ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Simulador De Accidentes De Tráfico Mediante Motor De Videojuegos Unity

AUTOR: Alejandro Fernández Arroyo
DIRECTOR: Esteban Egea López

Septiembre / 2013

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	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. Para ello se hará uso de diagramas de flujo que permitirán entender la secuencia de ejecución. Además se darán a conocer las interfaces gráficas desarrolladas para el manejo de la simulación.

Por último, se comentarán las líneas futuras del proyecto y cuáles son los objetivos cumplidos con este proyecto.

Titulación	I.T.T. Especialidad Telemática
Intensificación	
Departamento	Tecnologías de la Información y Comunicaciones
Fecha de Presentación	Septiembre 2013

Agradecimientos

En primer lugar, agradecer a mi director de proyecto Esteban Egea por darme la oportunidad de realizar este proyecto. Por facilitarme las cosas cuando parecía que decaían y por todo su tiempo y dedicación.

Agradecer por supuesto a mis padres y hermana por todo su apoyo y confianza depositada en mí. También agradecer esas palabras de ánimo cuando no todo sale bien.

Dar gracias también a Mireya porque siempre está ahí para ayudarme y apoyarme. Porque es la que ha confiado en mí desde el primer momento y porque es la principal culpable de que haya llegado hasta aquí.

Tampoco quiero olvidarme de agradecer a mis abuelos, que son personas ejemplares para mí y me han dado siempre ánimos para continuar.

Gracias también a todos los compañeros con los que he compartido muchos momentos. Compañeros, que fuera de la universidad, seguirán estando ahí para seguir compartiendo experiencias.

Por último, agradecer a todos esos profesores que están siempre ahí para ayudar a los alumnos y ponen a disposición su tiempo y esfuerzo para que no dejemos de aprender.

"Nuestra recompensa se encuentra en el esfuerzo y no en el resultado.

Un esfuerzo total es una victoria completa"

Mahatma Gandhi

ÍNDICE

1. Resumen y objetivos

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 de tráfico actuales se han desarrollado para la evaluación del flujo de tráfico en condiciones 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 ofrecen librerías de simulación de comportamiento físico de objetos muy realista así como de modelado gráfico del entorno. 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 muy a tener en cuenta, ya que los resultados obtenidos son factibles y con un coste asequible.

El principal objetivo de este proyecto trata de experimentar la reacción de los conductores frente a una situación de posible accidente mediante la realización de un simulador de conducción en primera persona. La aplicación permite realizar varios tipos de simulaciones diferentes pondrán a prueba los reflejos del conductor.

Este simulador está creado en Unity3D [1], 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 ofrece al usuario la posibilidad de conducir mediante un controlador de juegos - formado por volante y pedales – que otorga mayor realismo a las simulaciones.

También cabe la opción de modificar el nivel de tráfico en la simulación, así como de decidir si queremos conducir un vehículo automático o manual.

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

2. Tecnologías utilizadas

2.1 Unity3D

El motor del juego es el elemento más importante de todo el proyecto. Está compuesto por las instrucciones necesarias para la representación de la aplicación, el sonido, la renderización 2D y 3D, la inteligencia artificial, etc. La función de un motor gráfico es servir de puente entre el usuario final y la aplicación, por ello 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 PCs, Android, Wii, Xbox o iOS.

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

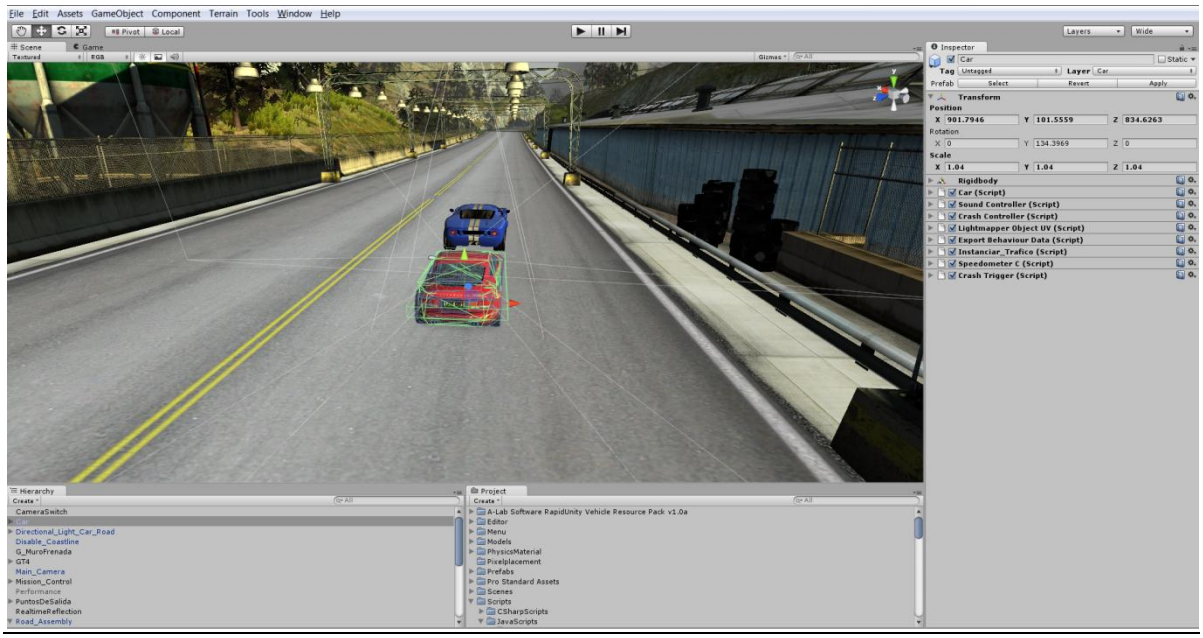


Fig. 2-1 Editor del IDE de Unity3D

En la imagen anterior se muestra el IDE de Unity. A la derecha de la imagen se encuentra el *Inspector*. Una vez que seleccionemos un objeto en la escena se mostrarán todos los componentes que tiene añadidos en dicha pestaña.

En la parte inferior izquierda está la pestaña *Hierarchy*, que contiene todos los *GameObjects* de la escena ordenados jerárquicamente.

Por último, a la derecha de la pestaña *Hierarchy*, encontramos la pestaña *Project*. En ésta, se muestran todos los recursos que tenemos disponibles en el proyecto. Como vemos, se divide en directorios contendrán componentes diferentes: texturas, shaders, scripts, sonidos, etc.

Cabe destacar en este momento las definiciones de *shader* y *textura* [2].

A la hora de representar o renderizar un objeto en Unity, se realiza mediante *shaders*. Éstas no son más que líneas de código en un lenguaje de sombreado que indica al hardware la forma de representar dicho objeto; este lenguaje de alto nivel específico para ello, se compila de forma independiente en Unity.

Por otro lado, una *textura* es una imagen que sirve para cubrir la superficie de un objeto virtual, tanto en 2D como en 3D.

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.

2.1.1 Funcionamiento general de una aplicación en Unity3D

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

Un proyecto se puede componer de tantas escenas como sean necesarias, formadas por diversos objetos denominados *GameObjets* [3]. Las escenas son niveles de ejecución independientes entre sí. Cada escena dispone de su propio ambiente, decoración y diseño único. Además, las escenas serán cargadas independientemente y de forma secuencial según el flujo de ejecución de nuestra aplicación. Para cambiar de una escena a otra, basta con que un *script* de la escena actual llame a la siguiente escena, y así sucesivamente.
Esto se realiza de la siguiente forma [4]:

```
Application.LoadLevel("NombreDeLaSiguienteEscena");
```

Por otro lado, los *GameObjets* no son más que contenedores, ya que no realizan ninguna función de manera independiente, necesitando estar compuestos por elementos que realicen acciones; estos elementos son los denominados *Components*. También pueden contener otros *GameObjects* que estarán por debajo en la jerarquía. Éstos serán objetos *hijos* del primero y así sucesivamente.

Los *Components* son las piezas funcionales del proyecto 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 "padre" o "hijo".

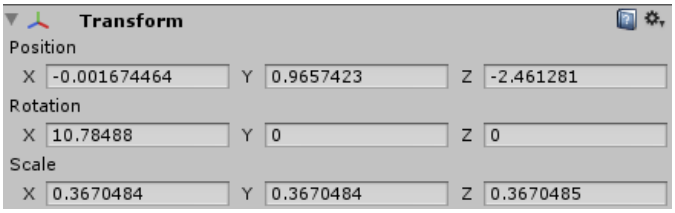


Fig. 2-2 Componente transform en el editor

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.

Por regla general, dentro de un proyecto se pueden crear tantos directorios como se desee. Sin embargo, 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:

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

Fase 2: se realiza lo mismo con 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: por último, se compila 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 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 éste se habilite.

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

Tras 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, está la llamada *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 la función *FixedUpdate*. Si el *frame ratio* es bajo, puede ejecutarse varias veces por *frame*. El intervalo de tiempo que transcurre entre la ejecución de dos *FixedUpdate* consecutivos es constante. Debido a esto, dicha función es idónea para albergar el código referente a las físicas. Las físicas son las fuerzas que se encargan del movimiento de objetos en las escenas. Por tanto, colocando dicho código en la función *FixedUpdate* se permite que los movimientos sean homogéneos y constantes.

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

Por útimo, 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, o algo similar.

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 es 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.

2.1.2 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 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.

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 que usa Unity.

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, Cinema 4D, 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.

2.1.3 Desarrollador Monodevelop

Dentro del paquete de Unity podemos encontrar la herramienta *Monodevelop*, que es un entorno de desarrollo de los lenguajes de programación que soporta Unity, pero que dispone de una completa quía en la que se pueden encontrar todas las funciones, variables o propiedades que ofrece el motor de Unity. En esta guía se puede ver la implementación de todas las funciones para conocer qué es lo que hacen y cuáles son los parámetros de entrada y salida de cada una. En la imagen podemos ver un ejemplo de ello, mostrándonos la implementación de la función *DrawTexture* y los parámetros de entrada.

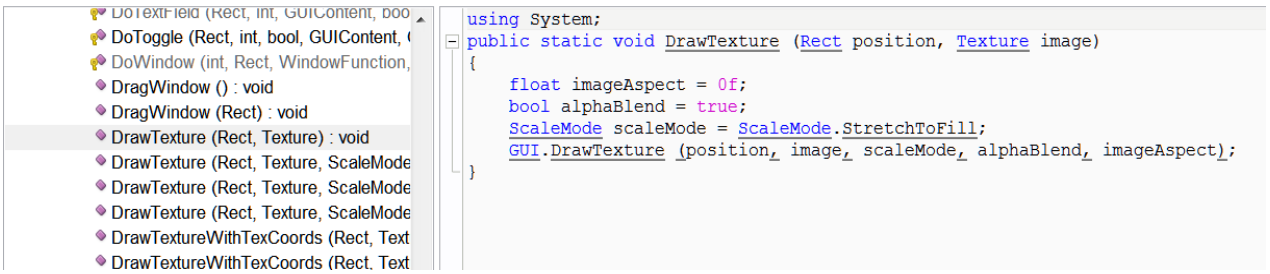


Fig. 2-3 Implementación de la función DrawTexture

Esto es posible hacerlo con cualquier lenguaje aceptado por Unity. También se pueden ver las clases disponibles, con sus variables y sus métodos. En la siguiente imagen podemos ver la clase *WheelCollider* y las variables y métodos que la componen.

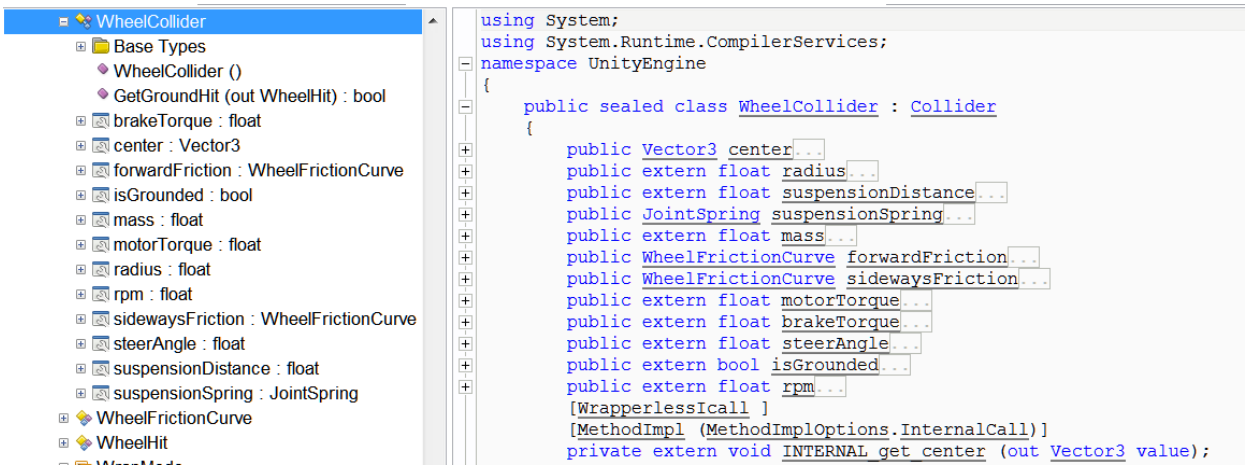


Fig. 2-4 Implementación de la clase WheelCollider

2.2 Herramientas alternativas

2.2.1 Motores gráficos de aplicaciones en 3D

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

2.2.1.1 Shiva 3D

Shiva 3D [5] se caracteriza por ser un motor que utiliza *LUA Script*, un tipo de script de mayor rendimiento, ideal para juegos que requieran mayor calidad gráfica. Es un motor más rápido que el de Unity, y además, consume menos memoria. En contrapartida, es un IDE más compleja y menos atractiva e interactiva que Unity, por lo que la curva de aprendizaje es más lenta. La licencia ilimitada de Shiva ronda los 1000$.

Fig. 2-5 Editor del IDE de Shiva 3D

2.2.1.2 Torque 3D

Si hacemos la comparación con Torque 3D [6], diremos que es de código abierto. Éste tiene mayores capacidades a la hora de renderizar que Unity, ya que 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++.

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

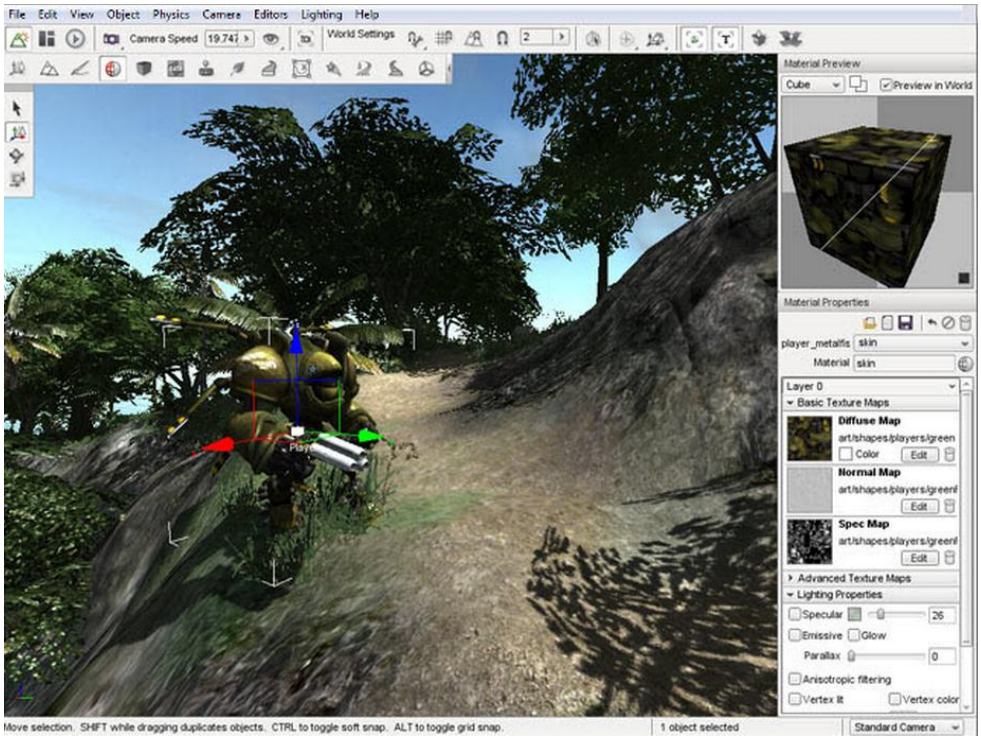


Fig. 2-6 Editor del IDE de Torque 3D

2.2.2 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, lo que era otra opción a tener en cuenta al realizar este proyecto.

Algunos de estos son: *Rfactor, Car-X Technologies, VDrift o OpenDs.*

2.2.2.1 Rfactor

Rfactor [7] 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 a su vez 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, pero no dispone de la flexibilidad suficiente para realizar nuestro proyecto.

2.2.2.2 Car-X

Car-X [8] 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 por ejemplo existen intersecciones o semáforos. 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.

Fig. 2-7 Sistema Car-X integrado en Unity3D

2.2.2.3 VDrift

VDrift [9] es una plataforma de simulación de conducción de código abierto basado principalmente en carreras de *drift*. El motor de físicas está recientemente implementado y está inspirado en el motor de físicas llamado *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.

Fig. 2-8 Simulación en el entorno VDrift

2.2.2.4 OpenDS

OpenDS [10] es un software de simulación de conducción de código abierto. Está orientado a fines de investigación y está completamente desarrollado en Java, basado en el *framework* de *jMonkey* Engine. Está enfocado para todas las plataformas OpenGL 2 y 3 que dispongan 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.

Fig. 2-9 Simulación mediante el software de OpenDS

3. Descripción de la aplicación

En este apartado vamos a describir el funcionamiento del simulador en su totalidad, desde la estructura que lo compone hasta la descripción detallada de los *scripts* que permiten su funcionamiento.

3.1 Estructura

En cuanto a la estructura del proyecto podemos hacer referencia a dos distinciones: la estructura temporal que sigue la aplicación cuando se ejecuta y la estructura en la que se dividen los archivos que lo componen.

3.1.1 Estructura temporal

Como ya se explicó anteriormente, todo proyecto está formado por escenas que, ejecutadas en un orden secuencial determinado, dan forma a la aplicación.

Este proyecto se divide en dos escenas claramente distinguidas: el menú principal que permite configurar la simulación y la escena donde se lleva a cabo la simulación de conducción.

Fig. 3-1 Estructura dividida en dos escenas

En primer lugar, al abrir la aplicación, se ejecutará la escena *Menu.* Tras configurar los parámetros deseados, el script del menú se encargará de dar paso a la siguiente escena al pulsar el botón de *Iniciar Simulación*.

La segunda escena, *CompleteScene*, contiene todos los elementos que permiten la simulación propiamente dicha.

Nuestro proyecto parte de esta escena, que fue descargada de la página de Unity3D como tutorial. La escena contenía todo a lo que el entorno se refiere y el vehículo del usuario junto con su script para controlarlo. Es decir, partíamos de un vehículo que el usuario podía controlar por todo el terreno.

Esta escena está formada por todos los *GameObjects* que se ven en la imagen, cada uno tiene su función en la escena.

Fig. 3-2 Objetos que forman la escena principal

Los dos elementos activos de esta escena son el objeto *Car, es decir,* es el coche que el usuario puede conducir; y el objeto *GT4*, que es vehículo que dispondrá de inteligencia artificial para circular por todo el trazado.

Al objeto *Car* se le han añadido algunos scripts aparte de los que llevaba por defecto: *Car, Sound Controller y Lightmapper Object UV* tal como se ve en la siguiente imagen. El script *Car* es el que permite que el vehículo sea controlado, definiendo todos los parámetros necesarios del vehículo: relaciones de marchas, suspensiones, velocidad máxima y máximo giro entre otras.

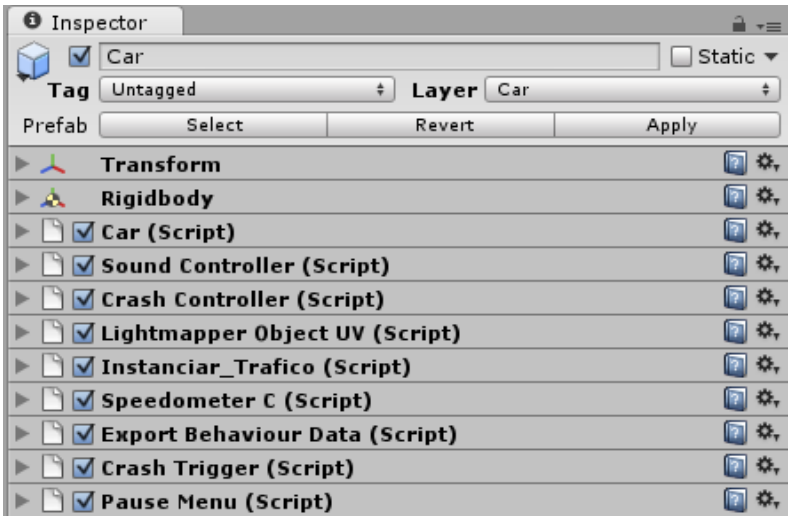


Fig. 3-3 Scripts asociados al objeto Car

Por último, esta escena tiene un menú de *Pause* que permite reiniciar la simulación, por lo que se volvería a cargar esta escena; o volver al menú principal en cuyo caso se cargará la escena *Menu*. También dispone de la opción *Salir* que terminará con la ejecución de la aplicación.

3.1.2 Estructura de los ficheros

En cuanto a la forma de organizar los ficheros, es simple: se dividen por el tipo de archivo que contiene. También hay que tener en cuenta los directorios de carácter general comentados en el punto 2.1.1.

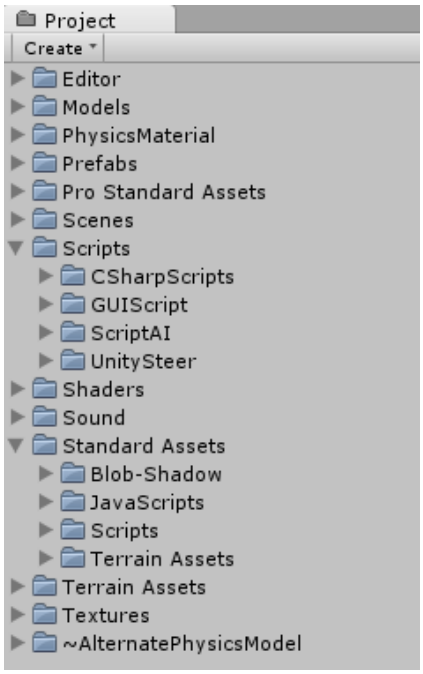


Fig. 3-4 Estructura de los ficheros en el proyecto

En la imagen anterior podemos ver la estructura que sigue nuestro proyecto. El directorio *Models* contiene los modelos 3D de los vehículos y de todos los elementos visibles que forman el mapa. Dentro de *Prefabs* se encuentran los objetos prefabricados, es decir, objetos que quizás contengan *scripts* y que pueden ser instanciados en la escena en cualquier momento. Por ejemplo, en nuestro caso, contiene el *prefab GT4_opposite,* que será instanciado en la escena tantas veces como vehículos en sentido contrario circulen.

Por otro lado, en la carpeta *Scripts*, están la mayoría de los scripts, de forma más concreta, aquéllos que serán compilados en la fase 3. Además, en *Standard Assets* se encuentran los *scripts* codificados en *JavaScript*, debido a que nos resultaba necesario que éstos fueran compilados en la fase 1. El directorio *ScriptsAI* contiene todos los *scripts* realizados tanto para la inteligencia artificial como para el comportamiento del coche de referencia.

También encontramos los directorios *Scenes, Shaders, Sound y Textures* que contienen lo anteriormente dicho.

3.2 *Scripts*

En este punto vamos a tratar de describir los *scripts* más relevantes para el funcionamiento de la aplicación. Para ello, los vamos a dividir en cinco subapartados: inteligencia artificial, generación de tráfico, control de frenadas, exportación de datos de simulación e interfaz gráfica de usuario.

3.2.1 Scripts de inteligencia artificial.

Para el objetivo de nuestro proyecto era necesario disponer de vehículos dotados de inteligencia artificial. Para ello, el primer paso fue conseguir que dicho vehículo fuera capaz de dar vueltas al trazado de forma autónoma y homogénea.

Para ello surgieron varias posibles soluciones. La primera de ellas fue realizar la trayectoria que debía seguir el coche mediante puntos de referencia o *waypoints*. El vehículo recorría secuencialmente un *array* de posiciones en el espacio. Tras implementar dicha solución, el comportamiento del vehículo no era el esperado, ya que, aunque seguía la ruta deseada, no lo hacía de forma natural ; a la hora de girar en las curvas no lo hacía de forma continua sino que hacía varios cambios de dirección antes de llegar a la posición deseada. Debido a esto, fue necesario buscar otro método que solucionase nuestro problema.

Otra solución fue realizarlo mediante una librería implementada en C# llamada *iTween* [11], que dispone de funciones que permiten movimientos continuos a través de una trayectoria dada. Esta propuesta solucionaba el conflicto de los giros en curvas, pero esta librería estaba pensada para el movimiento de objetos sin orientación. Por ello, aunque el vehículo seguía la ruta correctamente, no era capaz de orientarse frontalmente con la trayectoria. Consecuentemente tampoco se adecuaba a nuestras necesidades.

Por último, la propuesta definitiva que permitió el correcto funcionamiento fue *Unity Steer*. *Unity Steer* es un conjunto de librerías implementadas por Arges Systems [12] que permite dotar a los objetos de inteligencia artificial. Dichas librerías codificadas en C# permiten establecer dos comportamientos principales mediante dos clases: *Biped,* que está orientado a movimientos para bípedos; y *Autonomous Vehicle* cuya finalidad es ofrecer inteligencia a automóviles u otros objetos similares.

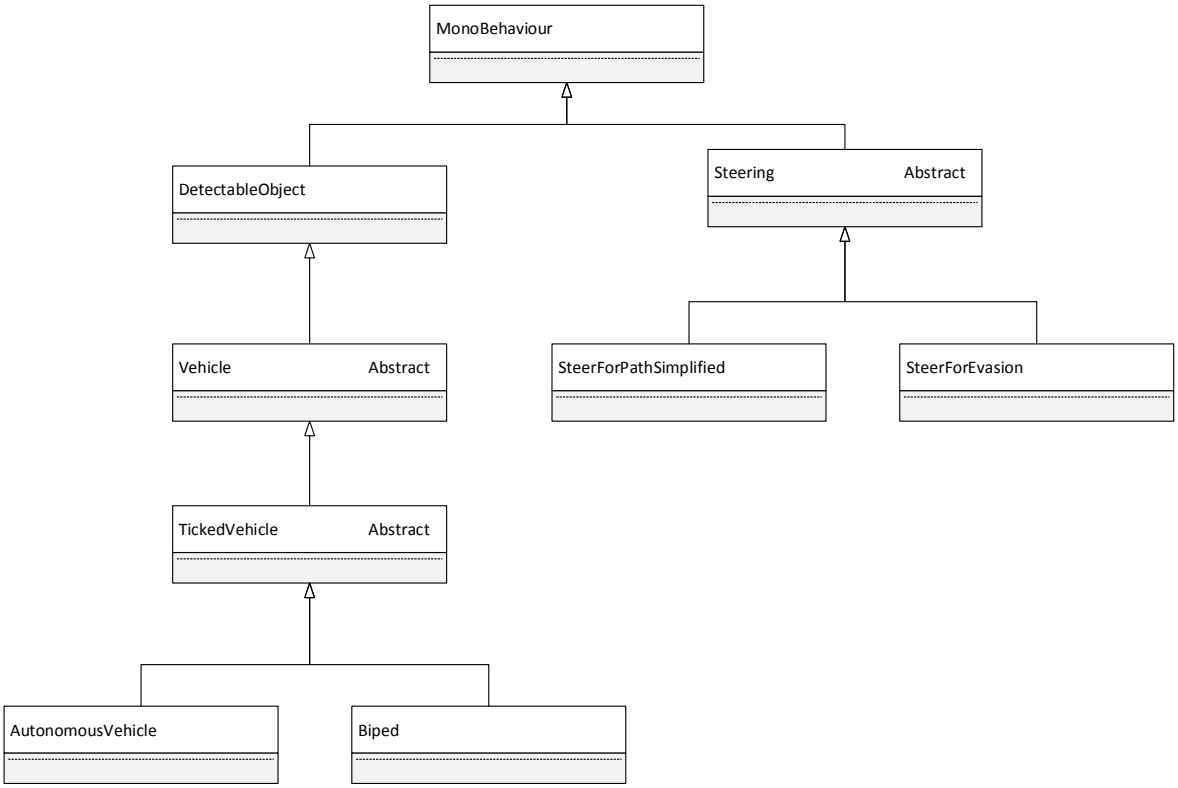


Fig. 3-5 Herencia de clases de las librerías Unity Steer

Como se puede ver, tanto *Biped* como *Autonomous Vehicle* heredan de una clase abstracta llama *Ticked Vehicle*. Ésta última también hereda de la clase abstracta *Vehicle.*

Las librerías *Unity Steer* también poseen una clase abstracta llamada *Steering*, de la cual heredan multitud de clases que implementan diferentes comportamientos adicionales. Algunos ejemplos de estas clases son:

- *SteerForEvasion*
- *SteerForTarget*
- *SteerForPath*

La primera de éstas permitirá al objeto huir de otro objeto determinado, permitiendo configurar la distancia mínima de seguridad. *SteerForTarget* permite al objeto ir hacia un objetivo determinado. Por otra parte, la clase *SteerForPath* permite - a partir de una ruta dada - hacer que el objeto siga esa trayectoria.

Estas clases han de ser añadidas al objeto que queramos dotar de inteligencia. Dicho objeto debe tener una de las dos clases principales adheridas, *Biped* o *Autonomous Vehicle*, que podrán hacer uso de estos comportamientos *Steering*.

En nuestro caso, el vehículo referente está dotado con las clases *Autonomous Vehicle y SteerForPathSimplified,* una versión simplificada de *SteerForPath*. Ambos, junto con un *script* adicional que sirve de unión entre ellos, harán posible que el coche siga la ruta deseada, tal y como pretendíamos. Este *script* llamado *RutaSteer,* se encarga de crear la ruta de *waypoints* y pasársela a la clase *SteerForPathSimplified* que se encargará de guiar al vehículo.

Esta ruta de *waypoints* ha sido creada a mano, es decir, se han creado alrededor de noventa *GameObjects* y han sido colocados minuciosamente en la escena para conseguir que la ruta se ajuste en todo momento a la carretera. Todos estos *waypoints* son almacenados en un *GameObject* llamado *Waypoints* que más tarde *RutaSteer* se encargará de procesar.

Fig. 3-6 Ruta de waypoints del vehículo de referencia

Como se observa en la imagen, la línea verde es el trazado que seguirá el vehículo de referencia; es la unión de todos los *waypoints* situados en la carretera.

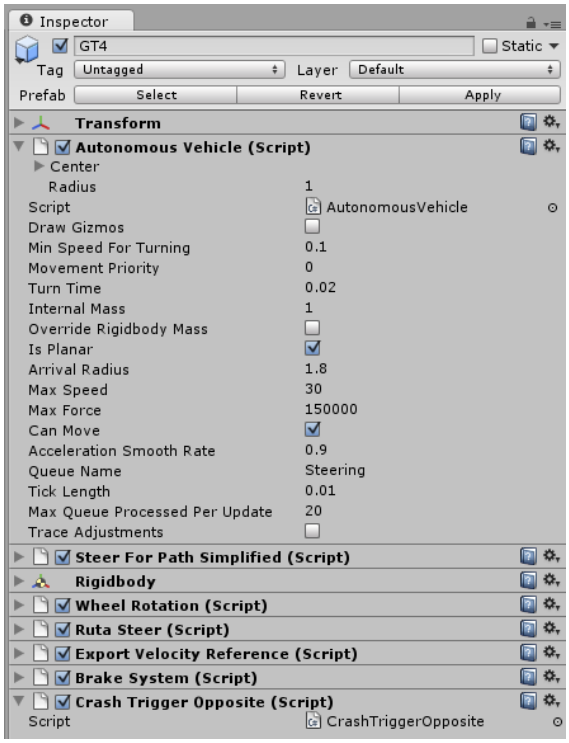


Fig. 3-7 Script AutonomousVehicle en el Inspector

Como se puede ver en la imagen superior, el *script Autonomous Vehicle* dispone de algunos parámetros que son configurables desde el *Inspector* de Unity. Por ejemplo, la opción *Arrival Radius*, es el radio que considera cuándo se ha alcanzado un *waypoint* y cuándo no. Otra de las opciones configurables en este *script* es el tiempo de giro, la velocidad máxima o la fuerza máxima que puede tener el *Auntonomous Vehicle.* Esta fuerza está relacionada con el cálculo de fuerzas mencionado anteriormente, por eso, este valor ha de ser muy alto para que el vehículo pueda alcanzar la velocidad deseada. El parámetro *Override Rigidbody Mass* se podrá activar para que *Unity Steer* no tenga en cuenta el peso del vehículo, sino el peso indicado en la variable *Internal Mass*.

Como modo de aclaración, es importante definir qué es el *Rigidbody.* Cuando queramos que un objeto cualquiera se ciña a las normas de la gravedad, es decir, se le apliquen las fuerzas físicas correspondientes a su peso, le añadiremos a dicho objeto el componente *Rigidbody.* Éste se compone de varios parámetros configurables como por ejemplo, su peso o su índice de rozamiento. También se le pueden aplicar ciertas restricciones, para que no experimente fuerzas en alguna de las componentes de rotación o posición, tal como se muestra en la figura inferior.

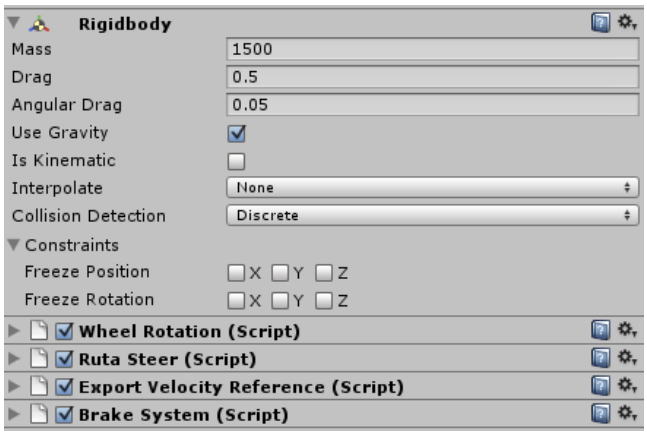
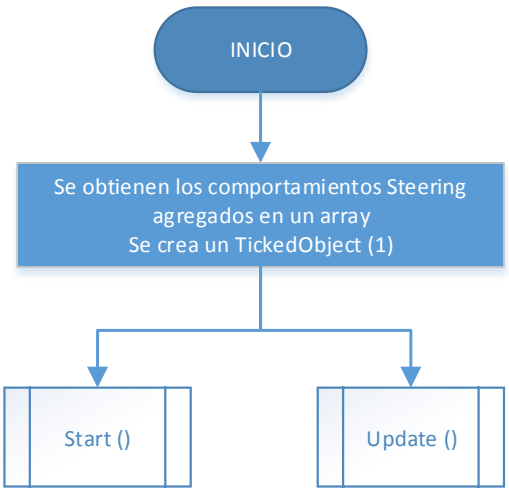
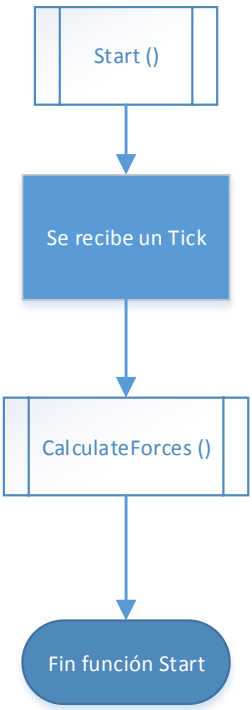
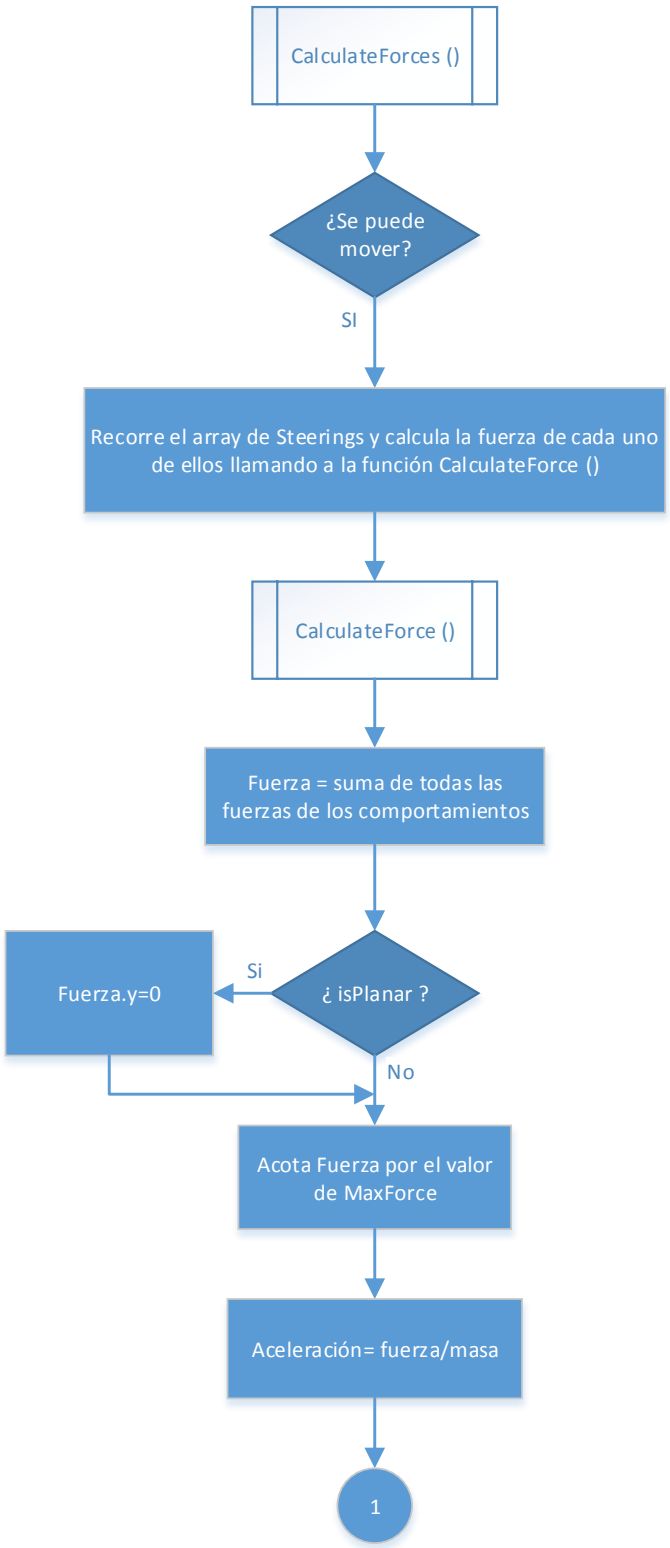


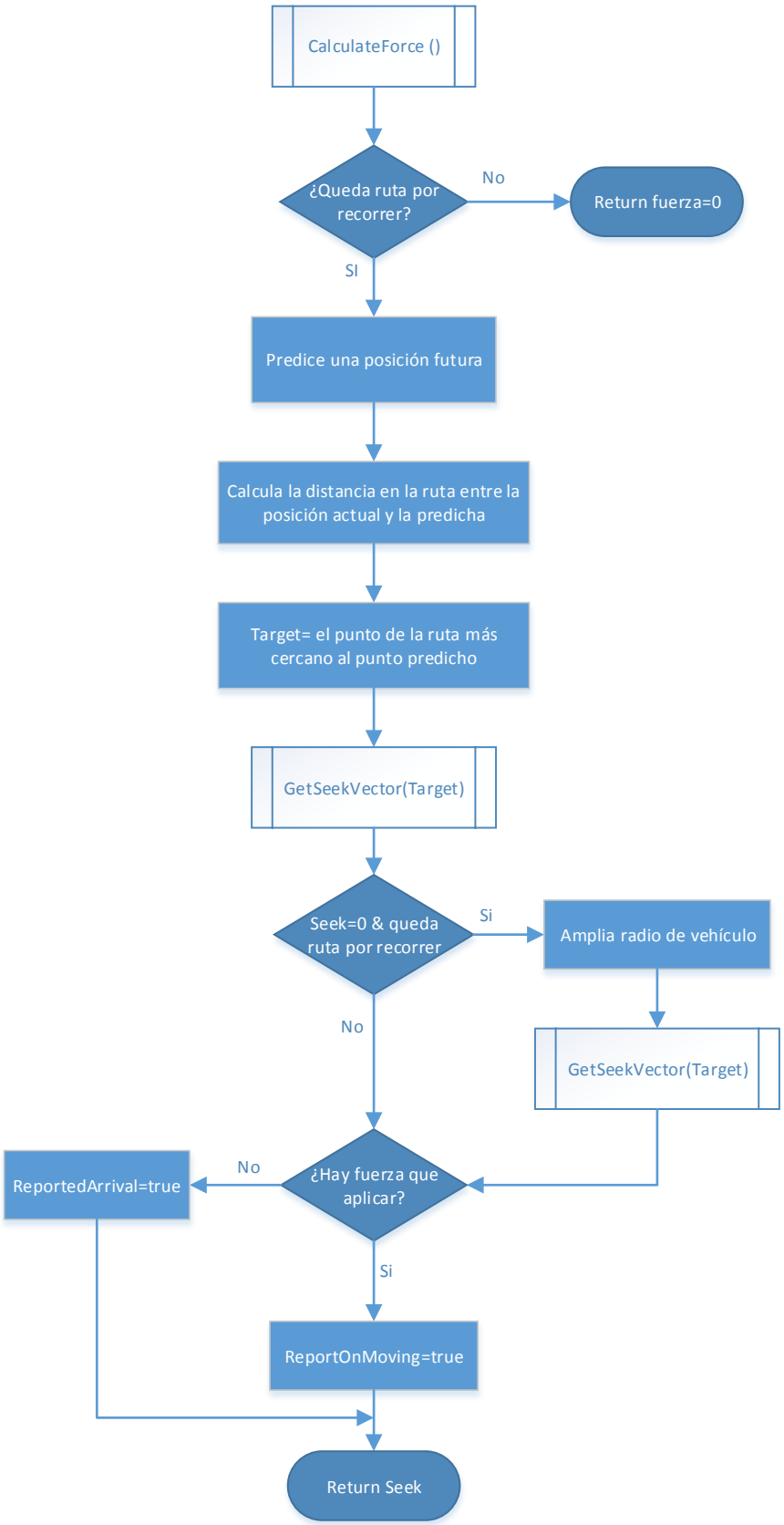
Fig. 3.8 Componente Rigidbody del vehículo

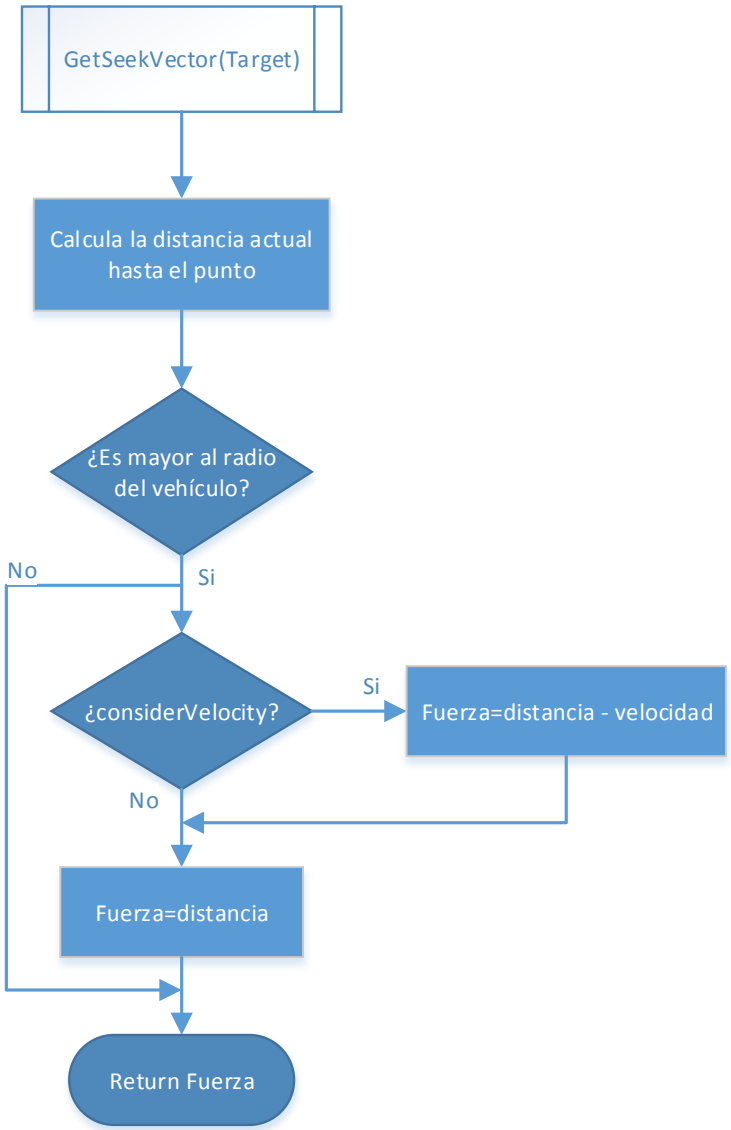
Para profundizar más en *Unity Steer,* diremos que se trata de un sistema basado en fuerzas [13]. Es un proceso cíclico que predice donde va a estar en instantes de tiempo muy cercanos y calcula la fuerza y la dirección con la que ha de moverse para llegar a su objetivo. A continuación, se describirá el proceso que realiza *Unity Steer* sobre nuestro vehículo mediante un diagrama de flujo.

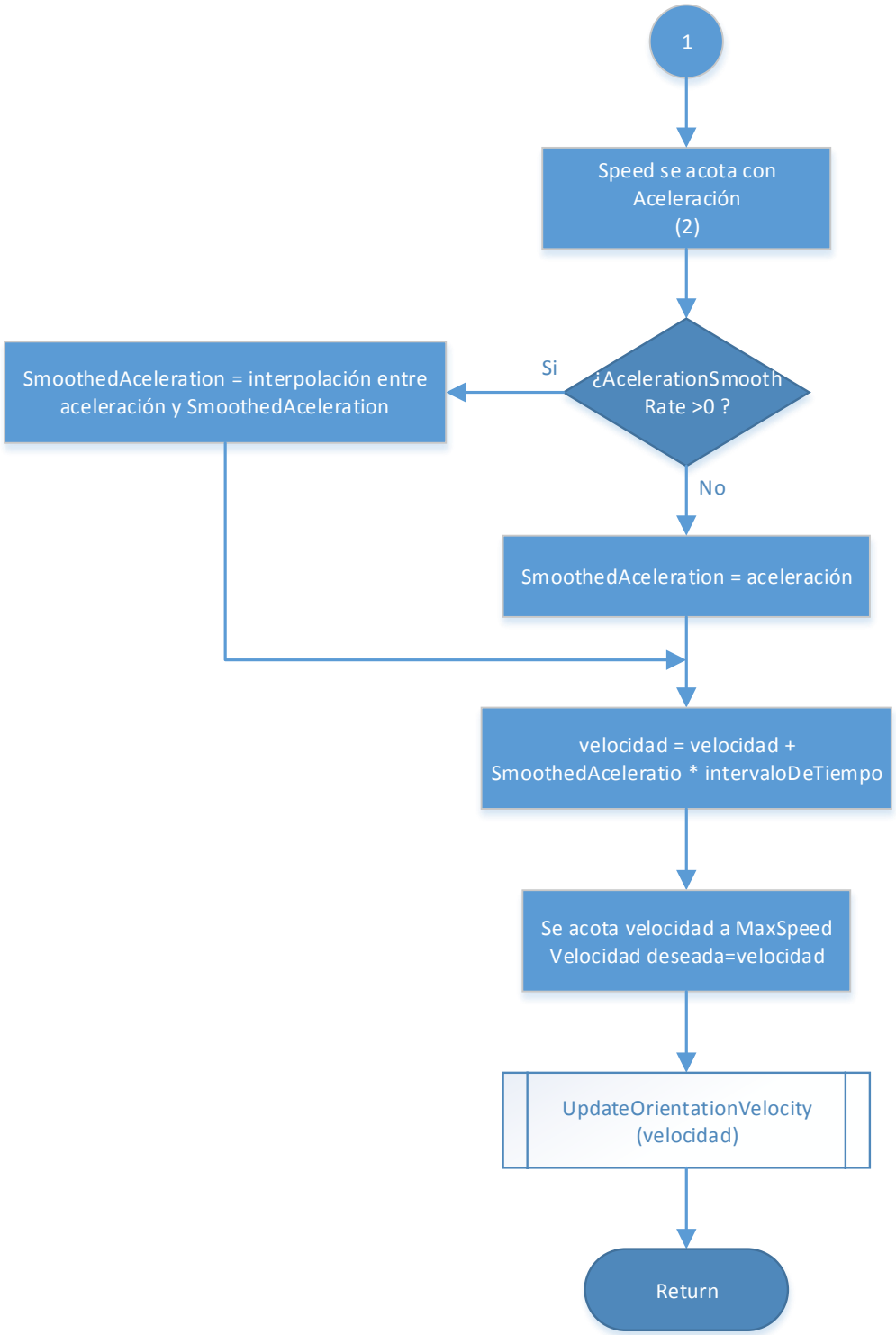


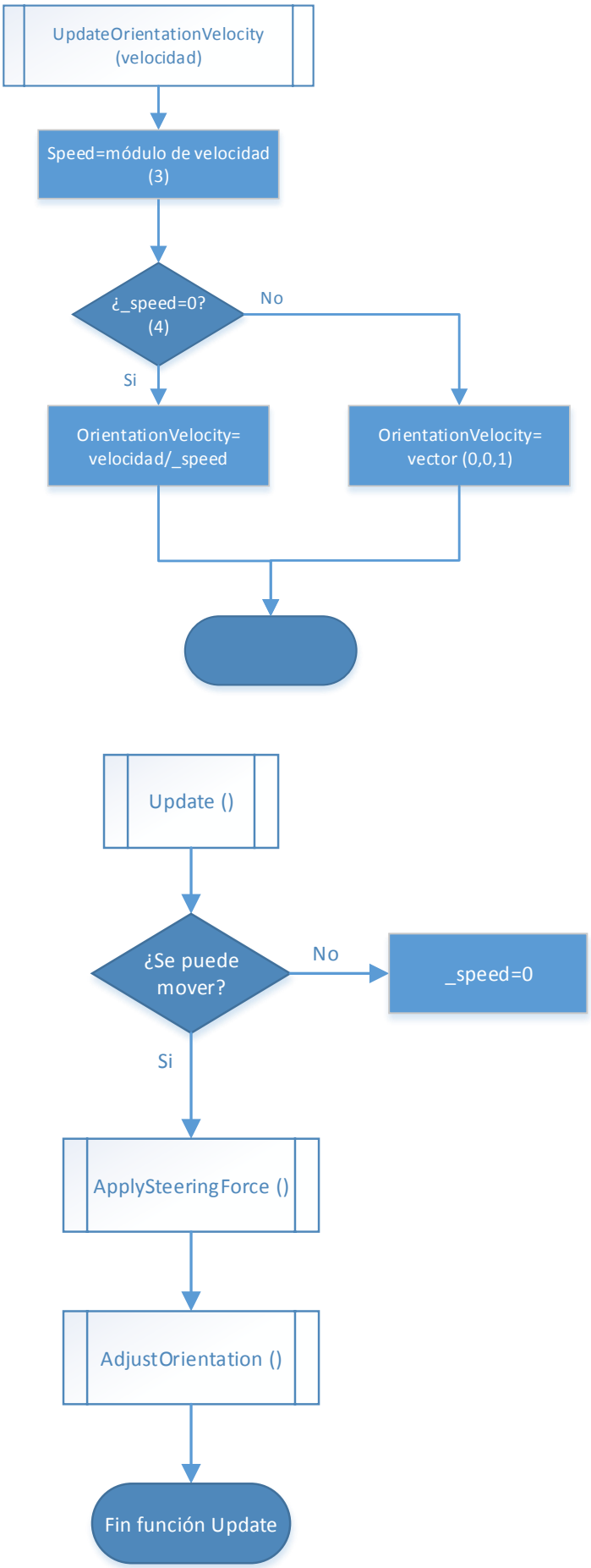

Start ()
Se recibe un Tick
CalculateForces ()
Fin función Start

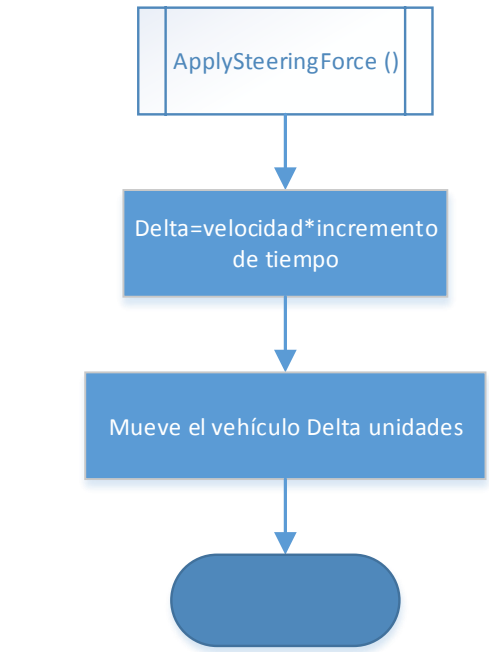

CalculateForces ()
¿Se puede mover?
SI
Recorre el array de Steerings y calcula la fuerza de cada uno de ellos llamando a la función CalculateForce ()
CalculateForce ()
Fuerza = suma de todas las fuerzas de los comportamientos
Fuerza.y=0
Si
¿ isPlanar ?
No
Acota Fuerza por el valor de MaxForce
Aceleración= fuerza/masa
1

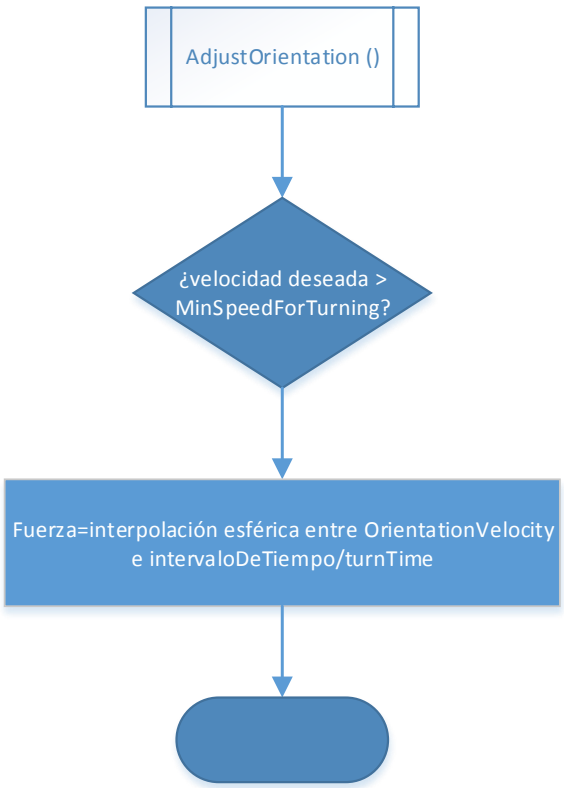

CalculateForce ()
¿Queda ruta por recorrer?
No
Return fuerza=0
SI
Predice una posición futura
Calcula la distancia en la ruta entre la posición actual y la predicha
Target= el punto de la ruta más cercano al punto predicho
GetSeekVector(Target)
Seek=0 & queda ruta por recorrer
Si
Amplia radio de vehículo
No
GetSeekVector(Target)
ReportedArrival=true
No
¿Hay fuerza que aplicar?
Si
ReportOnMoving=true
Return Seek


GetSeekVector(Target)
Calcula la distancia actual hasta el punto
¿Es mayor al radio del vehículo?
No
Si
¿considerVelocity?
Si
Fuerza=distancia - velocidad
No
Fuerza=distancia
Return Fuerza


1
Speed se acota con Aceleración (2)
¿AcelerationSmooth Rate >0 ?
Si
SmoothedAceleration = interpolación entre aceleración y SmoothedAceleration
No
SmoothedAceleration = aceleración
velocidad = velocidad + SmoothedAceleratio * intervaloDeTiempo
Se acota velocidad a MaxSpeed Velocidad deseada=velocidad
UpdateOrientationVelocity (velocidad)
Return


UpdateOrientationVelocity (velocidad)
Speed=módulo de velocidad (3)
¿_speed=0? (4)
No
Si
OrientationVelocity= velocidad/_speed
OrientationVelocity= vector (0,0,1)
Update ()
¿Se puede mover?
No
_speed=0
Si
ApplySteeringForce ()
AdjustOrientation ()
Fin función Update


ApplySteeringForce ()
Delta=velocidad*incremento de tiempo
Mueve el vehículo Delta unidades


AdjustOrientation ()
¿velocidad deseada > MinSpeedForTurning?
Fuerza=interpolación esférica entre OrientationVelocity e intervaloDeTiempo/turnTime

Tras el conjunto de flujogramas que explican el complejo funcionamiento de *Unity Steer*, cabe detallar algunos aspectos:

En primer lugar, al inicio de la ejecución se crea un *TickedObject (1),* un objeto que cada periodo de tiempo *t* recibe un evento. Este evento hará que se ejecute la función a la que se referenció al crear dicho objeto. Cada evento llegará como mínimo una vez por *frame*, por lo que la función *OnUpdateSteering* se ejecutará constantemente. Como vemos en el esquema, el primer evento o *Tick,* llegará al comenzar la función Start. La siguiente línea de código es la que permite lo comentado.

TickedObject = new TickedObject(OnUpdateSteering);

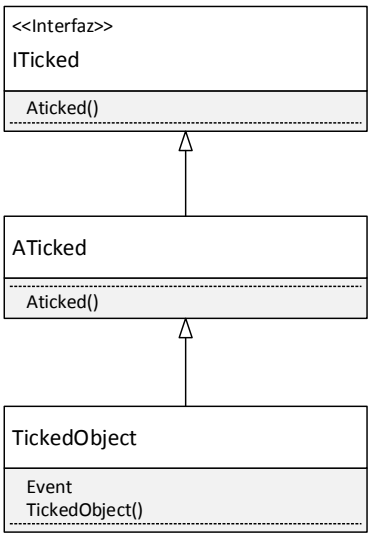


Fig. 3-9 Herencia de clases de TickedObject

La siguiente reseña en el flujograma la número (2), es la variable speed; diremos que se trata de una variable local de la función *CalculateForce* de la clase *SterrForPathSimplified*, ya que puede dar lugar a confusión con otras similares. Es una variable de tipo *float*. Por otro lado, *Speed* (3), es una propiedad disponible en la clase *AutonomousVehicle.* Por último, *_speed* (4), también pertenece a la clase anterior, pero en este caso es una variable general de tipo *float*.

Como vemos, *Unity Steer* es un sistema cuanto menos complejo, que anida en su ejecución gran cantidad de funciones, propiedades y clases.

3.2.2 Scripts de generación de tráfico

Una vez que conocemos cómo funciona *Unity Steer* y cómo es capaz de conducir de forma autónoma, vamos a ver cómo se genera el tráfico que circula en sentido contrario. En la siguiente imagen se muestra el aspecto de uno de los coches que conforman el tráfico en sentido contrario.

Fig. 3-10 Vehículo generado por el script de tráfico

En primer lugar, creamos un *prefab* con el mismo comportamiento básico que el vehículo de referencia llamado *GT4_opposite*. También es necesario crear la ruta en el sentido contrario, por lo que se crea otro contenedor de *waypoints* distinto.

Debido a que esta ruta es diferente a la definida para el vehículo referente, también es necesario que exista un *script* distinto al utilizado con anterioridad. Éste es llamado *RutaSteer_opposite* y a grandes rasgos, es igual al anterior, con la diferencia que origina una ruta para circular en el otro sentido.

Para generar el tráfico en tiempo de simulación, se procede a instanciar el *prefab GT4_opposite* en la escena. Tenemos que conseguir que el tráfico generado esté equidistante por todo el terreno, es decir, que no se encuentren todos vehículos demasiado cerca, sino que cada “x” tiempo nos encontremos con un vehículo.

Para solucionar eso, se han colocado de forma estratégica siete puntos de referencia en la escena, de forma que, si en un instante *t* se instancia un vehículo en cada uno de estos puntos, habremos generado un tráfico en toda la escena. Estos puntos de referencia se han denominado *PuntosDeSalida* en nuestro proyecto.

La clase encargada de generar el tráfico se denomina *Instanciar_Trafico* y permite escoger entre cuatro niveles de densidad de tráfico antes de realizar la simulación. Dicho nivel se cuantifica mediante un entero.

En caso de ser 0, no se instanciará ningún elemento en la escenaSi el nivel de densidad es 1, se generará lo comentado anteriormente: un vehículo por cada punto de salida en un instante determinado *t*. Por otra parte, si el tráfico es de nivel 2, además de los siete vehículos generados en el instante *t*, se volverán a instanciar otros siete vehículos más, en el instante *2t*. Puesto que los generados anteriormente ya habrán circulado "x" distancia, se podrán instanciar en los mismos puntos de salida iniciales. Para el caso de nivel 3, ocurrirá lo mismo pero en el instante *3t*.

De esta forma se habrá logrado obtener un tráfico de densidad deseada y uniformemente distribuido por el circuito.

Fig. 3-11 Ruta de waypoints en sentido contrario

3.2.3 Script de control de frenada

El sistema de frenado automático que compone a el de referencia, está formado únicamente por un *script* desarrollado en C# denominado *BrakeSystem*.

Este *script* permite disponer de dos comportamientos diferentes de frenado: de forma aleatoria y mediante distancia de seguridad. El primero de los métodos permite al coche frenar mediante una variable aleatoria distribuida uniformemente de media *t* segundos.

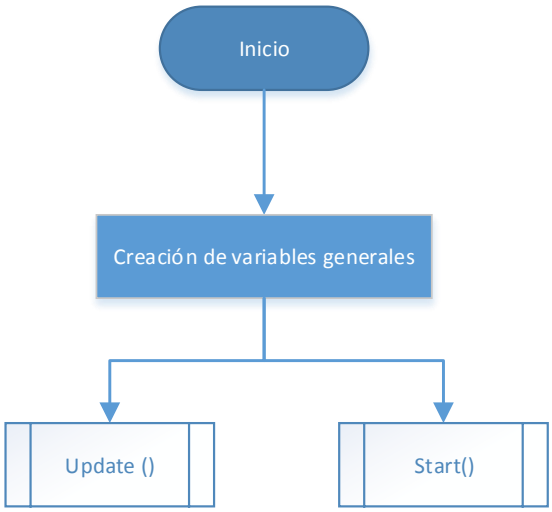
El segundo de ellos, es un sistema reactivo, es decir, ofrece una respuesta cuando percibe que algo ha cambiado. En otras palabras, si el vehículo conducido por el usuario no respeta la distancia con el coche de referencia, el que va por delante lo percibirá y reaccionará. Dicha reacción se corresponderá con un frenazo en un 40% de las veces que se supere la distancia de seguridad.

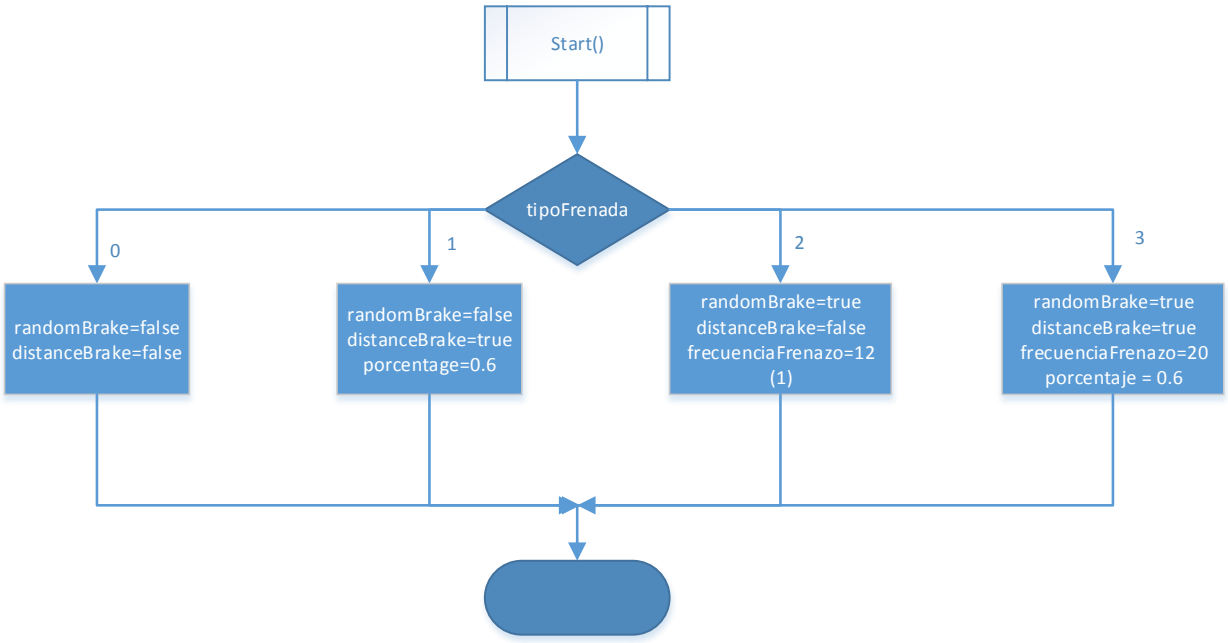
Con la combinación de estos dos métodos podemos conseguir cuatro situaciones diferentes que podrán ser configuradas en el menú principal. El primero de los comportamientos a elegir permite anular dicho sistema de frenado, por lo que el vehículo no se detendrá nunca.

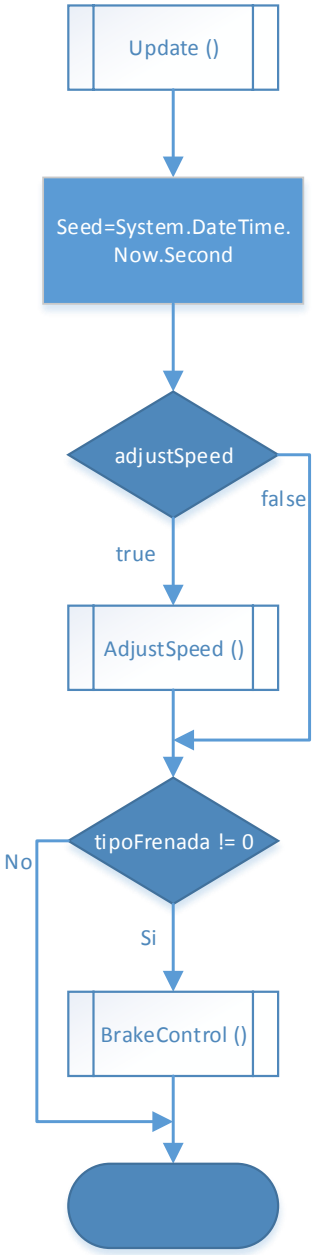
Otras dos opciones, son las nombradas anteriormente: de forma aleatoria y de forma reactiva.

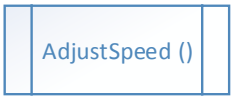
Por último, si combinamos los dos comportamientos anteriores podremos conseguir que el vehículo disponga de los dos sistemas de frenado automático de forma simultánea.

Lo siguiente será describir con más detalle el sistema a través de un diagrama de flujo.

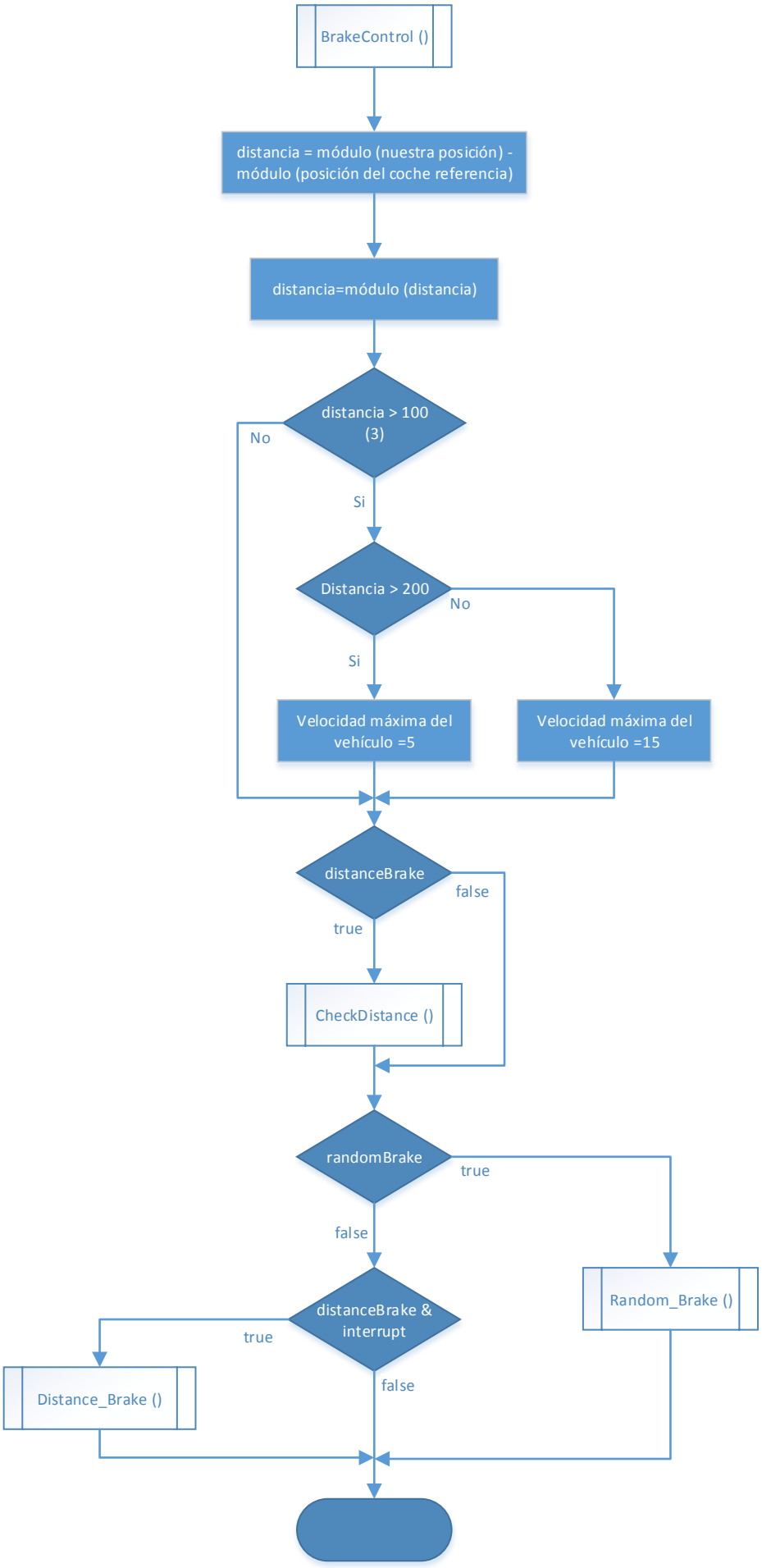

Inicio
Creación de variables generales
Update ()
Start()

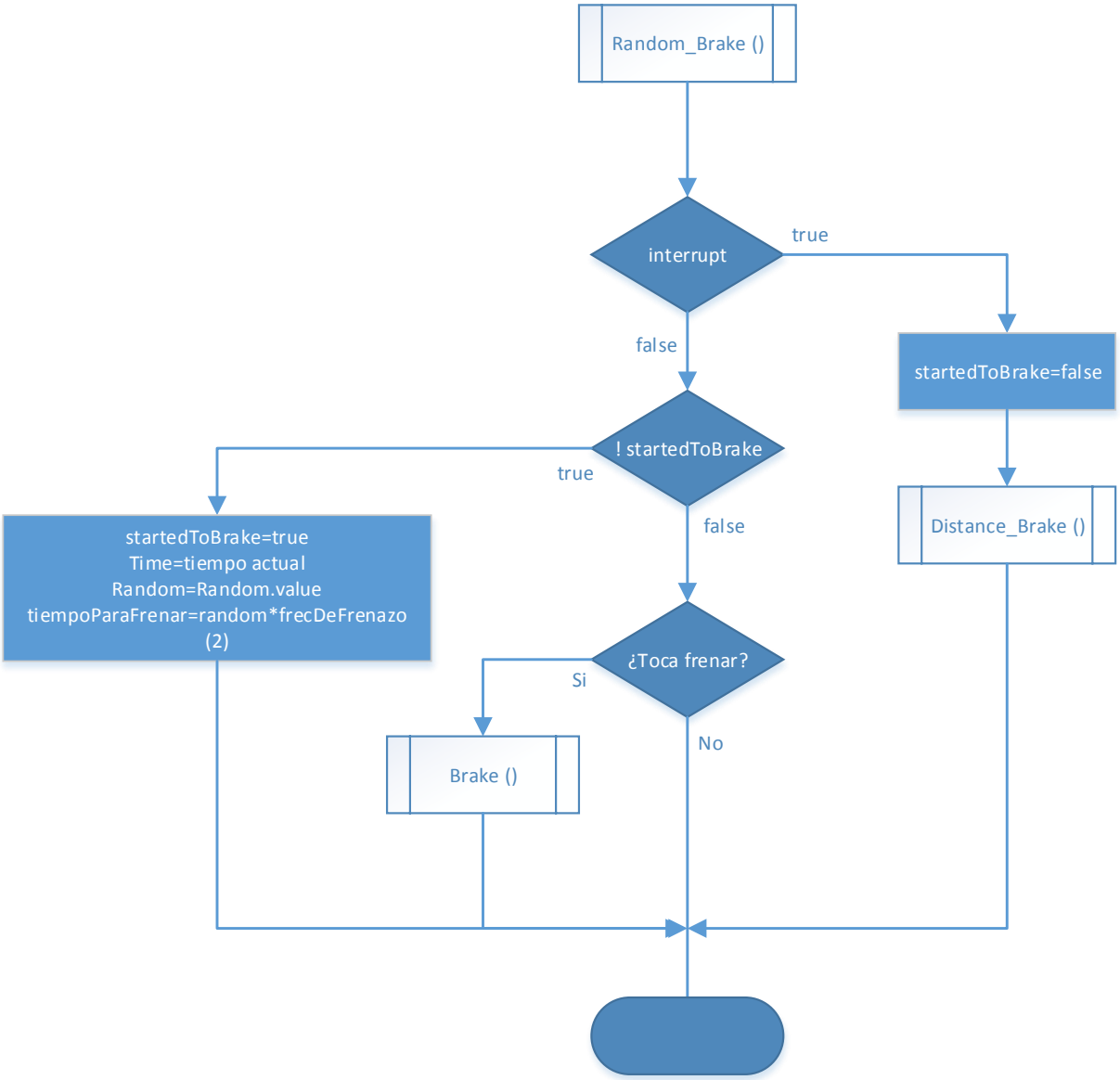

Start()
tipoFrenada
0
1
2
3
randomBrake=false
distanceBrake=false
randomBrake=false
distanceBrake=true
porcentage=0.6
randomBrake=true
distanceBrake=false
frecuenciaFrenazo=12
(1)
randomBrake=true
distanceBrake=true
frecuenciaFrenazo=20
porcentaje = 0.6

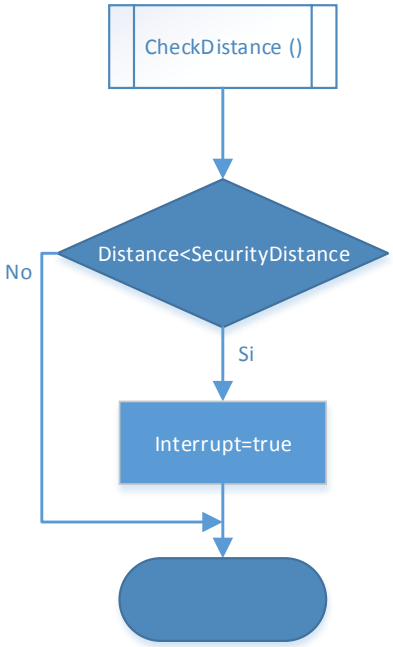


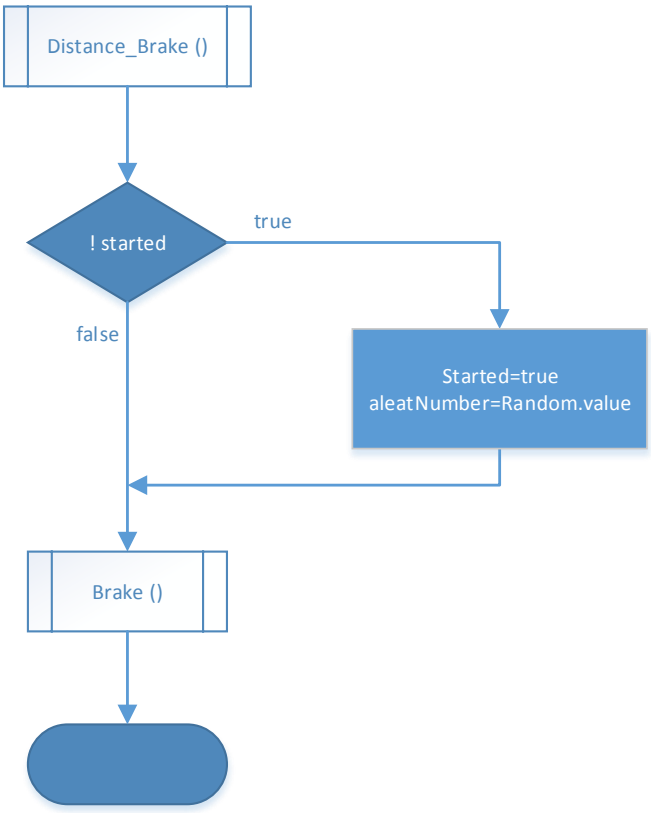


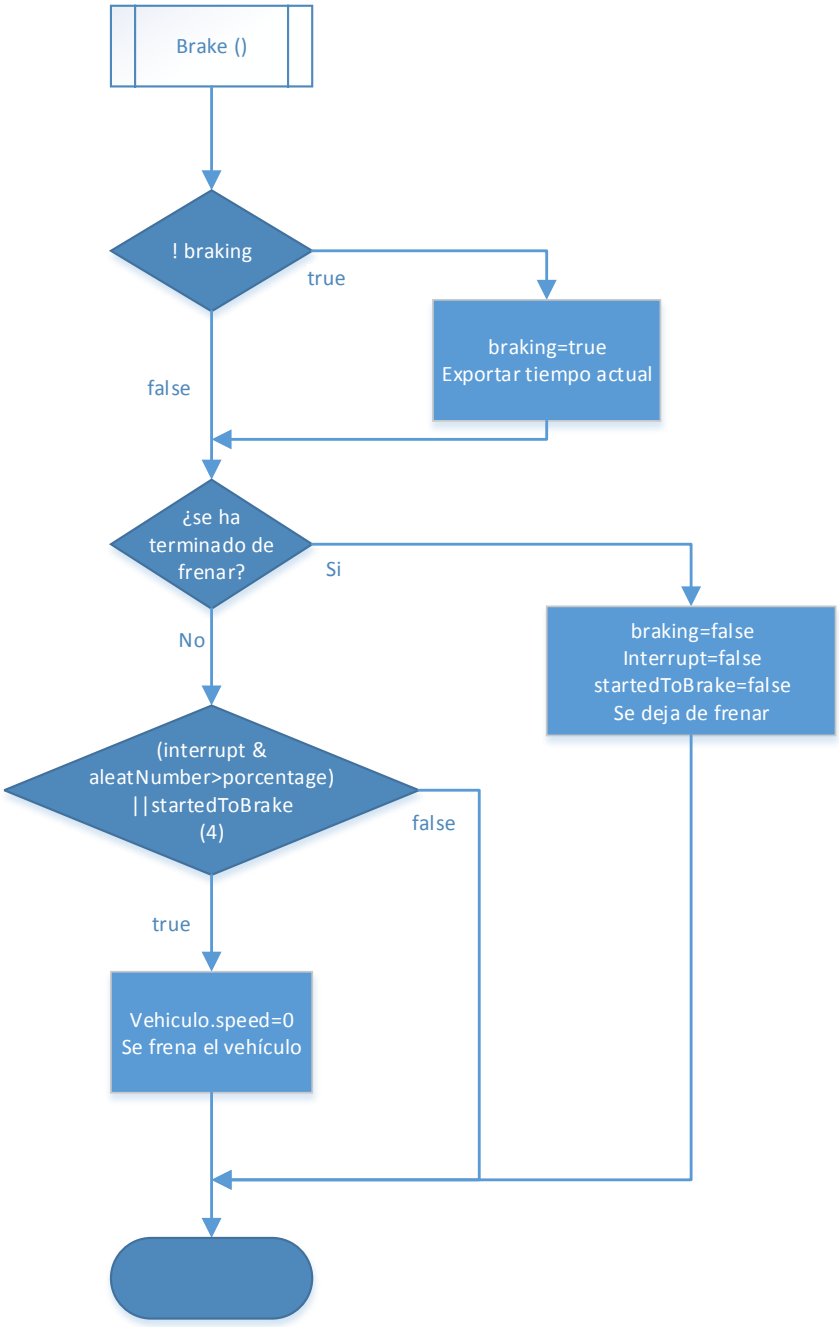
La función *AdjustSpeed* se ejecutará siempre que el vehículo ha terminado de frenar. Simplemente se encarga de proporcionar una aceleración progresiva para que el coche del usuario no quede rezagado tras la frenada. Cuando se alcanza la velocidad máxima, deja de ejecutarse.


BrakeControl ()
distancia = módulo (nuestra posición) - módulo (posición del coche referencia)
distancia=módulo (distancia)
distancia > 100 (3)
No
Si
Distancia > 200
No
Si
Velocidad máxima del vehículo =5
Velocidad máxima del vehículo =15
distanceBrake
false
true
CheckDistance ()
randomBrake
true
false
Random_Brake ()
distanceBrake & interrupt
true
false
Distance_Brake ()


Random_Brake ()
interrupt
true
false
startedToBrake=false
! startedToBrake
true
false
Distance_Brake ()
startedToBrake=true
Time=tiempo actual
Random=Random.value
tiempoParaFrenar=random*frecDeFrenazo
(2)
¿Toca frenar?
Si
No
Brake ()


CheckDistance ()
Distance<SecurityDistance
No
Si
Interrupt=true


Distance_Brake ()
! started
true
false
Started=true
aleatNumber=Random.value
Brake ()



De esta forma queda descrito el funcionamiento del sistema de frenado. A continuación vamos a aclarar las cuatro reseñas expuestas.

Como se puede observar, el sistema de frenado por distancia está dotado de prioridad ante el de frenada aleatoria. Si se está frenando de forma aleatoria y en cualquier momento se supera la distancia de seguridad, se iniciará el proceso para frenar por distancia.

En primer lugar, la variable *frecuenciaFrenazo* (1) no es más que un *float* que indica la media de tiempo con la que se frenará de forma aleatoria. Para el caso 2, en el que únicamente se utiliza dicho sistema, la media será de 12 segundos; mientras que para el caso 3 será de 20 segundos de media puesto que conviven ambos sistemas conjuntamente. En (2) se pondera la frecuencia por el número aleatorio uniformemente distribuido.

Por otro lado, para evitar que el coche de referencia se aleje demasiado del vehículo pilotado, se implementa un sistema que lo evita (3). Dicha implementación reducirá la velocidad del vehículo referente en caso de estar alejado.

Po último, como ya se comentó, sólo se frenará un porcentaje de las veces que se supere la distancia de seguridad con el vehículo de delante. En (4) se realiza dicha comprobación antes de empezar a frenar.

3.2.4 *Script* detección de colisiones

En primer lugar, para la detección de una colisión, es necesario tener un componente que nos indique cuándo se produce el contacto. Para ello se utiliza un componente de Unity llamado *Collider*. Se añadirá a cada vehículo un *Collider* llamado *Box Collider* que tendrá forma de cubo. Las dimensiones del cubo han de ser modificadas para que los planos queden alineados con el contorno del vehículo.

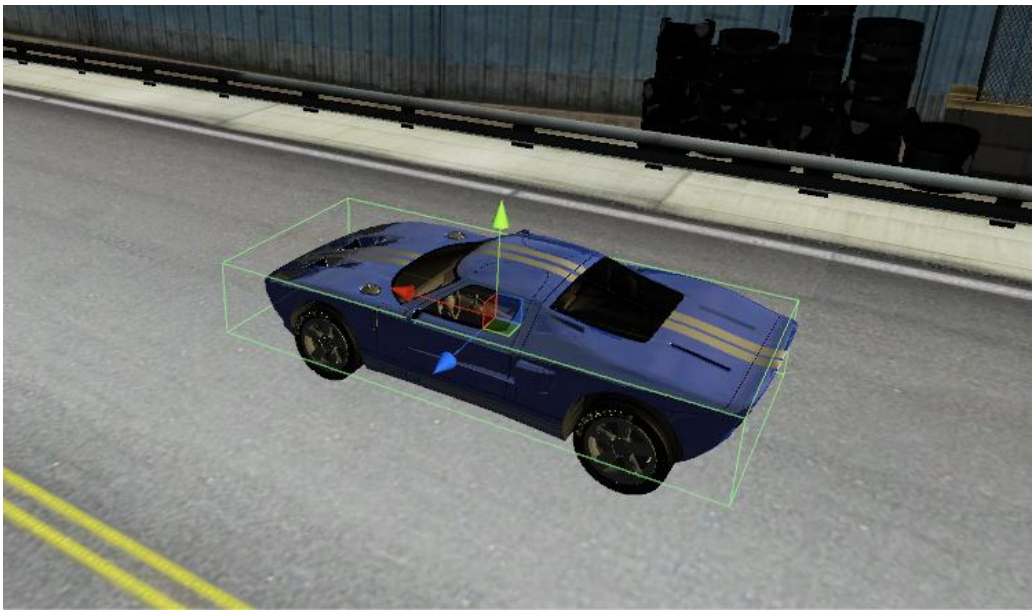


Fig. 3-12 Box Collider contorneado de color verde.

Además, dicho componente tiene una opción llamada *Is Trigger* que tenemos que marcar. Esta opción indicará al *Collider* que debe generar un evento cuando detecte una colisión con cualquier objeto.

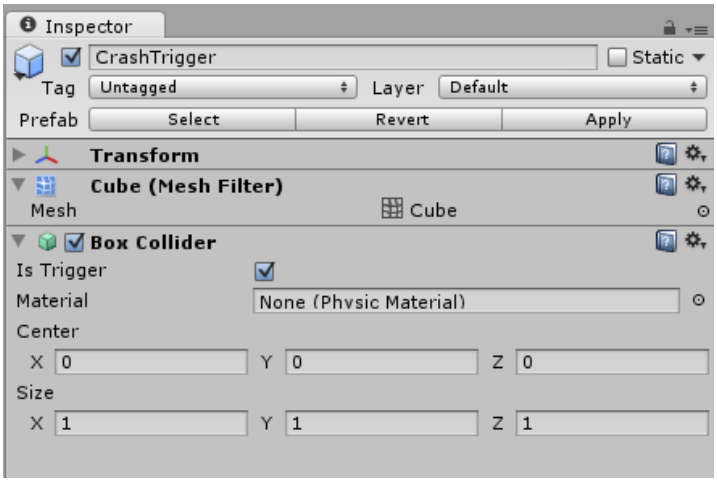
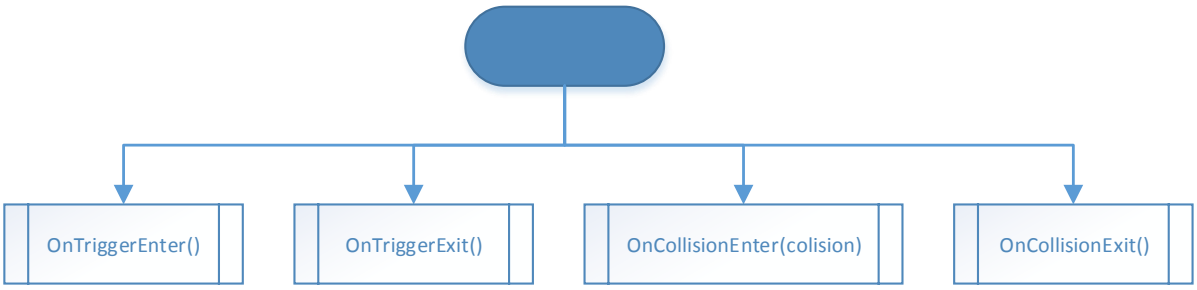
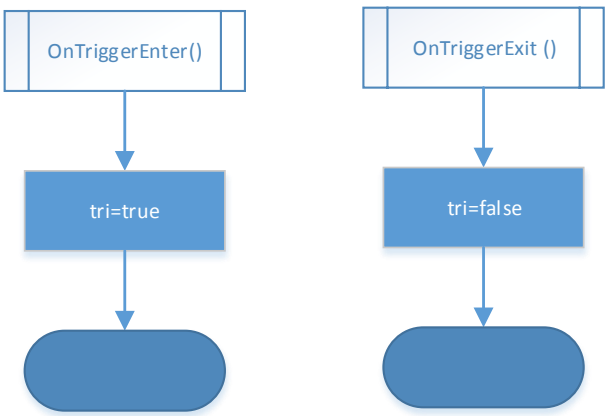
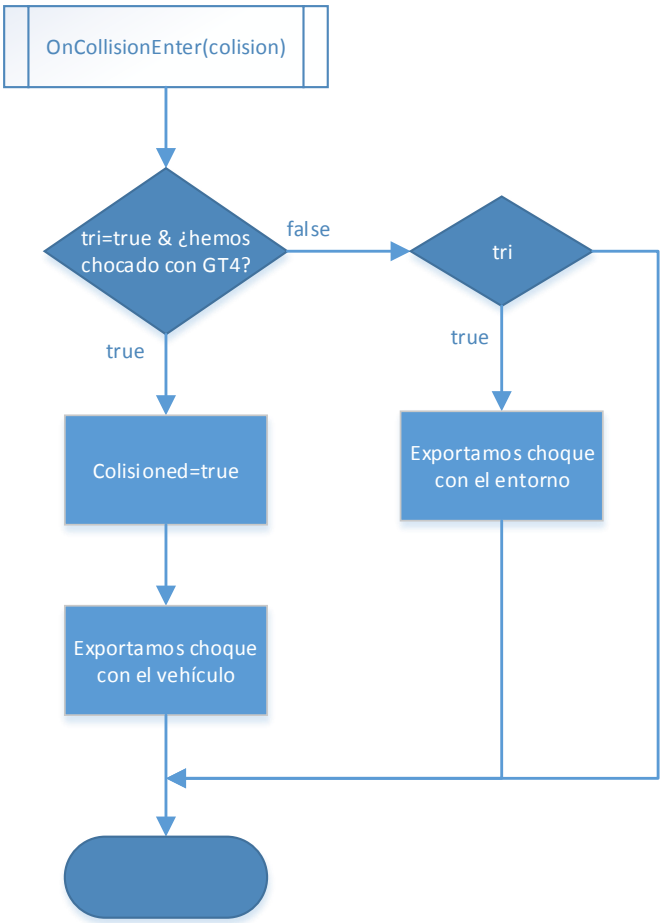


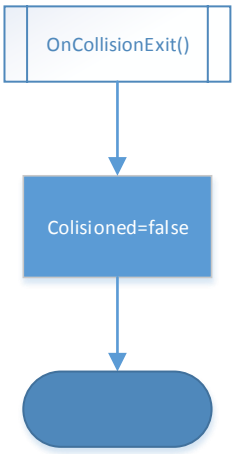
Fig. 3-13 Box Collider en el editor

Para manejar dichos eventos, se ha creado una clase llamada *CrashTrigger.* Dispondremos de dicha clase para el vehículo pilotado, pero existe una variante que llevarán asociados los vehículos autónomos, la clase *CrashTriggerReference*. Esta clase es similar, pero evitará que el vehículo pierda el contacto con el suelo al ser impactado por el coche del usuario. Simplemente corrige un comportamiento no deseado. A continuación se muestra el funcionamiento de *CrashTrigger* mediante un diagrama de flujo.






OnCollisionEnter(colision)
tri=true & ¿hemos chocado con GT4?
false
tri
true
true
Colisioned=true
Exportamos choque con el entorno
Exportamos choque con el vehículo


OnCollisionExit()
Colisioned=false

Cuando el *BoxCollider* esté en contacto con algo, se ejecutará la función *OnTriggerEnter* y cuando deje de estarlo, *OnTriggerExit*.

Para la correcta detección de choque con el vehículo han de cumplirse dos condiciones: que el *trigger* se active y que, además, se detecte una colisión con otro objeto, lo que hará que se ejecute la función *OnCollisionEnter.* Cuando el vehículo pierde el contacto con el objeto, se ejecutará *OnCollisionExit.* Una vez detectada la colisión, ya sea con otro vehículo o con el entorno, se exportará dicha información a un fichero como se verá en el siguiente punto.

3.2.5 Script de exportación de datos

Para obtener resultados de la simulación, se han implementado dos *scripts* que recogen datos en tiempo real y los guarda en tres ficheros.

Uno de los script*s*, llamado *ExportBehaviourData* está asociado al coche sin inteligencia artificial. Éste se encarga de crear dos ficheros. El primero de ellos se encarga de registrar la velocidad del vehículo en todo momento. El segundo fichero reflejará dos datos: los choques producidos con otro vehículo o con el entorno y los instantes en los que el coche de referencia comienza a frenar.

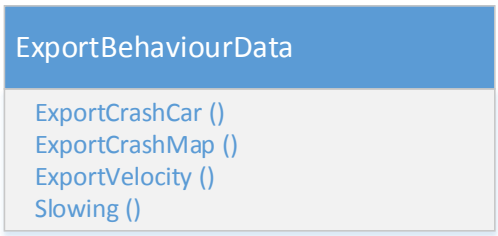


Fig. 3-14 Clase ExportBehaviourData

Las funciones *ExportCrashCar* y *ExportCrashMap* las llamará la clase *CrashTrigger* en el momento que detecte un choque con el otro vehículo o con el entorno respectivamente. Estos datos se almacenarán para una única simulación en un fichero llamado *crash_Car.txt.*

Por otro lado, la función *ExportVelocity* se ejecutará periódicamente cada "x" *frames* y generará el fichero *speed_Car.txt.*

Cabe destacar que la ruta donde se generarán los ficheros se puede modificar desde el menú inicial del simulador. En caso de no ser modificada, la ruta por defecto es:

C:\Users\Public\Documents

El otro *script* está contenido en el vehículo de referencia, denominándose *ExportVelocityReference*. Dicho *script* se encarga, al igual que anterior, de registrar la velocidad del vehículo durante toda la simulación. De esta manera, en caso de que se produzca un impacto entre vehículos, sabremos el momento exacto y la velocidad a la que circulaba cada uno.

Por tanto, tras la ejecución de una simulación se habrán creado tres ficheros de texto: crash_Car, speed_Car y speed_GT4. El primero contendrá los choques que se han producido y en qué momento. También se almacenarán los tiempos en los que el coche de referencia frena. Un ejemplo de lo que este fichero puede contener es el siguiente:

El coche referencia frena *t= 21.09195*
Se ha producido un choque con un vehiculo a 43.08 Km/h *t= 21.50491*

El segundo y tercer fichero reflejarán las velocidades de cada vehículo asociados al tiempo de la simulación:

velocidad: 1.03 Km/h *t= 1.671298*
velocidad: 3.1 Km/h *t= 1.798013*
velocidad: 5.05 Km/h *t= 1.923996*
velocidad: 6.96 Km/h *t= 2.051452*
velocidad: 9.14 Km/h *t= 2.186125*

3.2.6 *Script* de Interfaz Gráfica de Usuario

El *script* encargado de la GUI está desarrollado en C# y se llama *Menu*.

Cabe destacar, que fue necesario solucionar algunos aspectos de protección de variables. Uno de los problemas a solucionar fue el tener acceso a todas las variables necesarias, puesto que al encontrarnos en una escena diferente, dichas variables debían ser estáticas y públicas. De esa forma las variables son visibles desde otras escenas y pueden ser modificadas.

La solución anterior era válida para acceder sólo a las variables de las clases implementadas en C#, puesto que el motor compila por separado *scripts* de JavaScript y C# dentro de una fase de compilación. Debido a esto, no se tenía acceso a variables de los *scripts* en lenguaje *JavaScript*.

Para ello, había que compilar los *scripts* de Java en una fase anterior, por lo que fueron situados en el directorio *StandardAssets*.

Por otra parte, el *script Menu* sólo la compone una función, la función *OnGUI.* Ésta es una función específica de Unity, que se ejecuta constantemente para representar todos los objetos en la escena.

Todos los elementos del menú principal están situados con coordenadas relativas al tamaño de la pantalla donde se ejecute.

El *script* se compone de tres GUISkin diferentes. Cada GUISkin permite modificar el aspecto con el que se representarán los objetos. Diremos que un GUISkin almacena un perfil de parámetros que afectan a la forma de los botones, color de las letras, tipo de letra, etc. Por ejemplo, para la representación de las etiquetas disponemos de uno diferente al usado para los botones.

3.3 Interfaz gráfica de usuario

En primer lugar, el simulador dispone de una GUI que hace de menú principal, el cual permite realizar cambios previos en la simulación.

Este menú se divide en dos apartados*: vehículo del usuario* y *vehículo de referencia*. El primero de éstos permite realizar cambios sobre el vehículo que conducirá el usuario, pudiendo activar o desactivar la representación del velocímetro en la simulación [14]. Dicho velocímetro estará representado en la parte inferior derecha de la pantalla y mostrará la velocidad y la marcha actual del vehículo.

La segunda opción disponible permite habilitar la transmisión automática o manual de nuestro vehículo. En caso de habilitar la opción de *transmisión manual*, el cambio de marcha se realizará mediante los botones de cambio secuencial disponible en el controlador de juego Logitech G27, situados a la derecha e izquierda del volante.

La tercera opción de este apartado permite modificar el nivel de tráfico en el carril contrario de la vía. Existen cuatro niveles: *sin tráfico*, *tráfico fluido, tráfico medio y tráfico denso.* Por último en este apartado, disponemos de un cuadro de texto que permite modificar la ruta donde se almacenarán los datos de la última simulación realizada.

En cuanto al apartado de *vehículo de referencia,* el menú brinda la opción de activar o desactivar el vehículo de referencia que circulará en el mismo sentido que nosotros. También es posible elegir entre cuatro comportamientos diferentes de dicho vehículo. El primero de estos modos, el *Modo 1*, permitirá que el vehículo que va delante circule con normalidad durante todo el trayecto, es decir, sin grandes cambios de velocidad.

En segundo lugar, si seleccionamos la opción *Modo 2*, el vehículo referente frenará en caso de que no se respete la distancia de seguridad. En el *Modo 3*, el coche de referencia dispondrá de un sistema de frenado aleatorio, es decir, podrá frenar en cualquier momento de la simulación con un tiempo de media *t* entre frenazos. Por último la opción *Modo 4,* permitirá a dicho vehículo frenar mediante los dos sistemas anteriores, por distancia y de forma aleatoria.

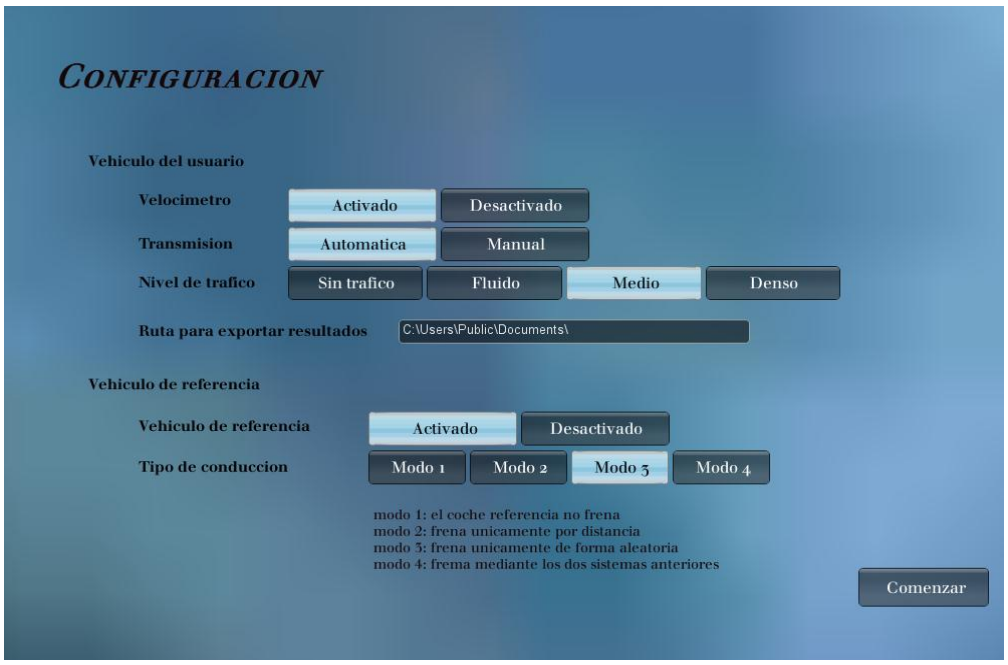


Fig. 3-15 Menú principal del simulador

3.4 Configuración del controlador de juegos Logitech G27

Para hacer aún más realista la conducción de nuestro simulador, se ha añadido la configuración de un controlador de juegos. Éste es el modelo G27 de la conocida marca Logitech. Se trata de un controlador formado por tres elementos: volante, plataforma de pedales y palanca de transmisión manual.

La simulación en modo de transmisión manual está configurada para cambiar de marcha con los botones laterales del volante, es la denominada transmisión secuencial. Debido a ésto, la palanca de cambio de velocidad no será utilizada para nuestras simulaciones.

Fig. 3-16 Controlador Logitech G27

En lo que a la configuración del controlador se refiere, ha sido necesario incluir algunas líneas de código en el *script* del vehículo a conducir. Algunas de ellas están representadas en la siguiente imagen.

```
function GetInput()
{

    if(Input.GetAxis("VerticalVolante")<0){
        throttle=0;
    }
    else if(Input.GetAxis("Vertical")!=0){
        throttle = Input.GetAxis("Vertical");
    }else{
        throttle = Input.GetAxis("VerticalVolante");
    }
    if(Input.GetAxis("Brake")<0){
        throttle = Input.GetAxis("Brake");
    }
    steer = Input.GetAxis("Horizontal");
```

Fig. 3-17 Código añadido para la configuración del controlador

Por otro lado, ha sido necesario configurar nuevas entradas en Unity que correspondiesen a los ejes y botones del controlador. En la siguiente imagen se pueden apreciar las entradas adicionales, mostrándose en las últimas cinco posiciones de la lista. Podemos ver que en el campo *Type* está seleccionado *Joystick Axis*, que corresponde al identificador del volante, es decir, el eje horizontal de movimiento. De esta forma han sido configuradas las entradas restantes.

Cabe destacar que el simulador está configurado para ser dirigido indistintamente tanto con el teclado del ordenador como con el controlador.

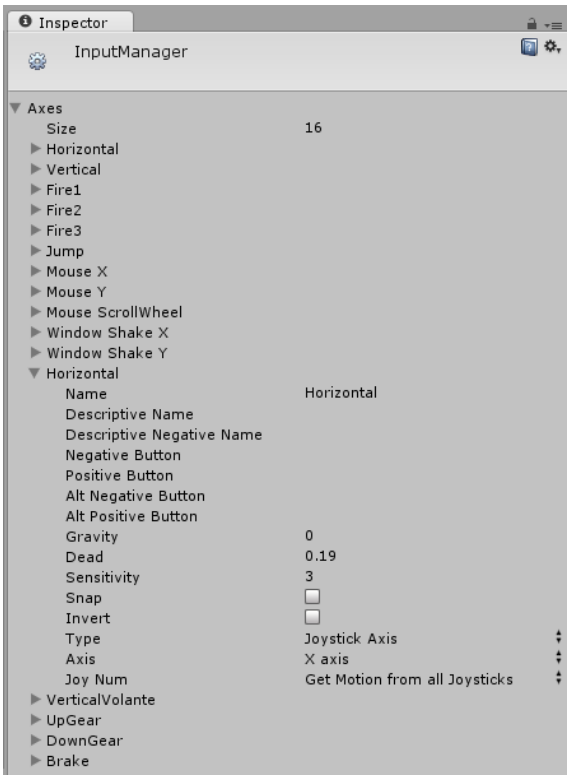


Fig. 3-18 Editor de entradas de Unity

4. Simulación

El trazado por el que transcurre la simulación es un circuito cerrado de dos carriles. Es una ruta con zonas muy variadas en que podemos encontrar curvas sin demasiada visibilidad, cambios de rasante e incluso un túnel. En cuanto al decorado, también existe una variedad notable. Durante la trayectoria nos vamos a encontrar con zonas montañosas y de mar, además de objetos e instalaciones que nos pueden recordar a un circuito de automovilismo.

La siguiente imagen muestra cómo sería la interfaz. Puesto que se trata de un simulador en primera persona, la cámara está situada en la posición del conductor. En la parte superior central de la imagen podemos encontrar el espejo retrovisor del vehículo.

Si queda previamente configurado, en la parte inferior derecha, podemos encontrar el velocímetro. Se trata de un diseño único, creado en Photoshop a partir de algunas ideas vistas en la red. En el interior del velocímetro se representa el número de velocidad a la que va el vehículo; en el ejemplo, va en primera marcha. Por último, en la parte superior izquierda, se encuentra representado el tiempo en segundos desde que inició la simulación.

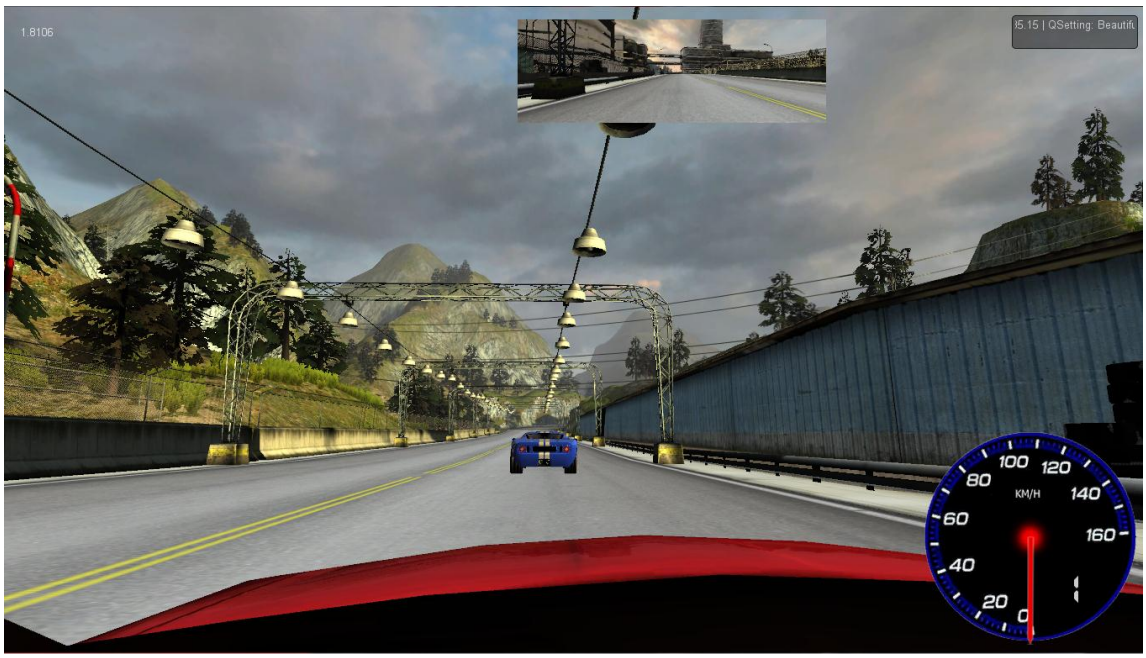


Fig. 4-1 Vista previa de la simulación

En la siguiente imagen queda representado el menú de pausa en mitad de la simulación. Se puede ver que la luz del paisaje se reduce y aparecen en la mitad de la pantalla las posibles opciones del menú.

Fig. 4-2 Menú de pausa en simulación

5. Conclusiones y líneas futuras

En este proyecto hemos diseñado un simulador de conducción en un entorno de desarrollo de videojuegos en tres dimensiones llamado Unity. El simulador trata de reproducir algunas situaciones de riesgo de accidente para así experimentar la reacción de los conductores.

Esta aplicación permite simular varios comportamientos en un vehículo de referencia. Dicho vehículo "líder" circulará por delante del coche que el usuario podrá conducir. Está desarrollado en primera persona y permite una conducción realista gracias a la incorporación de un controlador de juegos, formado por volante y pedales.

Unity es una herramienta de desarrollo muy intuitiva y fácil de utilizar que, mediante *scripts,* permite crear aplicaciones con muy altas prestaciones.

Este proyecto nos ha permitido acercarnos un poco más al mundo de las aplicaciones en 3D, sobre todo a la parte del desarrollo software. Durante el transcurso del proyecto se ha dado a conocer la infinidad de posibilidades que un entorno como éste presenta. No sólo nos referimos a la creación de simuladores de conducción, sino a la gran variedad de aplicaciones que se pueden implementar tanto en dos como en tres dimensiones.

Un proyecto de este tipo no dispone de muchas limitaciones a la hora de introducir modificaciones. A pesar de que esta aplicación tenga como objetivo conocer el comportamiento de los conductores, el abanico de funcionalidades que se pueden añadir es muy amplio.

Además, se ha dado a conocer el funcionamiento de las herramientas de desarrollo de aplicaciones 3D como Unity. Este entorno de edición se ha presentado como una muy buena opción a tener en cuenta para desarrolladores no tan expertos. Unity ofrece una buena curva de aprendizaje y posee una gran versatilidad y sencillez para crear desde cero aplicaciones o videojuegos 3D.

También se han descrito brevemente algunas de las opciones alternativas disponibles en el mercado, todas ellas comparadas con Unity que es el entorno elegido para nuestro trabajo. Algunas de estas opciones son muy interesantes para la implementación de simuladores de conducción como éste.

Por último se ha descrito en detalle todo lo que ha conllevado este proyecto, desde las consideraciones previas a tener en cuenta, hasta el desarrollo del simulador en su totalidad.

Cabe destacar que éste es el trabajo que inicia una investigación cuyo principal objetivo no se consigue únicamente con este proyecto. Esta aplicación abre la puerta para continuar con la investigación a futuros proyectos.

El objetivo final de la investigación es desarrollar un simulador distribuido en el que estén involucrados varios conductores, al que se integrará con un simulador de comunicaciones. Por otro lado, también se probarán otros algoritmos como *car-following.*

Car-following es un modelo creado por Newell que se utiliza para determinar cómo los vehículos circulan detrás de otros. La idea de este modelo parte de que un vehículo mantenga un intervalo de espacio-tiempo mínimo con el vehículo que le precede. Consecuentemente, si en condiciones de congestión de tráfico el vehículo "líder" varía su velocidad, el vehículo que le sigue modificará también su velocidad en un punto espacio-tiempo determinado.

6. Bibliografía

[1] Unity Technologies. *http://unity3d.com*

[2] Unity Technologies. *http://docs.unity3d.com/Documentation/Manual/index.html.*

[3] Unity Technologies. *http://docs.unity3d.com/Documentation/Components/index.html*

[4] Unity Technologies. *http://docs.unity3d.com/Documentation/ScriptReference/index.html*

[5] Stonetrip. Shiva3D. *http://www.shivaengine.com/*

http://www.stonetrip.com/developer/forum/viewtopic.php?f=18&t=25507

[6] GarageGames. Torque3D. *http://www.garagegames.com/products/torque-3d*

http://www.gamedev.net/topic/644163-torque-3d-vs-unity-3d/

[7] Image Space Incorporated. Rfactor. *http://www.rfactor.net/*

[8] CarX Technologies. Car-X. *http://www.carx-tech.com/*

[9] VDrift. *http://vdrift.net/*

[10] OpenDS. *http://www.opends.eu/*

[11] PixelPlacement. *http://itween.pixelplacement.com/index.php*

[12] Arges Systems Inc. *http://arges-systems.com/*

[13] *http://www.red3d.com/cwr/steer/ FUERZAS UNITYSTEER*

[14] *http://blind-soft.info/blindgui/examples/topic/example3*

Otras páginas que de apoyo:

http://unityscripts.blogspot.com.es/

http://www.unityspain.com/index.php/foro/38-unity

http://forum.unity3d.com/forum.php

http://www.c-sharpcorner.com/uploadfile/rohatash/working-with-file-class-in-C-Sharp/

https://github.com/ricardojmendez/UnitySteer/commits/development

http://www.youtube.com/watch?feature=player_embedded&v=XvBQELZTUKY#!

http://www.youtube.com/user/FenixDiscom?feature=watch

http://www.youtube.com/user/MrJocyf?feature=watch

http://trinit.es/2010/10/22/tutorial-de-unity3d-en-espanol/