle en el Anexo A.

Por otra parte, las clases instanciables de esta herramienta que deben ser añadidas al vehículo principal son las que se observan en la siguiente imagen, a excepción de la clase *ExportData*.

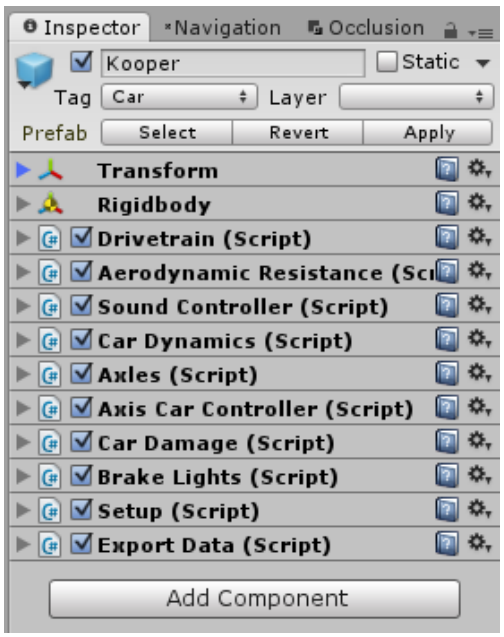

Imagen 17. Componentes del vehículo principal

Para finalizar la descripción del vehículo principal de la escena es necesario explicar ese último elemento que lo compone, llamado *ExportData*.

La clase *ExportData* ha sido implementada para llevar a cabo dos funciones bien diferenciadas. La primera de ellas consiste en obtener y volcar los datos de la simulación en dos archivos con formato ".txt". Es capaz de obtener medidas de velocidad, tiempo, posición y tiempos de frenada y exportarlas en tiempo de ejecución.

La segunda función que realiza este *script* es la de controlar el menú de pausa del simulador. Si en mitad de una simulación es usuario presiona la tecla *Escape*, aparecerá el siguiente menú de pausa.

Imagen 18. Menú de pausa

Como podemos ver, el menú ofrece la posibilidad de reanudar la simulación, reiniciarla, volver al menú inicial o salir de la aplicación. Se trata de un elemento indispensable en toda aplicación interactiva.

2.2.2 CONTENEDORES DE WAYPOINTS

Para comprender el funcionamiento de los vehículos de inteligencia artificial es primordial conocer con qué criterio se desplazan. Para ello, los vehículos se encargan de seguir una ruta establecida previamente. Dicha ruta será el resultado de la interpolación no lineal que resulta al unir una serie de puntos colocados en el espacio virtual. Estos puntos, a los que llamaremos *waypoints* o puntos de referencia, están colocados de forma intencionada en el espacio tridimensional para generar la ruta deseada.

En nuestra escena existen dos contenedores de *waypoints* asociados a cada vehículo autónomo, llamados *WPContainer* y *WPInterp*.

El primero de ellos contiene los *waypoints* creados manualmente y ordenados de forma secuencial para dar lugar a la ruta. En la siguiente imagen podemos ver los *waypoints* de una de las rutas representados en color rojo.

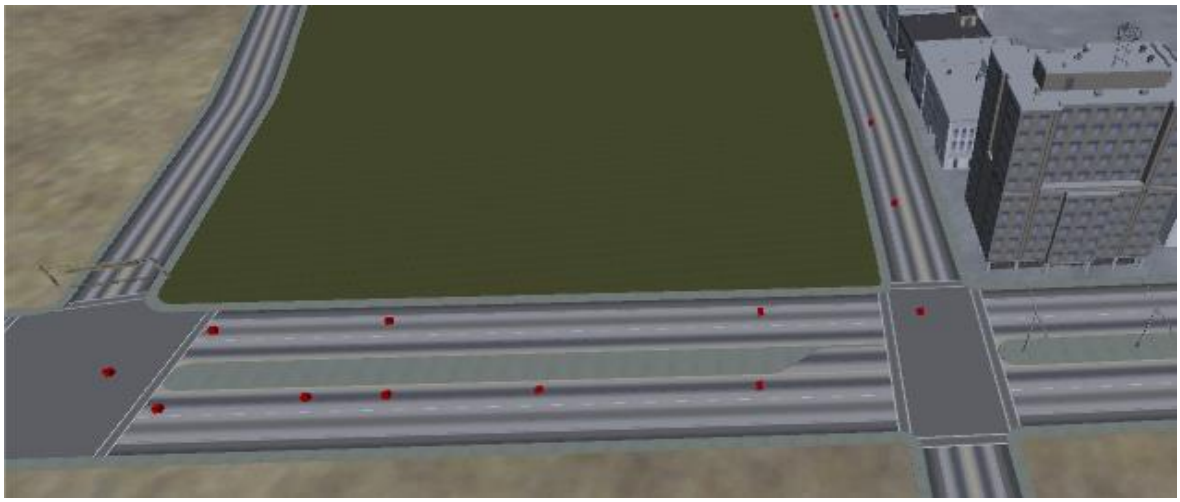

Imagen 19. WPContainer

Si unimos los *waypoints* obtendremos el tramo de una de las rutas que se han creado en este simulador. Por contrapartida, esta unión nos dará como resultado una trazada con poca resolución, es decir, los *waypoints* están demasiado separados entre sí, lo que provoca que la ruta no sea la más idónea para que los vehículos circulen.

Para solucionar ese problema se ha creado el contenedor *WPInterp* nombrado con anterioridad. Este contenedor hace uso de la clase *InterpWP*, cuya función es realizar una interpolación no lineal entre dos *waypoints* adyacentes añadiendo seis nuevos puntos de referencia entre ellos.

Inicialmente este contenedor estará vacío pero, al comienzo de la simulación, dicho script recogerá los *waypoints* de *WPContainer* para después almacenar los nuevos *waypoints* tras la interpolación. Con los puntos de referencia ya interpolados se procede a generar la ruta, ahora más precisa, que seguirán los vehículos autónomos.

En la siguiente imagen se puede observar claramente los dos tipos de *waypoints* descritos. De color rojo y forma cúbica se representan los *waypoints* iniciales generados de forma manual, y de color gris y forma esférica los que resultan tras

la operación de interpolación. Además, se dibuja de color magenta la ruta resultante y definitiva que seguirá el vehículo.

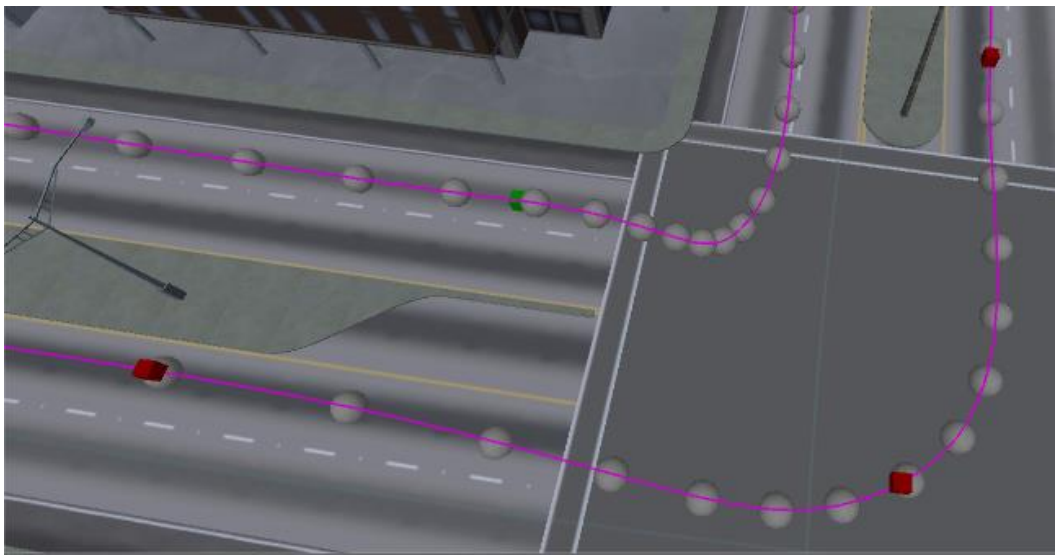

Imagen 20. Waypoints interpolados

Por otro lado, *WPContainer* lleva asociada la clase *InstantiateAICars*. Esta clase instanciará en tiempo de ejecución los vehículos de *IA* para cada una de las rutas, además de configurar los parámetros necesarios para su funcionamiento.

Llegado a este punto, mencionamos que existen en este proyecto dos tipos de rutas diferentes: las rutas cerradas y las no cerradas.

Las primeras se tratan de rutas cuya trayectoria es cerrada, es decir, el primer *waypoint* de ésta coincide con el último; por lo tanto cuando el vehículo ha concluido una vuelta sobre el trazado volverá a realizar otro, y así indefinidamente. El objetivo de este tipo de ruta es aportar al sistema un carácter de continuidad y periodicidad en el tiempo. Sobre cada una de éstas se instanciarán cuatro automóviles autónomos equiespaciados temporalmente. En definitiva, en todo momento habrá vehículos circulando por la escena.

Por otra parte se han definido algunas rutas no cerradas o abiertas cuya trayectoria es limitada, al igual que su duración. Éstas se activarán únicamente cuando el usuario pase por unos puntos determinados de la escena, que llamaremos muros de detección, y sólo se instanciará un vehículo por ruta. La finalidad de estas rutas

es generar una situación de riesgo de accidente sobre el usuario, que deberá reaccionar a tiempo para tratar de evitar la colisión.

Por último, cabe destacar que las dos escenas que componen el proyecto hacen uso de estos dos contenedores de *waypoints* dando lugar a diferentes rutas independientes, pero solo la escena principal generará rutas abiertas.

2.2.3 VEHÍCULOS CON IA

Como ya sabemos llegado a este punto, existen en este proyecto dos tipos de vehículos en escena: vehículos *IA*, y el vehículo principal. En este apartado describimos los vehículos del primer tipo.

Los automóviles dotados de inteligencia comparten con el vehículo principal todos los scripts de UnityCar que le aportan la función de movilidad. Sin embargo, a diferencia de éste, los vehículos IA están compuestos por una clase llamada *AICar*. Esta clase ha sido la más compleja y la que más tiempo ha llevado en su implementación puesto que es la que aporta esa inteligencia artificial.

A grandes rasgos, la inteligencia artificial está formada por dos funcionalidades principales: el seguimiento de *waypoints* y el control de frenada y colisiones.

Para describir cómo funciona el seguimiento de *waypoints* lo haremos siguiendo el orden de ejecución de las instrucciones del código que lo compone.

En primer lugar, tras instanciar un vehículo en la escena se procede a reconocer el contenedor que ha solicitado esta acción. Para ello, diferenciamos el número de la ruta que ha de seguir el vehículo, ya que las rutas poseen una numeración que las distingue.

Una vez conocemos la trazada que ha de seguir se obtienen los *waypoints* que componen dicha ruta y se procede a la interpolación de éstos para dar lugar a una mejor construida.

Más tarde se extraen de dicha trayectoria una serie de puntos y son almacenados en un *array* de *Transforms*. Estos puntos indicarán al vehículo la posición en la que deberán reducir la velocidad para trazar una curva y evitar salir de la carretera. Una vez hecho esto, el vehículo está listo para circular.

La segunda función que realiza la clase *AICar*, y no menos importante, es el control de frenadas y la evasión de colisiones.

Para realizar el control de frenada se hará uso del *array* de puntos que se explicó anteriormente. El vehículo detectará el paso por cada uno de esos *waypoints* y aminorará la velocidad a la mitad de forma moderada.

Todo esto permitirá a un vehículo realizar la trayectoria de forma controlada e ininterrumpida dando lugar a un sistema de duración infinita.

En cuanto al sistema de evasión de colisión se ha realizado mediante dos tipos de objetos diferentes. Una implementación está basada en la generación de *Raycasts*, mientras la otra hace uso de un *Collider*.

Un *Raycast* es un elemento que proporciona el motor de Unity para detectar colisiones y, a partir de ahí, extraer algunos datos de éstas. Si creamos un *Raycast* dentro de un objeto, en este caso en los vehículos *IA*, se van a lanzar de forma periódica desde ese punto una serie de “rayos”. Estos rayos son líneas rectas no visibles con una dirección y una longitud determinadas. Por lo tanto, con dicho rayo podemos detectar una colisión con otro objeto y, a su vez, podemos filtrar el tipo de objeto a detectar.

Haciendo uso de este elemento, se ha dotado a cada vehículo con un conjunto de cinco *Raycasts* frontales.

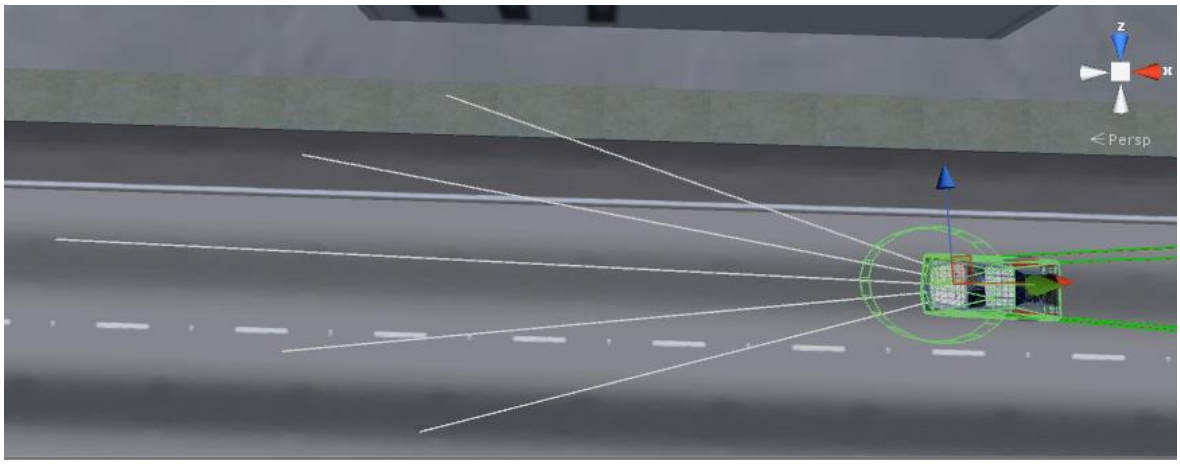

Imagen 21. Raycasts del vehículo IA

Los cinco *Raycasts* quedan representados en la imagen anterior como líneas blancas que parten del frontal de automóvil. Como se puede ver, están distribuidos de forma que el rayo más largo viaje en la misma dirección que el vehículo mientras que las otras dos parejas de rayos tienen menor longitud.

La longitud de los rayos es dependiente de la velocidad del coche al tratar de recrear la realidad con la mayor exactitud posible.Como es lógico, a menor velocidad, menor capacidad de reacción es necesaria para detener el vehículo en caso de frenada brusca y viceversa. Gracias a esta dependencia con la velocidad del vehículo, aumentará la longitud de los rayos, aportando así esa capacidad para prevenir colisiones entre vehículos, que comenzarán a frenar con mayor antelación.

Para su correcto funcionamiento hay que generar un rayo en cada *frame* de la simulación y poder detectar con más precisión las colisiones. Aun así, ésta ha sido la tarea más problemática del proyecto debido a que los rayos no reportan de forma continua el estado de las colisiones.

Además de los Raycasts, encontramos otro elemento complementario que favorece la evasión de colisiones, el *Collider*. Los *Colliders*, o “colisionadores”, son básicamente componentes que permiten la colisión entre los objetos de la escena. Se pueden, manejar con diferentes configuraciones pero, en este caso, los *Colliders* son meramente detectores de colisión.

Como se ve en la Imagen 21, el vehículo posee en la parte frontal un objeto circular pintado en verde, éste es el *Collider* de dicho automóvil. Con esta configuración, el “collisionador” funcionará de forma similar al *Raycast*: al ser atravesado por otro coche se realizará una acción en concreto. En este caso, si el *Collider* entra en contacto con otro vehículo, el vehículo detector se encuentra demasiado cerca del que le precede por lo que será necesario detener la marcha. Una vez que el coche de delante reanude la marcha y deje de estar en contacto con el *Collider*, el vehículo posterior reanudará la marcha también.

Por tanto, si antes decíamos que se trataba de un sistema infinito con un solo coche gracias a los contenedores y sus puntos de frenada, ahora diremos que el sistema es infinito con un número de vehículos indefinido, pues posee la capacidad de prevenir colisiones entre los mismos.

2.2.4 MUROS DE DETECCIÓN

Como ya se comentó en el apartado “Contenedores de waypoints”, una de las funcionalidades de esta aplicación es la creación de situaciones con riesgo de accidente. Para ello utilizamos las rutas abiertas que generan tráfico cuando el usuario pasa por un determinado punto de la escena.

Esos puntos colocados estratégicamente son los llamados muros de detección. Al ser atravesados por el vehículo que conduce el usuario activan una de las rutas abiertas, lo que provoca que se instancie un nuevo vehículo en la escena.

Este nuevo vehículo es el encargado de que el usuario tenga que reaccionar para evitar una colisión con el mismo, que realiza una maniobra anómala en su recorrido.

De la parte funcional de estos muros de detección se encarga una clase denominada *WallScript* asociada a dichos elementos.

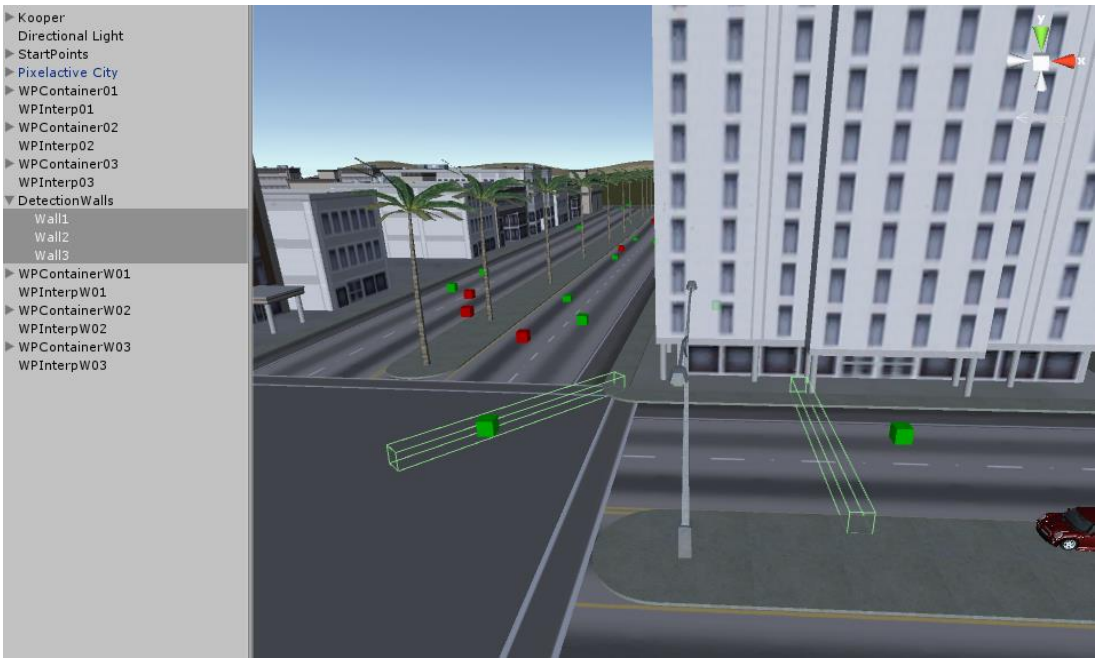

Imagen 22. Muros de detección

En la imagen anterior se pueden observar dos de los tres muros que existen en la escena. En la parte izquierda, los tres muros quedan ubicados dentro del *GameObject* padre llamado *DetectionWalls*.

Justo debajo de dicho objeto, podemos encontrar tres contenedores de *waypoints*: *WPContainerW01*, *WPContainerW02* y *WPContainerW03*. Estos contenedores son

del mismo tipo que los mencionados anteriormente, con la peculiaridad que las rutas que generan son las que denominamos “abiertas”.

2.2.5 CONFIGURACIÓN DEL CONTROLADOR

Este proyecto ha incluido la posibilidad de interactuar con el simulador a través de un controlador de juegos para ofrecer una experiencia más real.

El controlador que hemos usado para dicho cometido es el modelo G27 de la marca Logitech. Está formado por tres elementos: volante multifunción, pedales y palanca de cambio de velocidades. En este simulador se ha optado por la utilización de los dos primeros pues el vehículo principal es de transmisión automática. Para reducir la complejidad interactiva del usuario no se ha configurado el vehículo con transmisión manual, es decir, el sujeto no ha de manejar la palanca de cambio del automóvil.

Si se desea hacer uso de un controlador de este tipo en Unity es necesario configurar previamente el motor. Para ello, basta con añadir y modificar tres variables nuevas en la lista de parámetros de entrada del mismo, también llamadas Inputs.

En la siguiente imagen se observan las entradas administradas, entre las que se encuentran las tres que mencionamos: Horizontal, *Throttle* y *Brake*. A estas entradas se les ha insertado en el campo “Type” el nombre “Joystick Axis” para que sea el controlador el que aporte dicho valor de entrada.

Además, podemos ver como las tres entradas están duplicadas en la lista. Esto es motivo de que una de ellas responde a la entrada de teclado mientras la otra lo hace del *joystick*.

Funcionalmente, el Input *Brake* se encarga de pasar el valor recibido del pedal de freno como una variable de tipo *float* al sistema. Dicha variable es asignada al vehículo principal. Lo mismo ocurre con la entrada *Throttle*, ésta es la encargada de recibir del pedal de aceleración y trasmitirla al motor gráfico. Por último, *Horizontal* aporta la dirección al vehículo dependiendo del giro que el usuario realice sobre el volante.

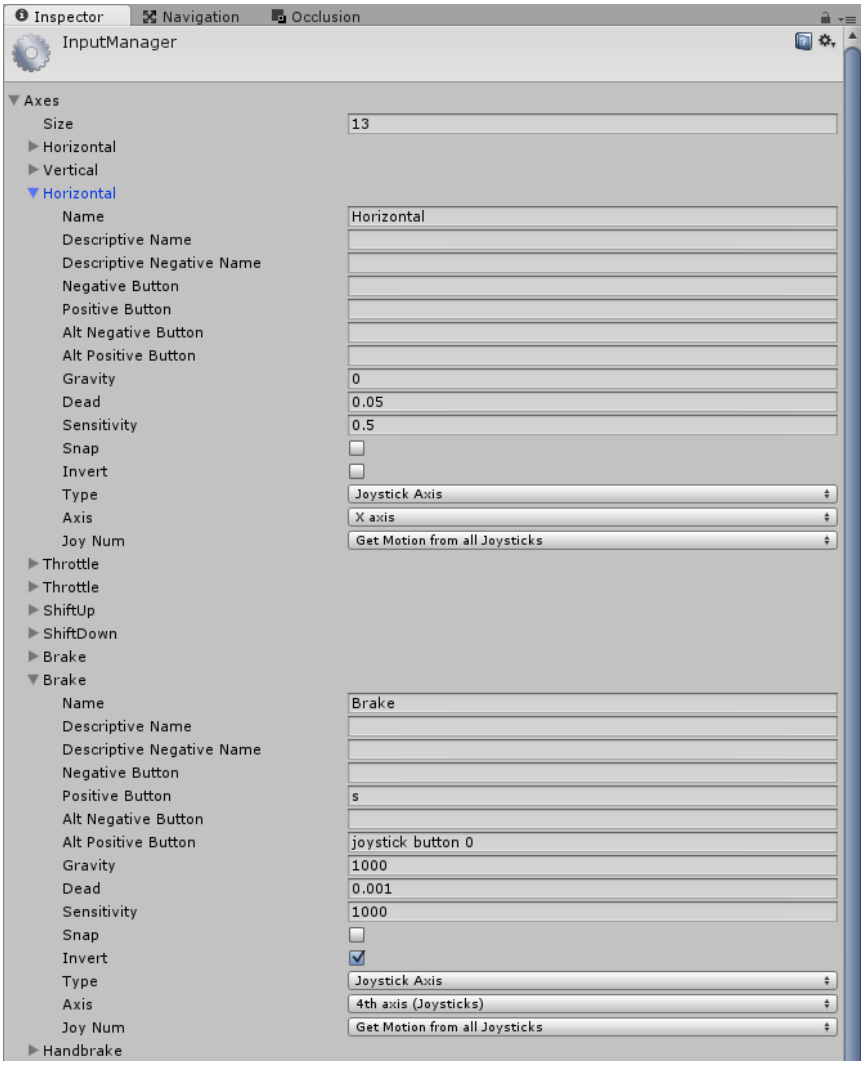

Imagen 23. Configuración del controlador

Líneas futuras y conclusiones

Es patente la necesidad que existe de tratar de evitar cualquier situación de peligro al volante de un vehículo. Para llevar a cabo medidas que puedan ayudar a dicho cometido es lógico pensar que debemos ser capaces de conocer los factores que intervienen en esas situaciones.

Una vez conocidos los parámetros que actúan durante la conducción, surge la necesidad de medirlos, y para ello, los simuladores son una herramienta fundamental.

Este simulador ha sido desarrollado con el objetivo de medir factores como la velocidad o el tiempo de reacción de un conductor en situaciones de riesgo. Por tanto, este proyecto se trata de una herramienta que apoya la labor de recreación de estas situaciones.

El simulador constituye una base donde queda fijada la forma de recrear escenarios, adquirir, representar y analizar datos; por tanto, este proyecto deja la puerta abierta a otras líneas futuras, dando lugar a posibles modificaciones y mejoras que podrán ser implementadas a partir del mismo.

Por otro lado, durante la realización de este trabajo, se han adquirido y potenciado habilidades y competencias necesarias para llevar a cabo proyectos de esta amplitud. Además, se han estudiado gran variedad de conceptos necesarios para investigar acerca de todo lo relacionado con el mundo del desarrollo 3D.

Aunque únicamente se ha trabajado sobre la parte funcional de la aplicación, es decir, la programación, la curiosidad siempre te lleva a intentar conocer el origen de todo lo que está relacionado con ella. Por ello, a los conocimientos básicos para la realización de este estudio se suman aquellos adicionales relativos a este sector de la ingeniería.

ANEXO A

Anexo A 1 *VideoScript*

Este *script* es el primero que se ejecuta en la simulación, pues es el encargado de generar el menú inicial del mismo.

Todo el código implementado en esta clase estará contenido en la función *OnGUI*. Esta función se encarga de representar todo lo relacionado con la interfaz gráfica de usuario, es decir, elementos como botones, etiquetas o áreas de texto.

Nuestro menú, además de representar el texto que se observa en la Imagen 12 nos permite recoger los valores que el usuario introduce en los campos “ruta para exportar archivo” y “nombre del archivo” y almacenarlos en variables de tipo *String*.

Por otro lado, Unity permite crear estilos de texto que pueden ser asociados a cualquier texto de la interfaz gráfica. Estos estilos, llamados *GUIStyle*, dan la posibilidad de aplicarle diferentes formatos al contenido. Un posible ejemplo es cambiar el color del texto cuando el puntero se sitúe sobre él. En la siguiente imagen se aprecia dicha clase, con todos sus parámetros, manualmente configurables.

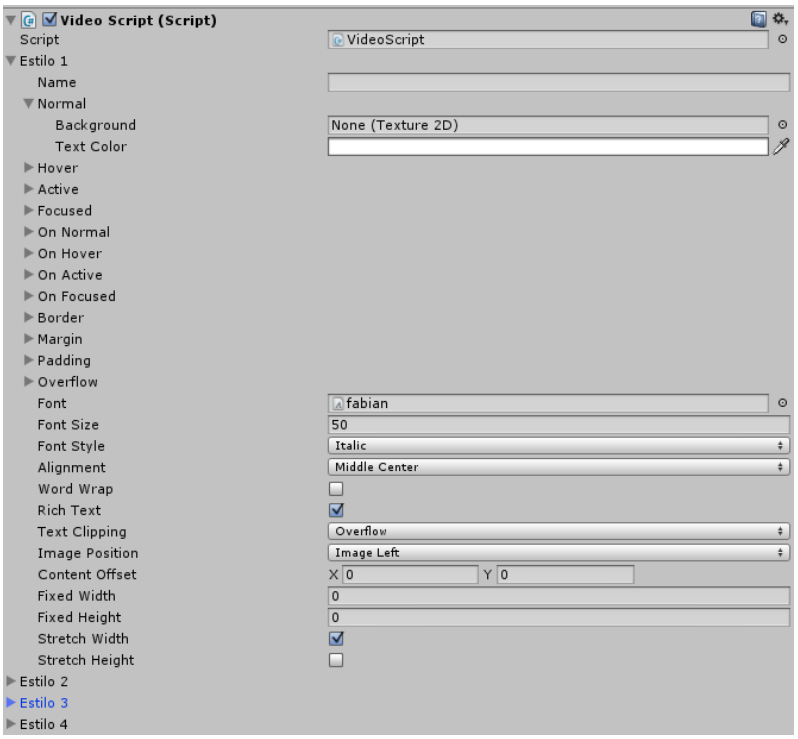

Imagen 24. VideoScript en el editor

Anexo A 2 *iTween*

iTween es una herramienta desarrollada por *PixelPlacement* exclusivamente para Unity. Se trata de un paquete que incluye tres clases principales: *iTween*, *iTweenEvent* y *iTweenPath*.

La primera de ellas es la clase más extensa pues alberga la implementación de todas las funciones que ofrece esta herramienta. Las dos clases restantes son un complemento de la anterior que permiten crear en el editor las rutas y eventos deseados. En nuestro proyecto se ha hecho uso de estas clases complementarias creando de forma manual dos rutas o *paths* en la escena, por lo que, no ha sido necesario escribir ningún código para ello.

Como se comentó anteriormente, hacemos uso de iTween para generar un vídeo alrededor de nuestro escenario. Esto se ha conseguido añadiendo una cámara a la escena y dotándola de movimiento siguiendo una ruta determinada. Concretamente, en nuestra escena existen dos rutas independientes que deberá recorrer la cámara de forma secuencial.

Para que un objeto, o en este caso una cámara, realice un comportamiento a través de una ruta, se requieren dos scripts: un *iTweenPath* y un *iTweenEvent*. Ambas clases deben estar presentes, pues una actúa sobre la otra y a la vez hacen uso de la clase principal *iTween*. Por ello, en nuestro caso existen cuatro scripts agregados a la cámara.

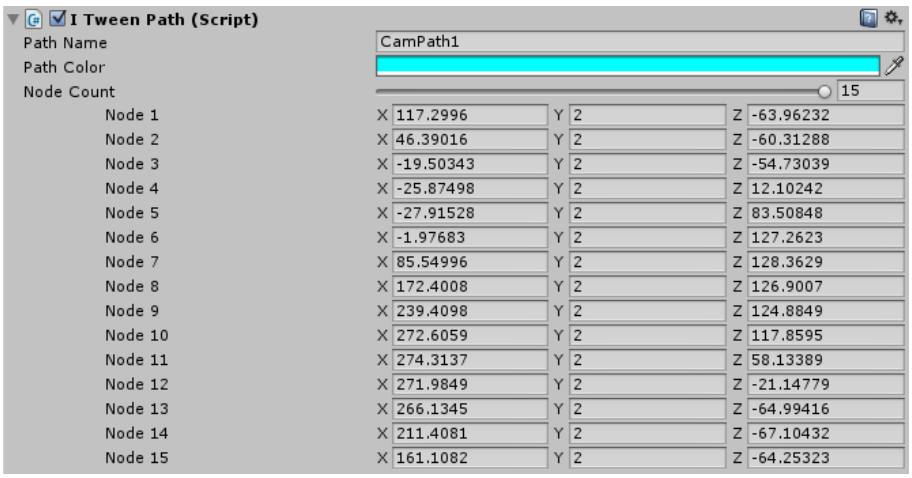

Imagen 25. iTweenPath en el editor

En la imagen anterior se muestra la clase *iTweenPath* en el editor de Unity tras haber definido una ruta. Esta ruta contiene 15 nodos que han sido colocados de forma determinada creando la ruta deseada, y es representada con una línea continua de color azul. Cada ruta ha de tener un nombre identificativo único para que la clase *iTweenEvent* trabaje sobre la misma.

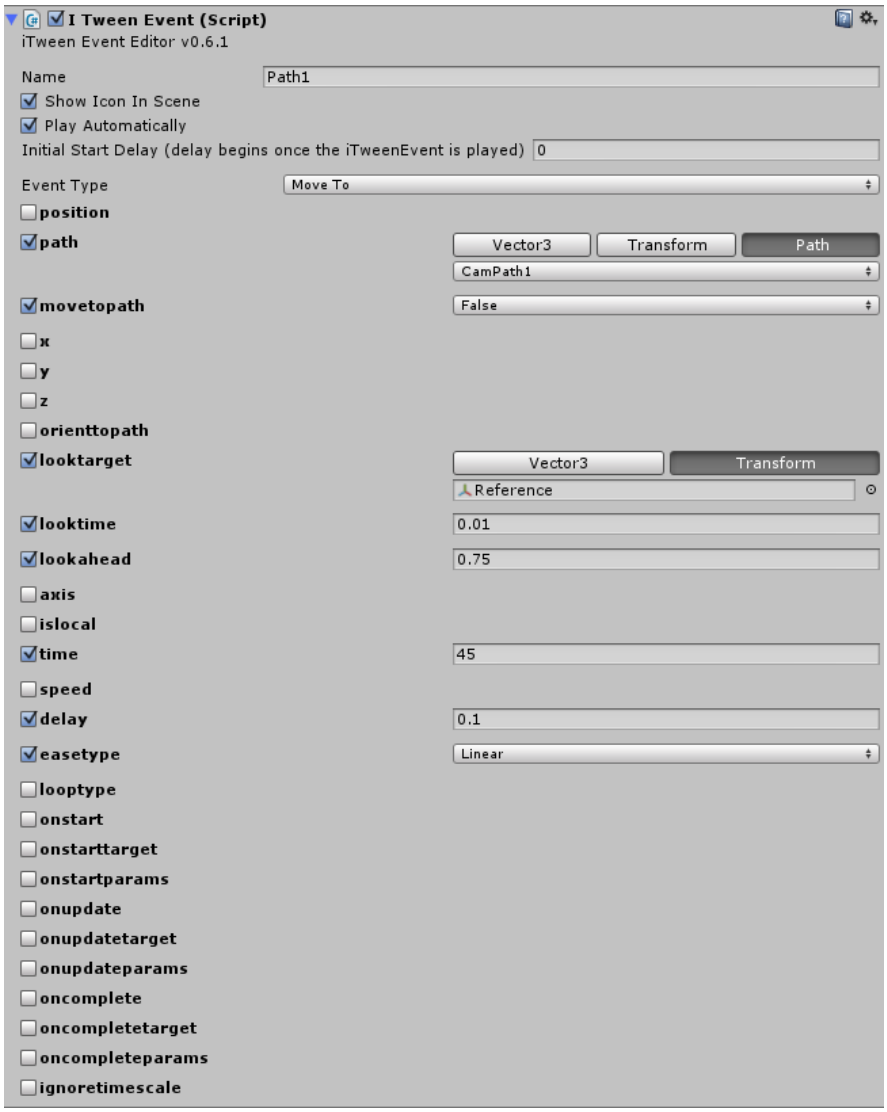

Imagen 26. iTweenEvent en el editor

En la imagen se puede observar la clase *iTweenEvent* asociada a la ruta descrita anteriormente. Esta clase permite hacer distintas configuraciones para recrear esa motricidad.

Si activamos la variable booleana *looktarget*, tal como se muestra, debemos administrarle un componente *Vector3* o *Transform* para que la cámara siempre esté

orientada hacia dicho punto. En este caso, nuestro objeto se orienta hacia el elemento *Reference* con una frecuencia determinada por la variable *looktime*. Como *looktime* es muy pequeño, dará la sensación de que siempre está orientado hacia nuestro objetivo.

Puesto que el efecto que perseguimos en esta ruta es la de realizar un recorrido sobre la carretera de la ciudad, el objeto de referencia debe ser un objeto móvil que circule justo por delante de nuestra cámara. De esta forma, si el objeto referencia realiza la misma ruta pero circula levemente por delante conseguimos que nuestra cámara esté en todo momento orientada hacia la carretera dando lugar al video deseado. Para lograr que el objeto *Reference* circule por delante, basta con añadirle la misma ruta para hacer que comience a moverse 0.1 segundo antes que la cámara. De ahí, que nuestra cámara tenga el valor de 0.1 en la variable *delay*.

Mientras tanto, el parámetro *path* debe estar marcado si el objeto se mueve sobre una ruta mientras que la variable *Event Type* contiene el tipo de acción o evento que se realiza, en este caso *Move*To, que hace una acción de movimiento. El tipo de movimiento que realiza el objeto en cuestión estará definido por las variables posteriores *easetype* o *looptype* si queremos que sea un movimiento único o repetido.

Una vez seleccionado el tipo de movimiento a realizar, iTween posee una gran variedad de funciones que permiten modificar el efecto de dicho movimiento.

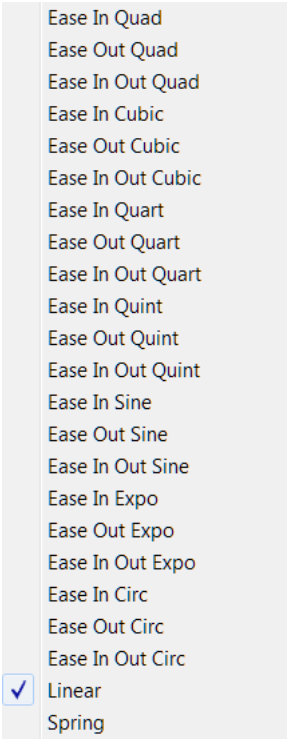

Imagen 27. Tipos de efecto de movimiento de iTween

Nuestra cámara realizará un movimiento con velocidad lineal, tal y como se muestra en la imagen, pero existen otras muchas configuraciones como Spring que genera un movimiento similar al de un muelle rebotando. El movimiento durará tanto tiempo como valga la variable *time*, en segundos.

Ahora que ya tenemos definida una ruta, hay que definir la segunda del mismo modo. Como queremos que este segundo movimiento comience justo al terminar el primero, solo hay que añadir un retardo de 45 segundos al segundo de los eventos.

A modo de ejemplo se muestra a continuación la implementación de la función *MoveTo* que utilizamos.

```
1435    public static void MoveTo(GameObject target, Hashtable args){
1436        //clean args:
1437        args = iTween.CleanArgs(args);
1438
1439        //additional property to ensure ConflictCheck can work correctly since Transforms are refrences:
1440        if(args.Contains("position")){
1441            if (args["position"].GetType() == typeof(Transform)) {
1442                Transform transform = (Transform)args["position"];
1443                args["position"]=new Vector3(transform.position.x,transform.position.y,transform.position.z);
1444                args["rotation"]=new Vector3(transform.eulerAngles.x,transform.eulerAngles.y,transform.eulerAngles.z);
1445                args["scale"]=new Vector3(transform.localScale.x,transform.localScale.y,transform.localScale.z);
1446            }
1447        }
1448
1449        //establish iTween:
1450        args["type"]="move";
1451        args["method"]="to";
1452        Launch(target,args);
1453    }
```

Imagen 28. Código de la función MoveTo de iTween

Queda demostrado que iTween en una herramienta muy útil para otorgar movimientos o añadir efectos a objetos en nuestra escena. En este proyecto únicamente se ha explotado una pequeña parte del potencial que posee dicho complemento.

Anexo A 3 *ExportData*

Esta clase ha sido completamente implementada por el autor para solventar dos necesidades: la realización de un menú de pausa, y la exportación de los datos de la simulación en tiempo real.

Para comprender como se ha realizado el menú de pausa podemos analizar el código que lo genera:

```
void OnGUI(){

    if ((endSimulation) && (Time.timeSinceLevelLoad - a) > 1.0f) {
        GUI.Label (new Rect (Screen.width / 2 - 200, Screen.height / 6, 400, 50), "FIN DE SIMULACION", style);
        if (GUI.Button (new Rect (Screen.width / 2 - 330, Screen.height / 2 +50, 660, 50), "REINICIAR SIMULACION", style2)) {
            escape = false;
            Time.timeScale = 1;
            FindObjectOfType<AudioListener> ().enabled = true;
            light.intensity = 1;
            Application.LoadLevel (Application.loadedLevelName);
        }

        if (GUI.Button (new Rect (Screen.width / 2 - 90, Screen.height / 2 +120, 180, 50), "MENU", style2)) {
            escape = false;
            Time.timeScale = 1;
            FindObjectOfType<AudioListener> ().enabled = true;
            light.intensity = 1;
            Application.LoadLevel ("MenuScene");
        }

        if (GUI.Button (new Rect (Screen.width / 2 - 90, Screen.height / 2 +190, 180, 50), "SALIR", style2)) {
            Application.Quit ();
        }
    }

    if (Input.GetKey (KeyCode.Escape)&&(!escape)) {
        escape = true;
    }


     if((escape)&&(!endSimulation)){
         Time.timeScale=0;
         FindObjectOfType<AudioListener>().enabled=false;
         light.intensity=0.5f;
         GUI.Label (new Rect (Screen.width / 2 - 100, Screen.height / 6, 200, 50), "PAUSA", style);

         if (GUI.Button (new Rect (Screen.width / 2 - 150, Screen.height / 2 - 30, 300, 50), "REANUDAR", style2)) {
             escape=false;
             Time.timeScale=1;
             FindObjectOfType<AudioListener>().enabled=true;
             light.intensity=1;
         }
         if (GUI.Button (new Rect (Screen.width / 2 - 330, Screen.height / 2 +50, 660, 50), "REINICIAR SIMULACION", style2)) {
             escape = false;
             Time.timeScale = 1;
             FindObjectOfType<AudioListener> ().enabled = true;
             light.intensity = 1;
             Application.LoadLevel (Application.loadedLevelName);
         }

         if (GUI.Button (new Rect (Screen.width / 2 - 90, Screen.height / 2 +120, 180, 50), "MENU", style2)) {
             escape = false;
             Time.timeScale = 1;
             FindObjectOfType<AudioListener> ().enabled = true;
             light.intensity = 1;
             Application.LoadLevel ("MenuScene");
         }

         if (GUI.Button (new Rect (Screen.width / 2 - 90, Screen.height / 2 +190, 180, 50), "SALIR", style2)) {
             Application.Quit ();
         }
     }
}
```

Imagen 29. Función OnGUI de la clase ExportData

Al igual que en la clase *VideoScript*, este tipo de contenido debe ir dentro de la función *OnGUI* de Unity que es la que se encarga de representar las interfaces gráficas.

Este código da solución a dos posibles situaciones en el simulador: el menú de pausa que aparece si el usuario presiona la tecla “Escape”, y el menú que aparece al terminar la simulación tras una colisión.

La primera parte de nuestra función se encarga de mostrar en pantalla tras una colisión el mensaje de “FIN DE SIMULACION”, además de representar los botones “REINICIAR SIMULACION”, “MENU” y “SALIR”.

Como aclaración, la función “*Application.LoadLevel*(“*MainScene*”)” cierra la escena actual y da paso a la escena “MainScene”. Además, la variable booleana “*endSimulation*” está activa si el vehículo ha colisionado en la escena y además ya se han exportado todos los datos necesarios.

Por otro lado, el fragmento que se encarga de exportar los parámetros de la simulación, crea en primer lugar los archivos con formato “.txt.” donde se almacenan dichos datos. El código que realiza esto es el siguiente:

```
//File variables
[HideInInspector]
public static string inputPath=@"C:\Users\Public\Documents\", inputCrashDataName= "crashInfo";
public string velPosPath, crashDataPath, crashDataName;
public string velPosDataName="VelPosInfo.data";
public float speed, otherCarSpeed;
private Vector3 position;
private bool collisioned = false, colWithAICar = false, colWithMap = false, braking=false;
private int cont=0;
private float speedFactor=3.6f, lastBrakeTime, lastBrakeTimeAI, a=0f,brakeDuration, endSimulationTime;
private bool endSimulation=false, escape=false;
//GUI variables
public GUIStyle style, style2;
public GameObject ligthObject;
private Light light = new Light ();
public Camera driverCamera, rearMirror;


void Start () {
    CreateFiles ();
    if (Screen.currentResolution.width / Screen.currentResolution.height == 4 / 3) {
        driverCamera.fieldOfView = 40f;
    } else {
        driverCamera.fieldOfView = 49;
    }
}

public void CreateFiles(){

    light = ligthObject.GetComponentInChildren<Light> ();
    light.intensity=1;
    //se crean las rutas
    velPosPath = inputPath + velPosDataName;
    UnityEngine.Debug.Log (velPosPath);
    crashDataName = inputCrashDataName + ".txt";
    crashDataPath = inputPath + crashDataName;
    //se eliminan si existen con anterioridad
    if(File.Exists(velPosPath)){
        File.Delete(velPosPath);
    }
    if(File.Exists(crashDataPath)){
        File.Delete(crashDataPath);
    }
    // se crean los archivos
    File.AppendAllText(velPosPath,"Tiempo(seg) \t Posicion en X \t Posicion en Z \t Velocidad(Km/h) \n");
    File.AppendAllText(velPosPath,Environment.NewLine);
    File.AppendAllText(velPosPath,Environment.NewLine);
}
```

Imagen 30. Código creación de archivos de la clase ExportData

Como vemos, la función “CreateFiles()”, en primer lugar, concatena los nombres de los archivos con la ruta destino que ha sido otorgada por el usuario en el menú inicial. Luego comprueba que los archivos no existen, y en caso de existir, los elimina.

Los dos archivos a crear son: “VelPosInfo.txt” y “crashInfo.txt”. El primero de ellos exporta cada cinco *frames* cuatro valores: tiempo en segundos, posición absoluta del vehículo en el eje X, posición absoluta en el eje Z, y velocidad en km/h.

Suponiendo que la simulación funciona a alrededor de 50 *frames* por segundo, cada 0,1 segundo se genera una nueva fila en el documento con el valor de esos cuatro parámetros en ese instante. Un ejemplo de este archivo es el siguiente:

Tiempo(seg)	Posicion en X	Posicion en Z	Velocidad(Km/h)
0.1	172.48	-65.56	0
0.2	172.479	-65.559	0
0.3	172.468	-65.558	15.1
0.5	172.379	-65.555	11.4
0.6	172.2	-65.546	10
0.7	171.937	-65.533	7.5
0.8	171.629	-65.518	10.9
0.9	171.294	-65.502	11.1
1.1	170.933	-65.484	11.3
1.2	170.553	-65.466	12
1.3	170.151	-65.446	12.6

Imagen 31. Ejemplo de archivo VelPosInfo.txt

Por otro lado, el archivo “crashInfo.txt” solo contiene información si la simulación ha terminado con una colisión.

La clase ExportData permite distinguir si dicha colisión se ha producido con otro vehículo o con algún objeto inmóvil del escenario. En caso de colisión se toman los datos: momento del impacto, velocidad a la que se circula al momento del impacto e instante en el que se comienza a frenar –en caso de frenada-.

Ahora bien, si la colisión se produce contra otro vehículo, se almacena también la velocidad del vehículo contrario, el instante en el que comienza a frenar dicho vehículo –en caso de frenada- y el tiempo de reacción del usuario respecto al del vehículo AI –en caso de frenada-. Si el conductor no frena en el instante antes de la colisión también queda reflejado, al igual que si el impacto se produce con el escenario. Un posible ejemplo del archivo crashInfo.txt es:

```
INFORMACION DE COLISION:

Se ha producido una colision en el instante 8.78 segundos cuando circulabas a una velocidad de 44.2 Km/h.
Se comenzo a frenar en el instante 8.24 segundos.
Duracion de la frenada: 0.54 segundos.
Ha colisionado con el entorno
```

Imagen 32. Ejemplo archivo CrashInfo.txt

Anexo A 4 *InterpWP*

Esta clase, fue creada tal y como se dijo en el apartado correspondiente, para generar nuevos puntos de referencia entre dos *waypoints* dados interpolando sus posiciones. De esta forma se consigue realizar una ruta de waypoints formada por un mayor número de puntos, dando lugar a una trayectoria mucho más precisa en lo que a la conducción se refiere.

En primer lugar se obtiene el *array* de puntos generados manualmente y se los pasa a una función llamada DPH. La función DPH es una adaptación de un fragmento de código que utiliza *iTween* para generar sus rutas. Esta función hace uso de otras dos funciones para conseguir así una interpolación no lineal de los puntos de referencia iniciales.

Una vez obtenida la interpolación, tenemos un array de puntos mucho más numeroso que el inicial, y ésta, sí es la ruta que siguen los vehículos de inteligencia artificial.

Anexo A 5 *InstantiateAICars*

La clase que se describe en este anexo es la encargada de la generación de tráfico en la escena. Cada contenedor, genera un tráfico de cuatro vehículos en instantes de tiempo equidistantes en un punto fijo, permitiendo que la ciudad esté compuesta por una serie de vehículos circulando de forma autónoma.

Cada contenedor de *waypoints* tiene asociada esta clase. Como vemos en la siguiente imagen, le podemos asignar la variable "*WaypointContainer*" que no será otra que el contenedor de puntos interpolados explicados en el anexo anterior, en este caso "*WPInterp02*". La variable "Posicion" de la imagen se trata del punto inicial donde se instancian los vehículos.

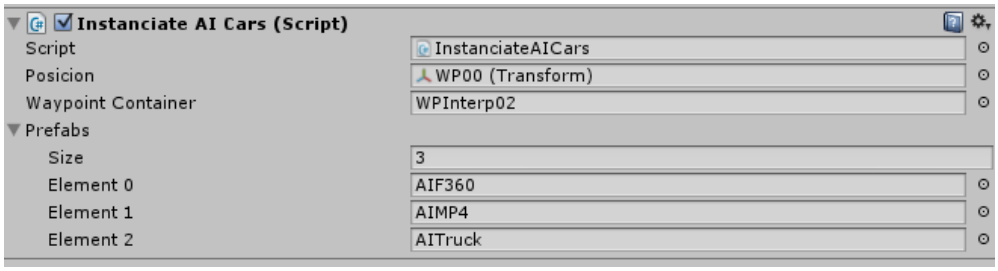

Imagen 33. InstantiateAICars en editor

Por último encontramos la variable "*Prefabs*", de tipo vector de *GameObjects*, cuyo tamaño es variable. Ésta, alberga los *prefabs* de los vehículos que pueden ser instanciados en dicha ruta. Esta variable otorga aleatoriedad a las simulaciones, pues el vehículo que se genera en un instante es definido por una variable aleatoria. Por lo que en el instante indicado, aparece en escena uno de los tres vehículos prefabricados contenidos en esa variable.

En primera instancia, esta clase detecta la ruta que tiene asociada para generar el vehículo en el lugar correcto. Luego se llama a la función "*InstantiateAI*()":

```
public void InstantiateAI(){
    if (flags [0] == false) {
        GameObject AICarClone = (GameObject)Instantiate (prefabs[alea[0]], posicion.position, posicion.rotation);
        AICarClone.GetComponent<AICar> ().waypointContainer = waypointContainer;
        AICarClone.GetComponent<Drivetrain>().gear=2;
        AICarClone.GetComponent<AICar> ().pathNumber =  number;
        if(number>3)
            AICarClone.GetComponent<AICar> ().soloIda=true;
        flags[0]=true;
    }


    if ((Time.timeSinceLevelLoad > aleaInc+n/2) && (flags [1] == false)&&(number<4)) {
        GameObject AICarClone2 = (GameObject)Instantiate (prefabs[alea[1]], posicion.position, posicion.rotation);
        AICarClone2.GetComponent<AICar> ().waypointContainer = waypointContainer;
        AICarClone2.GetComponent<Drivetrain>().gear=2;
        AICarClone2.GetComponent<AICar> ().pathNumber = number;
        flags[1]=true;
    }
    if ((Time.timeSinceLevelLoad > aleaInc*2+n/2) && (flags [2] == false)&&(number<4)) {
            GameObject AICarClone3 = (GameObject)Instantiate (prefabs[alea[2]], posicion.position, posicion.rotation);
        AICarClone3.GetComponent<AICar> ().waypointContainer = waypointContainer;
        AICarClone3.GetComponent<Drivetrain>().gear=2;
        AICarClone3.GetComponent<AICar> ().pathNumber = number;
        flags[2]=true;
    }
    if ((Time.timeSinceLevelLoad > aleaInc*3+n/2) && (flags [3] == false)&&(number<4)) {
        GameObject AICarClone4 = (GameObject)Instantiate (prefabs[alea[3]], posicion.position, posicion.rotation);
        AICarClone4.GetComponent<AICar> ().waypointContainer = waypointContainer;
        AICarClone4.GetComponent<Drivetrain>().gear=2;
        AICarClone4.GetComponent<AICar> ().pathNumber = number;
        flags[3]=true;
    }

}
```

Imagen 34. Función InstantiateAI de la clase InstantiateAICars

Se generan cuatro automóviles por ruta. El primero se crea en el momento que comienza la simulación, mientras que los tres restantes lo hacen aproximadamente en los instantes 7, 14 y 21 segundos después.

La variable "*flags*" no es más que un *array* de booleanos que indica para cada uno de los 4 instantes si el vehículo ha sido instanciado. Se instancia un vehículo si han pasado los 7 segundos que los separa más un incremento de n/2 segundos siendo "n" el número de ruta en cuestión. Ésta es una medida para evitar una sobrecarga computacional sobre la GPU en dichos instantes, pues cada ruta dispondrá de un incremento de tiempo diferente a las demás.

Por último, la condición "number<4" permite separar las rutas abiertas de las cerradas gracias a que las rutas abiertas tienen un identificador mayor a tres por tanto únicamente se instancia un vehículo en ella.

Anexo A 6 *UnityCar* Pro

Teniendo en cuenta que uno de los objetivos de nuestro proyecto es el de recrear con la mayor precisión posible el comportamiento de un vehículo. Entonces, resulta indispensable poder calcular en simulación el mayor número de fuerzas que intervienen en su funcionamiento. Para ello hacemos uso de la herramienta *UnityCar Pro*, que tiene en cuenta todas las fuerzas y variables que hacen funcionar un vehículo en la vida real, como por ejemplo, el tiempo entre cambios de velocidad, potencia del motor, fuerza de torsión del motor, inercia del motor, etc.

UnityCar Pro está disponible en el *Asset Store* de *Unity*, y se trata de un paquete cuya licencia no es gratuita. Dicho paquete contiene un gran número de clases que lo hacen funcionar que podemos ver en la Imagen 16.

A continuación se describen los scripts que son instanciables y que por tanto están asociados a los vehículos de escena.

En primer lugar encontramos la clase "Axles" que se encarga de obtener y configurar los ejes que posee el vehículo en cuestión.

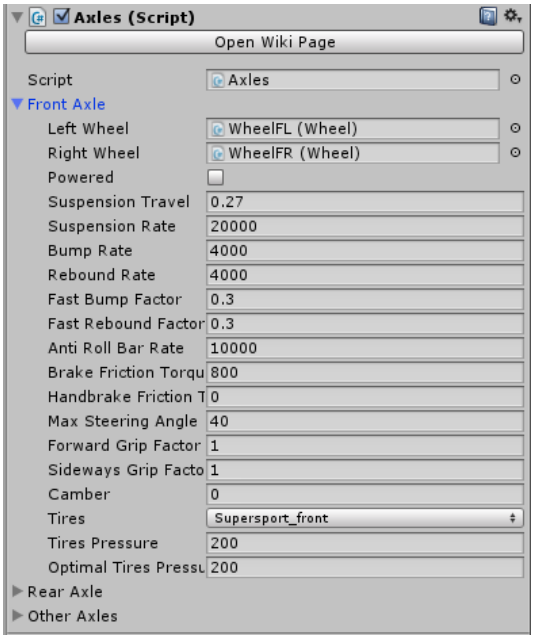

Imagen 35. Clase Axles en editor

La imagen muestra desplegada la información relativa al eje frontal del coche, pero en la parte inferior podemos encontrar el resto de ejes.

Como se observa, esta clase tiene en cuenta todo lo referente a los ejes y ruedas del automóvil. Podemos destacar como ejemplos, el máximo ángulo de giro, el factor de amortiguación o la fricción del freno de mano en dicho eje.

La siguiente clase que se adjunta es “*Drivetrain*”, que traduciendo al español es “transmisión”. Por tanto, ésta es la responsable de todas las fuerzas que afectar al motor del vehículo.

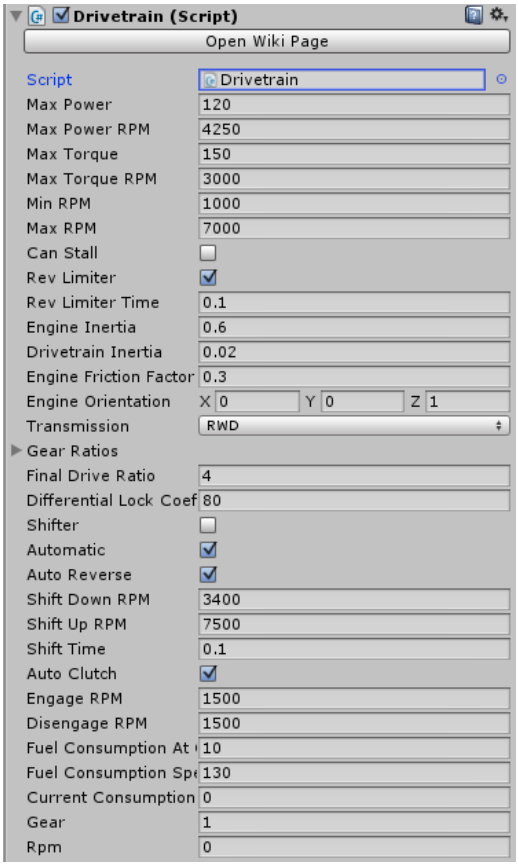

Imagen 36. Clase Drivetrain en editor.

En ella se tienen en cuenta variables como la potencia, fuerza de torsión, máximas y mínimas revoluciones o ratios de las velocidades. También nos da la posibilidad de configurar el coche con transmisión manual o automática, e incluso modificar el eje de tracción del mismo.

Otra de las clases a tener en cuenta es la llamada “*CarDynamics*”:

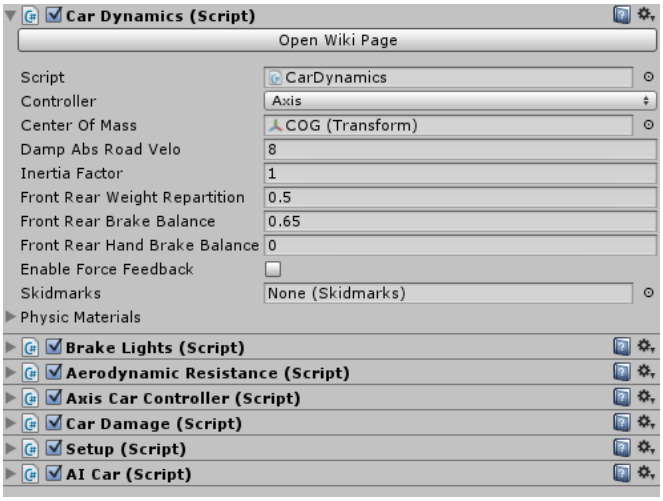

Imagen 37. Clase CarDynamics en editor

Dicha clase trabaja con variables que afectan al controlador que hace funcionar a UnityCar y otros factores como el factor de inercia del vehículo, el centro de gravedad del mismo, o el reparto de pesos de los frenos. Además, permite configurar las huellas que pueda dejar el vehículo en frenadas o derrapes.

Por otra parte, la clase que gestiona las fuerzas de rozamiento del coche con fluidos como el aire o el agua es la llamada "*AerodynamicsResistance*". Ésta permite configurar el parámetro de coeficiente de rozamiento y la dimensión frontal del mismo en metros cuadrados, modificando así el factor de rozamiento.

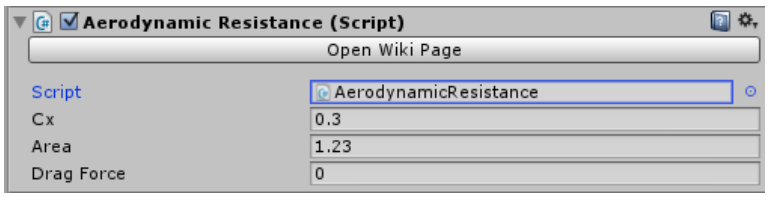

Imagen 38. Clase AerodynamicsResistance en editor

La variable *DragForce* es de sólo lectura, e indica la fuerza de rozamiento en cada instante.

La siguiente clase que encontramos asociada a los vehículos es "*AxisCarController*" y su función es gestionar las entradas de control que moverán el vehículo.

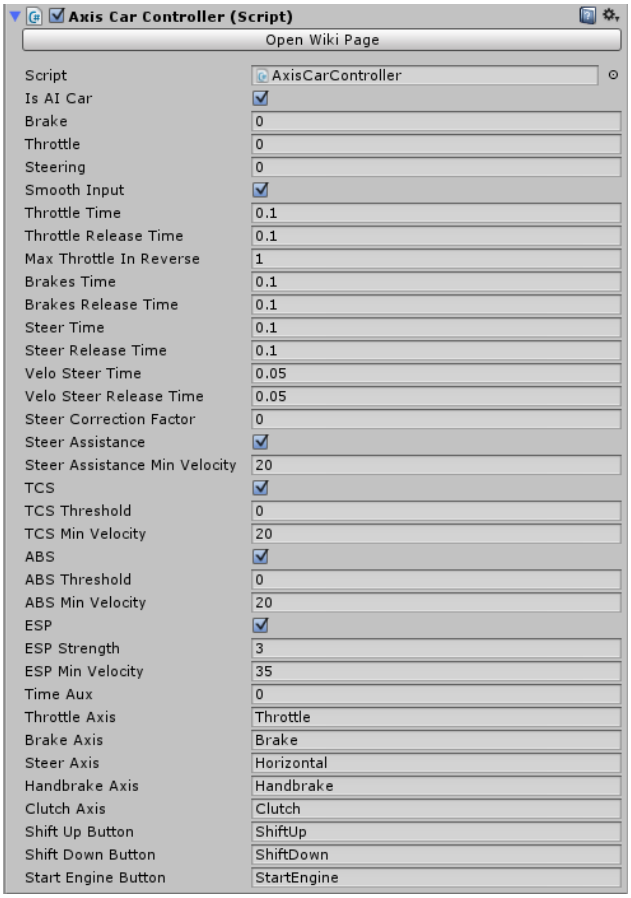

Imagen 39. Clase AxisCarController en editor.

Es importante destacar que esta clase hereda de la clase "*CarController*", por tanto es la que proporciona a sus clases hijas la mayor parte de variables. La función de esta clase es modificar la forma en la que el automóvil reacciona frente a las entradas de control. Para ello tiene en cuenta aspectos como el suavizado de aceleración y frenada, o el tiempo necesario para que el acelerador alcance su valor máximo. Además permite configurar el sistema de control de tracción –TCS-, ABS o el sistema de estabilidad –ESP-.

Es importante decir que esta clase tuvo que ser levemente modificada para que tanto el vehículo principal como los de inteligencia artificial pudieran hacer uso de este paquete. Para ello, se introdujo una variable booleana llamada "*IsAICar*" que distingue ambos tipos de objetos. Para el caso de los vehículos AI, la clase *CarController* debe obtener las entradas de movimiento de la clase AICar, mientras que para el coche principal, dichos valores son aportados por las entradas de teclado o *joystick*.

Por último, encontramos las clases “CarDamage” y “Setup”. La primera, se encarga de permitir que el vehículo reciba daños en su aspecto físico tras una colisión. La clase, “Setup”, inicializa todas las variables de UnityCar con unos valores concretos dependiendo del vehículo en cuestión. Por ejemplo, si se trata de un vehículo deportivo, las variables que afectan a la potencia del motor serán diferentes a las de un vehículo utilitario o a las de un camión. Del mismo modo para el resto de variables como el peso, rozamiento o ángulo de giro.

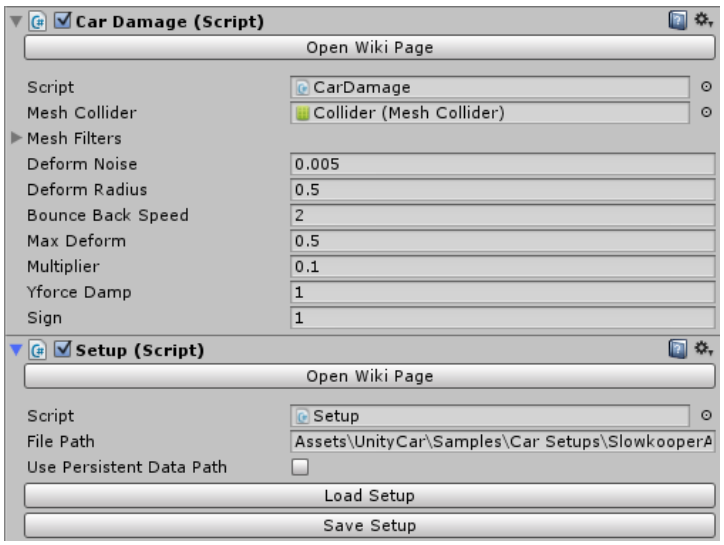

Imagen 40. Clases CarDamage y Setup en editor.

Anexo A 7 *AICar*

Como se comentó en anterioridad, la clase “*AICar*” aporta la inteligencia artificial a los vehículos y está formada por dos funcionalidades principales: el seguimiento de *waypoints* y el control de frenada y colisiones.

Para dar forma a las explicaciones del apartado Vehículos con IA vamos a incluir fragmentos del código que la constituye.

```
47      void Start () {
48          motorForce = 0;
49          GetWaypoints();
50          SetPathNumber ();
51
52      }
53
54      void FixedUpdate () {
55          NavigateTowardsWaypoint ();
56          HitControl ();
57          BrakeSystem ();
58          BrakeCheck ();
59      }
```

Imagen 41. Implementación de funciones Start y FixedUpdate de AICar

Para comenzar con esta clase, encontramos las dos clases principales: las funciónes *Start* y *FixedUpdate*. La primera de ellas, es la primera en ejecutarse y por ello es la que se encarga de obtener los *waypoints* que recorre el vehículo para luego detectar el número de ruta en la que se encuentra y crear un array con los puntos de referencia en los que debe frenar.

```
public void GetWaypoints(){
    Transform[] potencialWaypoints = waypointContainer.GetComponentsInChildren<Transform>();
    waypoints = new Transform[potencialWaypoints.Length-1];

    int i = 0;
    foreach(Transform potencialWaypoint in potencialWaypoints)
        if(potencialWaypoint != waypointContainer.transform)
            waypoints[i++] = potencialWaypoint;
}
```

Imagen 42. Implementación función GetWaypoints.

El flujo principal lo lleva a cabo *FixedUpdate* que se ejecuta de forma ininterrumpida. Por orden de ejecución se realizan las siguientes tareas: *NavigateTowardsWaypoints*, *HitControl*, *BrakeSystem* y *BrakeCheck*.

La primera de ellas es la encargada del movimiento del vehículo. Para ello hace que éste recorra todos los waypoints, encargándose de las aceleraciones y los giros.

```
public void NavigateTowardsWaypoint(){

    speed = Mathf.Abs (transform.InverseTransformDirection (GetComponent<Rigidbody>().velocity).z);
    realspeed = transform.InverseTransformDirection (GetComponent<Rigidbody> ().velocity).z;
    motorForce = Mathf.Clamp (motorForce, -1f, 0.8f);

    Vector3 RelativeWaypointPosition = transform.InverseTransformPoint(new Vector3( waypoints[currentWaypoint].position.x,
                transform.position.y,waypoints[currentWaypoint].position.z));
    steerAngle =(RelativeWaypointPosition.x) / RelativeWaypointPosition.magnitude;
    // El coche frena en las curvas.

    if (justStop) {
        if(tooClose){
            if(realspeed>1f){
                motorForce-=motorForce/4;
            }else{
                motorForce=0;
            }
        }
        if(realspeed>0.7f){//0.05f
            motorForce=-brakeFactor;
        }else
            motorForce = 0;
        //Debug.Log("Mantenemos sin acelerar");
    } else {
        if ((!braking) && (!brakingByDistance) && (Mathf.Abs(steerAngle) < 0.7f)) {//probar con menos
            motorForce = RelativeWaypointPosition.z / RelativeWaypointPosition.magnitude;//- Mathf.Abs (steerAngle / 3);
        } else
            if (steerAngle > 0.75)
            motorForce = 0.4f;
        else{
            if(Time.timeSinceLevelLoad>4)
                motorForce = 0;
        }
        if (RelativeWaypointPosition.magnitude < 15) {//15 original
            if(!soloIda){
                currentWaypoint++;
                if (currentWaypoint >= waypoints.Length) {
                    currentWaypoint = 1;
                }


            }else{
                if(currentWaypoint+1>=waypoints.Length-1 && !justStop){ //tamaño 46
                    if(realspeed<1){
                        Debug.Log("hay que frenar");
                        justStop=true;
                    }
                    else
                        motorForce=-brakeFactor;
                }
                else
                    currentWaypoint++;
            }
        }
    }
}
```

Imagen 43. Implementación función NavigateTowardsWaypoint

Por otro lado, *HitControl* aplica la evasión de colisiones formado por los Raycasts o rayos de colisión. El siguiente fragmento de código modifica la longitud del rayo dependiendo de la velocidad actual y crea el *Raycast* frontal.

```
public void HitControl(){
    raycastDistance = (speed * raycastRatio);//en 1.2 va bien

    //Raycast frontal
    if (Physics.Raycast (transform.position, transform.TransformDirection (Vector3.forward), out hit, raycastDistance)) {
        if ((!onCollision)&&(!justStop)) {
            if ((hit.collider.tag == "AICar") || (hit.collider.gameObject.tag == "Car")) { //comprobamos que nos aproximamos a otro vehiculo
                brakeForCollision = Time.timeSinceLevelLoad;
                if (Time.timeSinceLevelLoad - timeAux > 3) {
                    brakeForCollision = 0;
                }
                timeAux = Time.timeSinceLevelLoad;
                onCollision = true;
                prueba = hit.distance;
            }
        }
    }
```

Imagen 44. Fragmento de implementación HitControl

Una vez lanzado el rayo, si se detecta colisión se ejecuta el siguiente código que permitirá reducir la velocidad mientras el objeto siga dentro del alcance del rayo.

```
//CONTROL DE FRENADA POR DISTANCIA
if (onCollision) { //detecta colision el raycast?
    if (hit.distance < 0.1f) {//4.5f //demasiado cerca
        if(!justStop){
            justStop = true;
        }
    }
    if (realspeed > 0.05 && !justStop) {     //frenamos si llevamos cierta velocidad
        brakingByDistance = true;
        motorForce = -brakeFactor;
    } else {
        justStop=true;
        brakingByDistance = false;
    }
}
if ((Time.timeSinceLevelLoad - timeAux > 0.1f)) {//comprueba cada 0.15sec si estamos en colision.
    //porque el hit no se queda continuamente activado.Hay que dar continuidad. SOLO SI NO ESTAMOS DEMASIADO CERCA
    onCollision = false;
    brakingByDistance = false;

}
```

Imagen 45. 2º Fragmento de implementación HitControl

Más tarde, la función *BrakeSystem* detecta si el vehículo debe frenar por *waypoint*.

```
public void BrakeSystem(){
    if ((brakeWaypoint.Length > brakeWaypointIndex) && (waypoints [currentWaypoint - 1].position == brakeWaypoint [brakeWaypointIndex].position)&&(braking==false)) {
        braking = true;
        brakeForCollision=Time.timeSinceLevelLoad;
        if(Time.timeSinceLevelLoad-timeAux>3){
            brakeForCollision=0;
        }
        if (braking)
            desiredSpeed = speed / curveFactor [brakeWaypointIndex];
        brakeWaypointIndex++;
        if (brakeWaypointIndex >= brakeWaypoint.Length) {
            brakeWaypointIndex = 0;
        }
    }
}
```

Imagen 46. Implementación función BrakeSystem

Como vemos, si debe frenar al llegar a un *waypoint* determinado, la velocidad resultante tras la frenada será la denominada "*desiredSpeed*", que depende de la velocidad actual y de un factor de frenada asociado a dicho waypoint.

Y por último, BrakeCheck comprueba cuando debe de terminar de frenar el vehículo, además, es la que realmente aplica la fuerza negativa al motor para que se aplique la frenada.

Anexo A 8 *WallScript*

En cuanto a la parte funcional que permite la generación de tráfico activa, se realiza mediante los muros de detección, tal como se explicó anteriormente. Para ello se

ha implementado la clase “*WallScript*”. Ésta, está asociada a cada uno de los muros de la escena.

En primera instancia, detecta qué ruta tiene asociada e inicializa las variables necesarias. Más tarde, puesto que dichos detectores son Colliders, al ser atravesados ejecutan las funciones “*OnTrigger*” de las clases que tiene añadidas. Una de dichas funciones es la llamada “*OnTriggerEnter*” que es ejecutada únicamente una vez y en momento en el que se alcanza el primer contacto.

```
4  public class WallScript : MonoBehaviour {
5
6      private GameObject WPContainer,WPInterp;
7      private bool enabled=false;
8
9      void Start () {
10
11         if (gameObject.name == "Wall1") {
12             WPContainer = GameObject.Find ("WPContainerW01");
13             WPInterp=GameObject.Find("WPInterpW01");
14         } else if (gameObject.name == "Wall2") {
15             WPContainer = GameObject.Find ("WPContainerW02");
16             WPInterp=GameObject.Find("WPInterpW02");
17         }else if (gameObject.name == "Wall3") {
18             WPContainer = GameObject.Find ("WPContainerW03");
19             WPInterp=GameObject.Find("WPInterpW03");
20         }
21     }
22
23     void OnTriggerEnter(Collider c){
24         if (c.tag == "Car" && !enabled) {
25             WPContainer.GetComponentInChildren<InstanciateAICars> ().enabled = true;
26             WPInterp.GetComponentInChildren<InterpWP> ().enabled = true;
27             enabled=true;
28         }
29     }
30
31
32 }
```

Imagen 47. Implementación clase WallScript

Si el objeto que entra en contacto es un objeto etiquetado como “Car”, o lo que es lo mismo, el vehículo principal, se activan los scripts *InstantiateAICars* y *InterpWP* de la ruta de waypoints asociada al muro en cuestión. Al activarse, entran en escena, lo que provoca que se instancie el vehículo y comience a trazar su trayectoria.

Anexo A 9 *ProcessData.m*

Tras realizar una simulación de conducción, se hace indispensable poder analizar la información obtenida. Para ello, se ha creado un script en *Matlab* llamado “*ProcessData.m*” que permite representar los datos resultantes. Este script está

incluido entre los archivos que forman la aplicación, y al finalizar la simulación es copiado en el directorio que el usuario introduce en el menú inicial.

La siguiente imagen muestra la implementación de este script.

```
clc;
close all;
fileToRead1='VelPosInfo.txt';
newData1 = importdata(fileToRead1);
vars = fieldnames(newData1);
for i = 1:length(vars)
    assignin('base', vars{i}, newData1.(vars{i}));
end

tiempo=data(:,1);
velocidad=data(:,7);
figure;
plot(tiempo,velocidad,'linewidth',3,'Color','g');
title('Velocidad del vehiculo')
xlabel('Tiempo (s)')
ylabel('Velocidad (km/h)')

varX=data(:,3);
varZ=data(:,5);
x = linspace(min(varX),max(varX));
Z = linspace(min(varZ),max(varZ));
figure;
plot(varX,varZ,'linewidth',3,'Color','r');
hold on;
plot(varX(1),varZ(1),'ro','markersize',10,'markerfacecolor','r');
plot(varX(end),varZ(end),'rx','markersize',12,'linewidth',3);
title('Posicion del vehiculo')
xlabel('X')
ylabel('Z')
```

Imagen 48. Implementación ProcessData.m

Este código lee en primera instancia el archivo exportado por Unity con nombre “VelPosData.txt” y extrae los datos útiles de sus columnas. Luego, simplemente representa la información en dos gráficas separas. La primera, muestra la velocidad del vehículo durante la simulación frente al tiempo de la misma.

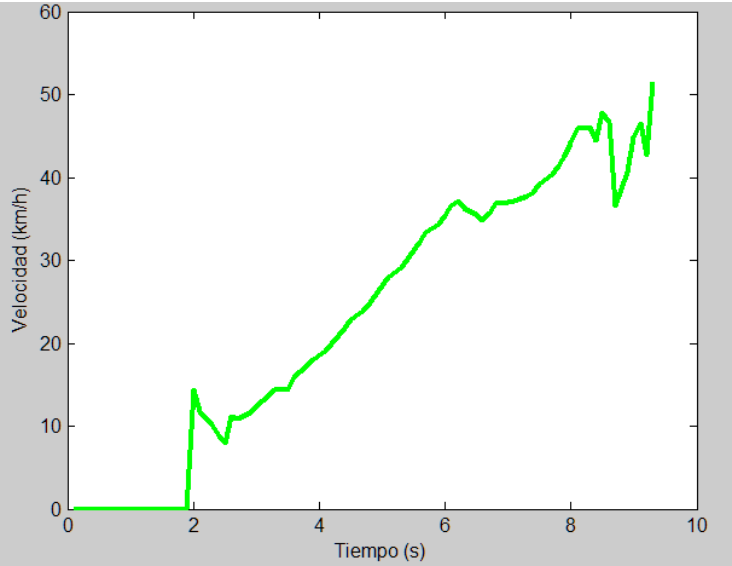

Imagen 49. Gráfica Vel-Tiempo ProcessData

La representación restante contiene la información del movimiento que realiza el usuario, es decir, pinta el recorrido que sigue el vehículo. En el eje X de la imagen se refleja la posición absoluta del vehículo en el eje X del espacio global de Unity, mientras en el eje Y, el eje Z del mismo. Con un círculo de color sólido se representa el punto de partida del vehícul0,o mientras que la cruz indica el lugar donde se ha producido la colisión.

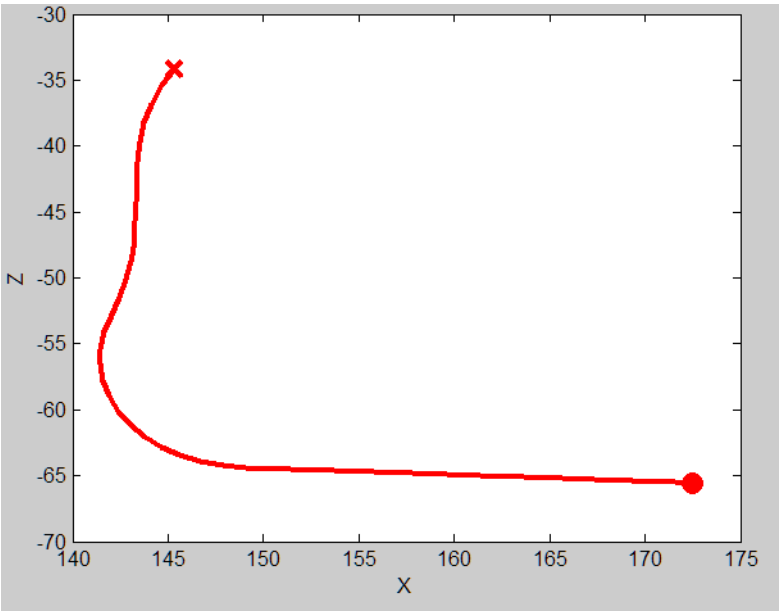

Imagen 50. Gráfica X-Z Process Data

Bibliografía

[1] M. Geig, «InformIT,» Diciembre 2013. [En línea]. Available: http://www.informit.com/articles/article.aspx?p=2162089&seqNum=2.

[2] Unity3D, «https://unity3d.com/es,» Unity Technologies, 12 1 2010. [En línea]. Available: http://docs.unity3d.com/ScriptReference/. [Último acceso: 12 Marzo 2015].

[3] A. Mocholí, «Yeeply,» 2012. [En línea]. Available: https://www.yeeply.com/blog/desarrollo-de-juegos-con-unity-3d/.

[4] T. S. ShiVa Technologies, «Shiva Engine,» 2014. [En línea]. Available: http://www.shivaengine.com/.

[5] G. Games, «Garage Games - Torque3D,» 2000. [En línea]. Available: http://www.garagegames.com/products/torque-3d.

[6] Image Space Inc., «RFactor,» Image Space Inc., 1992. [En línea]. Available: http://www.rfactor.net/.

[7] C. T. «CarX,» CarX Technologies, [En línea]. Available: http://www.carx-tech.com/.

[8] J. Cuello, «Openlanak,» 2012. [En línea]. Available: http://www.openlanak.com/2012/06/vdrift-simulador-de-carreras-opensource.html.

[9] OpenDS, 2011. [En línea]. Available: https://opends.de/.

[10] F. W. d. P. LWP, «La Web Del Programador,» 2010. [En línea]. Available: http://www.lawebdelprogramador.com/foros/C-sharp/.

[11] B. Berkebile, «iTween,» PixelPlacement™, 2012. [En línea]. Available: http://itween.pixelplacement.com/index.php.

[12] Unity3D, «Unity Scripting Reference,» 2010. [En línea]. Available: http://docs.unity3d.com/ScriptReference/.

[13] UnityCarPro, «Unity Car Wiki,» 07 Diciembre 2012. [En línea]. Available: http://unitypackages.net/unitycar/wiki/index.php/Main_Page.

[14] F. Unity Spain, «Unity Spain,» 2011. [En línea]. Available: http://unityspain.com/.

[15] F. Discom, «Canal de Youtube,» 18 Marzo 2011. [En línea]. Available: https://www.youtube.com/user/FenixDiscom.

[16] Microsoft, «C# Documentation,» [En línea]. Available: https://msdn.microsoft.com/es-es.

[17] F. GameDev, «Gamedev.net,» 1999. [En línea]. Available: http://www.gamedev.net/topic/644163-torque-3d-vs-unity-3d/.

[18] Vdrift. [En línea]. Available: http://vdrift.net/.

[19] Wikipedia, Wikimedia Foundation, Inc., [En línea]. Available: https://es.wikipedia.org/wiki/Unity_(software).