UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO EN INFORMÁTICA

DESARROLLO DE UN JUEGO DE CARRERAS CON SIMULACIÓN FÍSICA Y APRENDIZAJE AUTOMÁTICO

Realizado por

D. SERGIO PAQUE MARTÍN

Dirigido por

DR. D. MARLON NÚÑEZ PAZ

Departamento

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

MÁLAGA, de 2009

UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO EN INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente Dº/Dª. _____

Secretario Dº/Dª. _____

Vocal Dº/Dª. _____

para juzgar el proyecto Fin de Carrera titulado:

DESARROLLO DE UN JUEGO DE CARRERAS CON SIMULACIÓN FÍSICA Y APRENDIZAJE AUTOMÁTICO

realizado por **D. SERGIO PAQUE MARTÍN**

dirigido por **DR. D. MARLON NÚÑEZ PAZ**

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga a ___ de _____ del 2010

El/La Presidente/a	El/La Secretario/a	El/La Vocal
Fdo.:	Fdo.:	Fdo.:

Índice general

Índice de figuras

Índice de Listados

Capítulo 1

Introducción

1.1 Antecedentes

Cincuenta años después de su aparición, el videojuego se ha convertido recientemente en uno de los campos de estudio más de moda y volátil dentro de la informática. En los últimos años, si se hacía referencia a los videojuegos era sólo para ponerlos como un ejemplo entre muchos de las nuevas tecnologías (y además de forma marginal). Pero a medida que han ido madurando, los videojuegos se han convertido en un elemento clave entre los medios digitales, y finalmente empieza a reconocerse su importancia.

Actualmente el videojuego es considerado todo tipo de cosas. Se considera narración, simulación y arte; una potencial herramienta para la educación o un objeto de estudio para la psicología del comportamiento; un medio para la interacción social y, por supuesto, un medio de distracción. Por otra parte el desarrollo de videojuegos es una tarea multidisciplinar que abarca campos tales como la ingeniería del software, los gráficos por ordenador, la física y la inteligencia artificial, los cuales constituyen los pilares básicos.

La calidad gráfica que exhiben los juegos actuales, también conocidos como juegos de nueva generación, es asombrosa. Algo impensable hace unos años, y que ha sido posible gracias a la amplia variedad de investigaciones en gráficos por ordenador, a la evolución del hardware gráfico (muchas veces impulsado por la misma industria de los videojuegos) y a la introducción de conceptos tales como la programación de dicho hardware gráfico. Así mismo, pese a la creciente complejidad en lo referente al procesamiento gráfico, existe una tendencia incipiente a facilitar las tareas en este apartado. Este hecho se demuestra con la aparición de lo que se conoce como “motor gráfico”: un componente software que proporciona una abstracción de todo el procesamiento que conlleva el manejo de gráficos en una aplicación. En este ámbito han aparecido varias alternativas, algunas de ellas para uso comercial y otras para uso libre. Cabe destacar OGRE 3D, un popular motor gráfico de código abierto que ha

sido usado en un amplio abanico de aplicaciones gráficas. También podemos destacar motores gráficos comerciales como CryEngine 3 o Unreal Engine 3, que constituyen un claro exponente del alto nivel de realismo visual que se puede conseguir con la tecnología actual. En el mundo Java podemos citar el Xith3D y el Java3D, los cuales ofrecen grandes facilidades de modelado y programación 3D, aunque han sido menos utilizados que los anteriores en el desarrollo de videojuegos comerciales debido un menor realismo en sus escenas.

La física es una disciplina muy amplia que tiene cientos de campos de estudio. Cada uno de ellos describe un aspecto del mundo físico, y algunos resultan útiles en el desarrollo de videojuegos. La simulación de la dinámica y de la interacción física entre los objetos presentes en el juego es un factor muy importante a la hora de dar sensación de realismo. Tanto es así que se ha llegado incluso a comercializar hardware exclusivamente dedicado al procesamiento físico. Al igual que ocurre con los gráficos, la complejidad de la física presente en los juegos ha dado pie a la aparición de motores físicos, que tiene como fin facilitar al programador la tarea de hacer que la dinámica del juego sea fiel a la realidad.

Un aspecto que influye decisivamente en la calidad de un juego es la inteligencia artificial. El objetivo que se persigue al incorporar inteligencia a un juego es dotar a un personaje de una cierta capacidad de decisión y de un comportamiento autónomo. Esto produce en el jugador la ilusión de que está jugando contra un jugador inteligente y aumenta la jugabilidad. A menudo el término inteligencia artificial se usa para referirse a un amplio conjunto de algoritmos y técnicas que se utilizan para crear agentes inteligentes. Por lo tanto, la definición de inteligencia artificial en un juego es tan amplia como flexible. Al igual que ocurre con el uso de unas físicas reales, una buena inteligencia artificial ayuda a la inmersión del jugador en el juego, y por lo tanto constituye un elemento clave. Dentro del amplio rango de técnicas disponibles podemos mencionar algunas de las más utilizadas actualmente en el ámbito de los videojuegos: algoritmos de navegación, máquinas de estado finito, lógica difusa y algoritmos de aprendizaje mediante redes neuronales, algoritmos genéticos y minería de datos entre otras. En definitiva la inteligencia artificial es uno de los campos más presentes en departamentos de investigación y desarrollo de muchas compañías dentro de la industria del videojuego.

1.2 Objetivos

El objetivo de este Proyecto Fin de Carrera es desarrollar un videojuego de carreras en 3D para PC. El jugador asumirá el papel del conductor de uno de los vehículos, correrá contra diversos adversarios y tendrá como meta finalizar la carrera en primer lugar. El juego podrá tener lugar tanto en superficies de asfalto como en superficies de barro o gravilla.

Para desarrollar el juego será necesario apoyarse en los cuatro pilares básicos mencionados anteriormente. Es indispensable utilizar técnicas de ingeniería del software para seguir una metodología que asegure la calidad del software final. Para ello se utilizará un lenguaje orientado a objetos como es el lenguaje de programación C++ bajo el entorno de programación Visual Studio 2005.

Para que en el apartado gráfico se pueda conseguir un aspecto visual atractivo se hará uso del motor gráfico OGRE 3D. Este constituye la elección perfecta debido a su amplio conjunto de características, al buen diseño de su API y a la cantidad de documentación disponible, además de la existencia de una comunidad de usuarios bastante activa y de múltiples herramientas integradas en los programas de modelado 3D más populares.

En lo referente a la física de los vehículos, lo que se persigue es conseguir que la dinámica de dichos vehículos se asemeje lo máximo posible a la realidad. Será necesario encontrar un equilibrio entre el realismo y el coste computacional que esto conlleva. Para lograr este objetivo se utilizará el motor físico Havok, que ofrece un excelente compromiso entre rapidez, fiabilidad y precisión. Igualmente cuenta con una extensa documentación y con varias herramientas compatibles con las aplicaciones de modelado 3D más habituales. Así mismo ofrece la posibilidad de obtener soporte a través de un foro público.

Controlar un coche en una superficie resbaladiza, como barro o gravilla, es bastante más complicado que hacerlo en una superficie de asfalto. Por ejemplo, en asfalto tan sólo hay que dirigir el coche en la dirección adecuada y mantener la velocidad correcta, mientras que en superficies resbaladizas el vehículo tiene que reducir la velocidad, tomar antes la curva y derrapar. Crear un conjunto de reglas para contemplar cada posible situación resultaría extremadamente complejo. Por lo tanto la inteligencia artificial de los vehículos estará implementada mediante técnicas de aprendizaje automático, en particular se utilizarán redes

neuronales. Lo que se persigue con esto es que un vehículo sea capaz de imitar las acciones de un jugador humano y que en última instancia llegue a ser competitivo a un nivel en el que la jugabilidad sea aceptable.

El proyecto exige pues la reunión de los cuatro aspectos mencionados en los puntos anteriores, poniendo especial énfasis en la simulación física y en el aprendizaje automático.

1.3 Contenidos de la memoria

La presente memoria se divide en los siguientes capítulos:

- Simulación física

  En este capítulo se pone en contexto la simulación física dentro de los videojuegos, se hace una introducción a los motores físicos y una vez hecha la elección de uno de ellos tratamos sus aspectos fundamentales. Por último nos adentramos en la simulación física de vehículos utilizando el motor físico.

- Aprendizaje automático

  Este capítulo hace un repaso de los conceptos fundamentales relacionados con el aprendizaje automático. Presentamos las técnicas actuales y su uso en videojuegos. A lo largo del capítulo se explica el desarrollo de las redes neuronales utilizadas, desde el análisis, pasando por el diseño, hasta la implementación y el entrenamiento.

- Requisitos de la aplicación

  A partir de este capítulo nos centramos en el desarrollo de la aplicación como un proyecto software. Como tal, comenzamos realizando un análisis del proyecto, recogiendo los requisitos y elaborando los casos de uso.

- Análisis y diseño de la aplicación

  Tras la fase anterior podemos continuar con el análisis y diseño de la misma. Diseñamos la arquitectura del sistema y presentamos desde el punto de vista del diseño el motor gráfico que tenemos que integrar en nuestro proyecto. Así mismo se

explica el diseño del contenido del juego y se presentan las clases de las que está compuesta nuestra aplicación.

- Implementación de la aplicación

  Este capítulo, al dedicarse a la implementación del proyecto, es el de mayor extensión. Se explica el desarrollo de la aplicación basándose en los capítulos anteriores. Una parte fundamental del capítulo es la presentación del conjunto de bibliotecas que se utilizan en la implementación del proyecto.

- Entrenamiento y pruebas

  En este capítulo presentamos los resultados obtenidos tras realizar el entrenamiento de las redes neuronales, así como las pruebas realizadas.

- Conclusiones

  Por último en este capítulo se expresan las conclusiones del proyecto. Se revisa el cumplimiento de los objetivos originales y se exploran posibles mejoras a desarrollar en el futuro.

Capítulo 2

Simulación física

2.1 Simulación física en videojuegos

La física es una amplia disciplina con cientos de subcampos. Cada uno de ellos describe un aspecto del mundo físico, ya sea la forma en la que funciona la luz o las reacciones nucleares dentro de una estrella. Pero cuando hablamos de física en el contexto de un videojego nos referimos a la mecánica clásica: las leyes que gobiernan el movimiento de grandes objetos bajo la influencia de la fuerza de la gravedad u otras fuerzas.

La simulación física en los videojuegos ha estado presente prácticamente desde los comienzos y a medida que la potencia computacional iba creciendo también lo hacía la calidad de las simulaciones.

Aunque la física siempre ha estado presente en los juegos, la forma en que se implementa ha cambiado en los últimos años al incrementar la complejidad de la simulación. Surgió lo que se conoce como motor físico: un software capaz de realizar una simulación física general, pero que no está programado con las particularidades de cada posible escenario de un juego.

Existen dos tipos de motores físicos: en tiempo real y de alta precisión. La simulación física de los motores físicos de alta precisión requiere más potencia de procesamiento y por lo general son utilizados en el ámbito científico y en películas de animación. En los videojuegos, u otro software interactivo, el motor físico tendrá que simplificar sus cálculos y disminuir su precisión para que puedan llevarse a cabo en tiempo real y tenga una respuesta apropiada para el desarrollo del juego.

Por lo general, en un videojuego el motor físico tiene que realizar tres tareas básicas:

- **Detección de colisiones**. Rastrear el movimiento de los objetos de la escena y detectar cuando colisionan entre sí.

- **Actualizar el sistema**. Determinar las respuestas apropiadas para los objetos que han colisionado resolviendo la colisión de acuerdo a sus propiedades y actualizar los demás objetos atendiendo a las fuerzas que actúan sobre ellos.
- **Interfaz con la representación gráfica**. Una vez que las nuevas posiciones de los objetos han sido calculadas, normalmente necesitamos pasarle esta información al sistema gráfico.

En la actualidad existe un amplio rango de motores físicos para juegos, cada uno con sus características y limitaciones, por lo que decantarse por uno es una ardua tarea. A la hora de escoger uno de ellos nos tenemos que fijar en cuatro aspectos: características, documentación, usabilidad y herramientas disponibles.

De entre todos los motores físicos disponibles nos centramos en los siguientes:

- Open Dynamics Engine,
- PhysX,
- Bullet y
- Havok.

Tras una evaluación de todos ellos nos decantamos por Havok por las siguientes razones:

- Incluye un extenso y robusto rango de características, incluyendo simulación física de vehículos, que cubren todas nuestras necesidades y están ampliamente probadas a través de su uso en cientos de juegos comerciales.
- La documentación del motor físico es excelente, no sólo se documenta el API, también se documenta el funcionamiento del motor físico y los principios físicos que lo rigen. Además está disponible un activo foro donde se ofrece soporte técnico.
- La documentación del motor físico hace posible un rápido aprendizaje del mismo. Además el diseño de Havok ofrece una alta usabilidad y fácil integración en cualquier aplicación que requiera incluir simulaciones físicas en tiempo real.
- Havok proporciona un conjunto de herramientas que lo hacen tremendamente valioso. Se puede integrar en programas de modelado 3D y serializar los contenidos para su uso posterior en el juego. Otra importante herramienta es el depurador visual que ofrece, lo que nos permitirá detectar y arreglar posibles fallos en la simulación física

de una manera sencilla.

2.2 Havok

El principal objetivo de Havok es proporcionar una simulación física que parece real. En varias ocaciones se tienen que realizar suposiciones y utilizar “atajos” para obtener una simulación lo más rápida posible. Con estos “atajos” se intenta disminuir la precisión, pero no la credibilidad y uno de los mayores aliados es el caos. El mundo físico es inherentemente caótico y uno de los objetivos del motor físico es imitar este caos.

Havok es un motor físico multihebra que permite dotar a objetos de cualidades físicas en tres dimensiones. Mediante la simulación de procesos físicos de Havok es posible crear mundos virtuales verosímiles, crear objetos físicos dentro del mismo, asignar propiedades físicas a estos objetos, avanzar la simulación física en el tiempo y examinar los resultados de la simulación. Se puede personalizar casi cualquier aspecto físico durante la simulación (e.g. cambiar la fuerza de la gravedad, aplicar fuerzas o pares de torsión a los objetos, añadir o eliminar objetos dinámicamente).

Además proporciona un amplio abanico de herramientas:

- Herramientas para creación de contenido.
- Herramientas para depuración y análisis de comportamiento.
- Integración con programas de modelado como 3D Studio Max o Maya.
- Herramientas para realizar modificaciones y puestas a punto.

Así mismo, Havok proporciona un marco de trabajo orientado a objetos y diseñado para ser modular y extensible.

De entre la multitud de clases de Havok, nosotros estamos interesados en un subconjunto de ellas (ver Figura 2.1), las cuales tendremos que integrar en el diseño de nuestra aplicación.

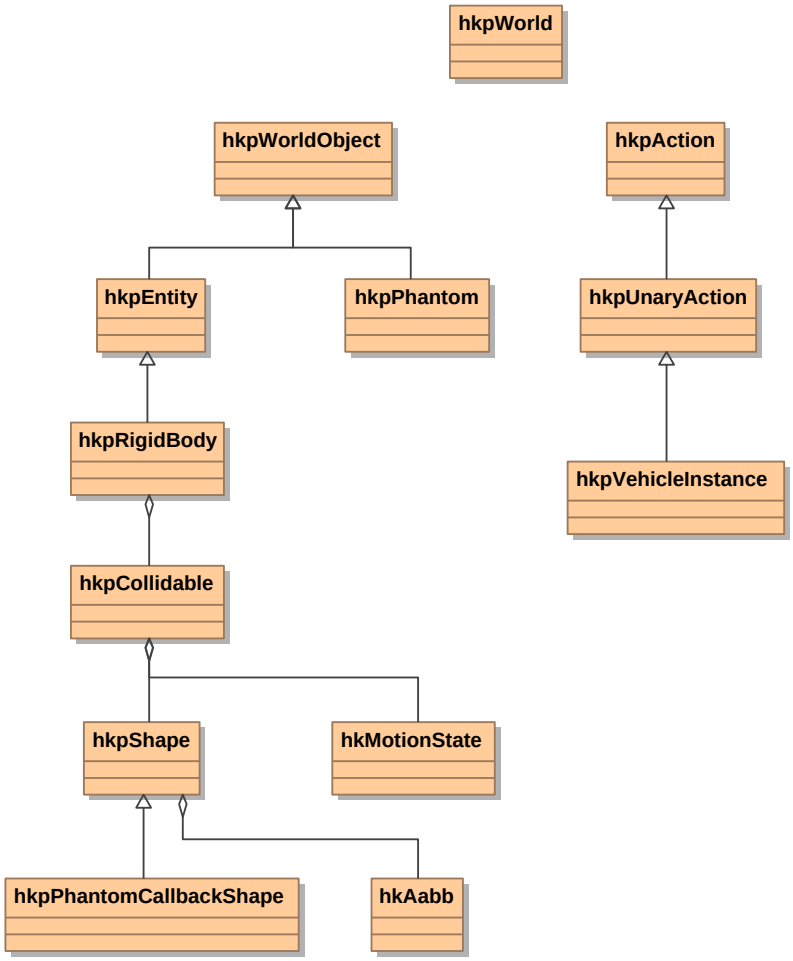

Figura 2.1. Clases de Havok

2.2.1 World

Cada simulación en Havok tiene un mundo físico o instancia de *hkpWorld*. Esta actúa como contenedor para los objetos físicos, y también es usada para avanzar la simulación en el tiempo.

Para que pasen a formar parte de la simulación, elementos tales como cuerpos rígidos (*hkpRigidBody*), fantasmas (*hkpPhantom*) o acciones (*hkpAction*) tienen que ser añadidos al mundo físico.

2.2.2 Shape

A través de la clase *hkpShape* se define la forma de un objeto y cómo colisiona contra otros objetos. La forma de un objeto puede ser implícita, como una esfera o un plano, o explícita,

representada por una malla de puntos exportada de un modelador.

Puesto que sólo contiene propiedades estáticas las instancias de *hkpShape* pueden ser compartidas por más de una entidad.

A cada *hkpShape* se le asocia un volumen que lo envuelve y que además está alineado con los ejes de coordenadas (*hkAabb*), también conocido como AABB (Axis Aligned Bounding Box) y que viene dado por las coordenadas mínimas de cada eje. Esta información se usa en la fase de detección de colisiones de Havok.

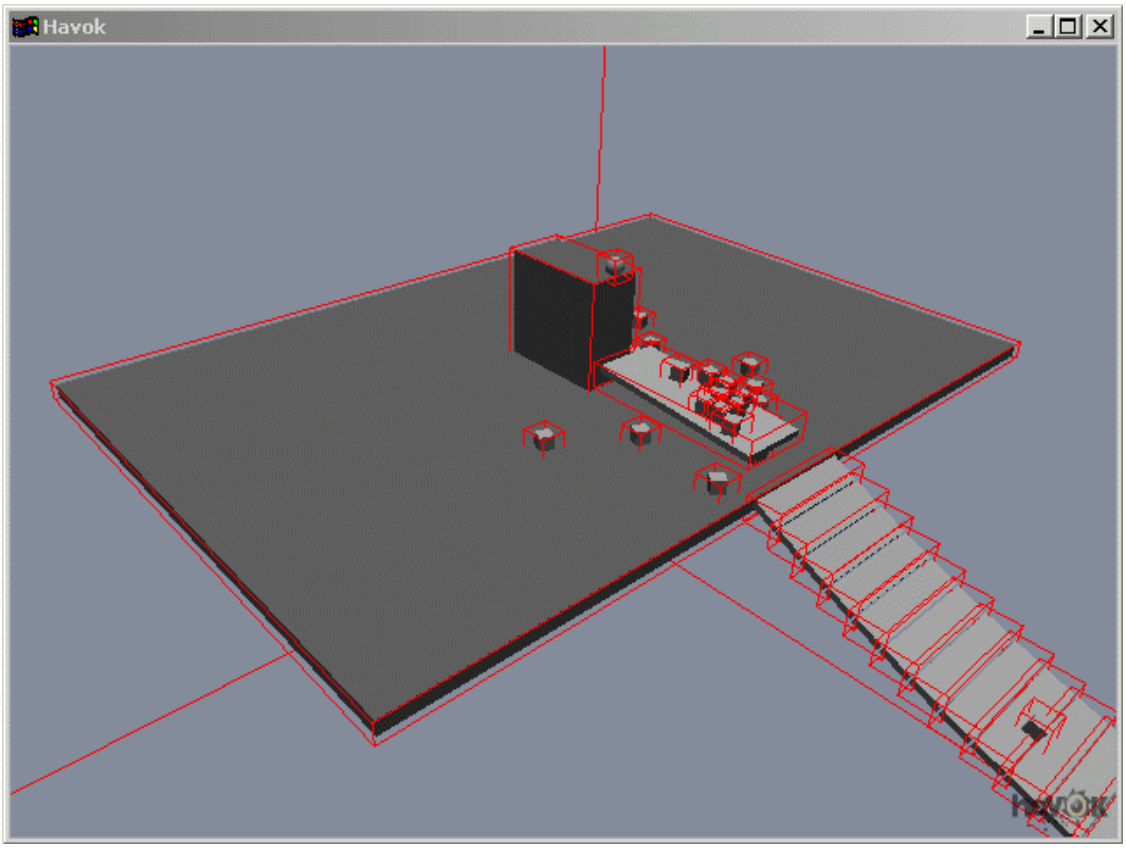

Figura 2.2. AABB en Havok

2.2.3 Collidable

El objeto más importante de cara a la detección de colisiones es *hkpCollidable*. Para llevar a cabo la detección de colisiones, cada instancia de *hkpCollidable* necesita la representación de su superficie (*hkpShape*) y una posición (*hkMotionState*).

Havok funciona con cuatro dimensiones, siendo la cuarta dimensión el tiempo, por lo que

hkMotionState también incluye información sobre el movimiento de un objeto entre dos instantes de tiempo.

Una importante característica es la posibilidad de realizar un filtrado de colisiones, es decir, se puede habilitar o deshabilitar la colisión entre ciertos objetos.

2.2.4 Rigid Body

Los cuerpos rígidos (*hkpRigidBody*) son parte central de Havok. Cualquier objeto real que no cambie su forma, desde el chasis de un coche a una piedra, puede ser simulado por Havok como un cuerpo rígido.

Contiene atributos tales como masa, centro de masa, tensor de inercia, fricción, restitución, posición o velocidad. Sobre un cuerpo rígido podemos realizar operaciones como cambiar su velocidad, aplicarle fuerzas, pares de torsión o impulsos.

Un cuerpo rígido puede ser de tipo dinámico o fijo. Un cuerpo fijo tendrá masa infinita y nunca cambiará su posición o rotación.

2.2.5 Phantom

Un fantasma (*hkpPhantom*) no tiene presencia física en la simulación y se usa principalmente para mantener una lista de entidades (u otros fantasmas) con los que se superpone. La aplicación más común de un fantasma es detectar las entidades que entran o dejan un volumen de espacio determinado para, por ejemplo, iniciar una escena de vídeo o alertar a un agente inteligente de una presencia.

Figura 2.3. Posible uso de una forma fantasma

2.2.6 Action

Las acciones (*hkpAction*) en Havok proporcionan un simple mecanismo para controlar el comportamiento de ciertos objetos durante la simulación. Pueden ir desde un simple efecto anti gravitatorio para hacer flotar un objeto, hasta situaciones más complejas como el control de un vehículo como veremos en el próximo punto.

2.3 Utilización de Havok

Además de las clases dedicadas a la simulación física, Havok proporciona un completo sistema para la gestión de memoria, temporización, tipos de datos básicos y contenedores y un mecanismo para ejecución de varias hebras.

En este apartado se presentan los aspectos más importantes que utilizaremos en la implementación del proyecto.

2.3.1 Inicialización

Havok está compuesto por un cierto número de subsistemas, cada uno de los cuales requiere ser inicializado. Bajo el nombre de espacio *hkBaseSystem* se encuentra la funcionalidad para crear los subsistemas de Havok. Muchos de estos siguen el patrón de diseño singleton y tienen que ser inicializados en un orden concreto, por ejemplo: gestión de memoria, manejo de errores, manejo de flujos.

Tras invocar al método `init()` podemos empezar a utilizar el motor físico Havok. Así mismo, es necesario inicializar Havok desde cualquier hebra en que se utilice mediante `initThread()`.

2.3.2 Gestión de objetos

Ciertos objetos pueden tener más de un dueño, por lo que para cada objeto Havok mantiene información acerca de la cantidad de objetos que lo referencian. Esta información se utiliza para facilitar la gestión de la memoria. Este tipo de objetos heredan las propiedades de *hkReferencedObject* y mantienen una cuenta de referencias, la cual puede ser incrementada mediante el método `addReference()` o decrementada mediante `removeReference()`.

2.3.3 Serialización de objetos

La serialización es el proceso de convertir datos a un formato que se pueda volcar a un fichero. Una vez serializado, los datos originales pueden ser reconstruidos.

Cuando se serializan objetos físicos de Havok, sólo se guardan las propiedades que se necesitan para recrearlos, tales como posición, orientación, dimensiones e impulsos.

La clase *hkpPhysicsSystem* es un contenedor que almacena colecciones de objetos y sus acciones y restricciones asociadas para serializarlos conjuntamente. Está formado por cuatro tipos de componentes: *hkpRigidBody*, *hkConstraint*, *hkpAction* y *hkpPhantom*. Por ejemplo, un personaje puede consistir en una cantidad de cuerpos rígidos y las restricciones que los mantienen juntos, mientras que un vehículo está definido por un cuerpo rígido, un objeto fantasma y una acción.

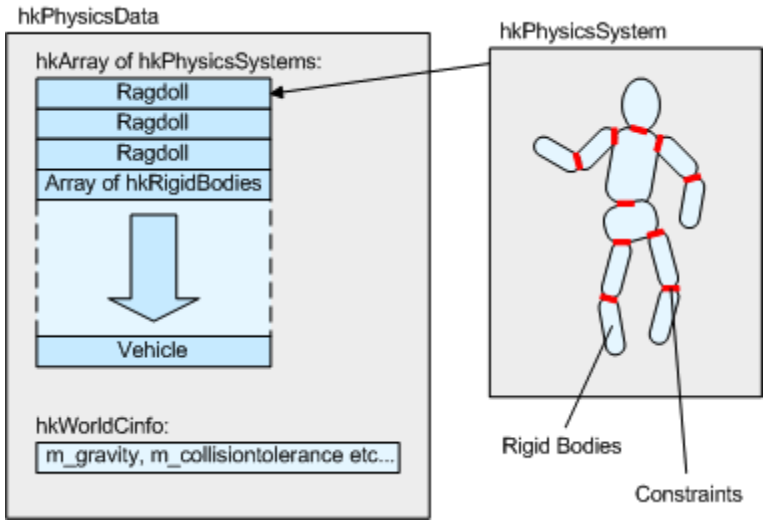

Figura 2.4. Serialización en Havok

Las herramientas de Havok para Autodesk 3ds Max agrupan los objetos físicos en una instancia de *hkpPhysicsData* para serializarla y empaquetarla en un archivo.

2.3.4 Mundo físico

Para realizar la simulación física se necesita crear una instancia de la clase *hkpWorld*, la que actuará de contenedor para todos los objetos físicos. Para mejorar el rendimiento el mundo físico se particiona en islas de simulación que pueden ser procesadas independientemente.

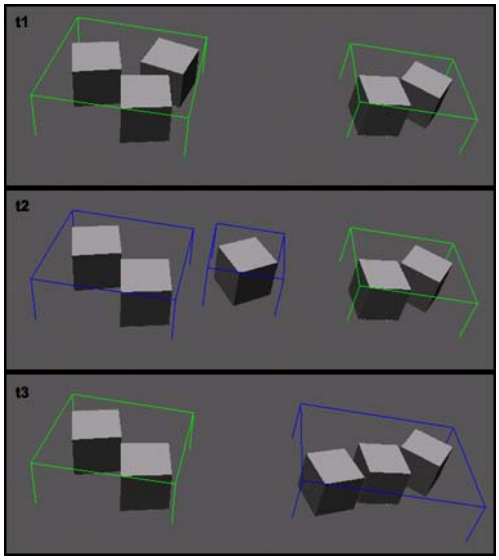

Figura 2.5. Islas de simulación en Havok

Cuando queremos crear un objeto *hkpWorld* necesitamos utilizar la estructura *hkpWorldCinfo*, la cual contiene toda la información para su creación. Entre otros, podemos especificar parámetros tales como la fuerza de la gravedad, tolerancia de colisión, tipo de simulación o el tamaño del mundo físico.

2.3.5 Cuerpos rígidos

La creación de un cuerpo rígido también necesita de una estructura auxiliar, *hkpRigidBodyCinfo*, que contiene la configuración del mismo. Esta información puede ser forma del objeto, información de filtrado para la colisión, material, posición, orientación, masa, etc.

Una vez creado el objeto podemos añadirlo al mundo físico mediante `addEntity()`, cambiar su estado de movimiento aplicando fuerzas con `applyForce()` e impulsos con `applyLinearImpulse()`, desactivarlo mediante `deactivate()` y eliminarlo del mundo físico a través de `removeEntity()`.

2.3.6 Avanzar la simulación

Como se ha mencionado en apartados anteriores, *hkpWorld* puede ser utilizado para avanzar la simulación en el tiempo. Esto se consigue llamando al método `stepDeltaTime()` pasándole como argumento la diferencia de tiempo en segundos. Después de llamarlo, las posiciones y

velocidades de los objetos físicos están actualizadas y se puede obtener dicha información para utilizarla con el propósito que creamos conveniente.

2.4 Simulación de vehículos con Havok

La simulación física del vehículo es un importante aspecto del proyecto, por lo que le prestaremos especial atención a este apartado.

Un subconjunto de Havok está dedicado a la simulación de vehículos, lo cual nos ayuda a la hora de desarrollar vehículos físicamente realistas. Incluye una serie de clases que definen e implementan el conjunto de comportamientos y algoritmos asociados a un coche y su conducción.

Como pudimos observar en el diagrama de clases de Havok, un vehículo (*hkpVehicleInstance*) se deriva de una acción (*hkpAction*), por lo que su comportamiento está regido por los componentes que lo conforman.

Los principales componentes que gobiernan el funcionamiento del vehículo son:

- *hkpVehicleInstance* es la clase principal que mantiene al resto de componentes.
- *hkpVehicleData* es un contenedor para las especificaciones del vehículo, tales como número de ruedas, el radio, masa y fricción de cada rueda, orientación del chasis el factor de balanceo que afecta al chasis del vehículo, etc.
- *hkpVehicleWheelCollide* maneja la detección de colisiones entre las ruedas y el terreno.
- Los demás componentes tienen una estructura similar y contienen datos usados para la configuración del vehículo. Cada uno contiene un método para calcular su efecto sobre el vehículo.

Figura 2.6. Componentes de un vehículo

2.4.1 Física del vehículo

En este apartado veremos los principales componentes que gobiernan funcionamiento del vehículo y los cuales tendremos que configurar para crearlo.

2.4.1.1 Sistema de coordenadas

El sistema de coordenadas de un vehículo define las posiciones y direcciones relativas al mismo.

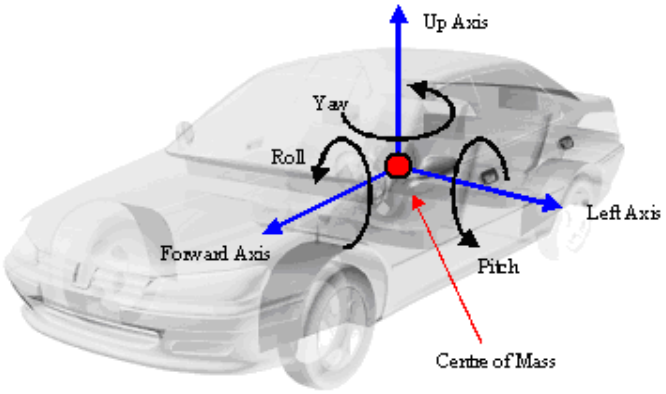

Figura 2.7. Sistema de coordenadas del vehículo

Para construir un vehículo es necesario proporcionar sus ejes de coordenadas.

2.4.1.2 Suspensiones

La suspensión del vehículo es un sistema mecánico que mantiene juntas las ruedas con el chasis. En general todos los sistemas de suspensión permiten a las ruedas:

- Rotar sobre su eje.
- Moverse hacia arriba y abajo dentro de un rango específico, normalmente siguiendo una línea recta.

Mediante un muelle, una barra estabilizadora y un amortiguador se controlan las fuerzas que actúan sobre las ruedas.

En Havok se hacen las siguientes suposiciones para simplificar el mecanismo de suspensión:

- El movimiento de la rueda se restringe a una línea recta en la dirección de la suspensión.
- La fuerza de la suspensión es independiente de la masa del vehículo.
- La suspensión se conecta al chasis del vehículo en una posición llamada punto de anclaje (hardpoint) y se extiende hasta el centro de la rueda.

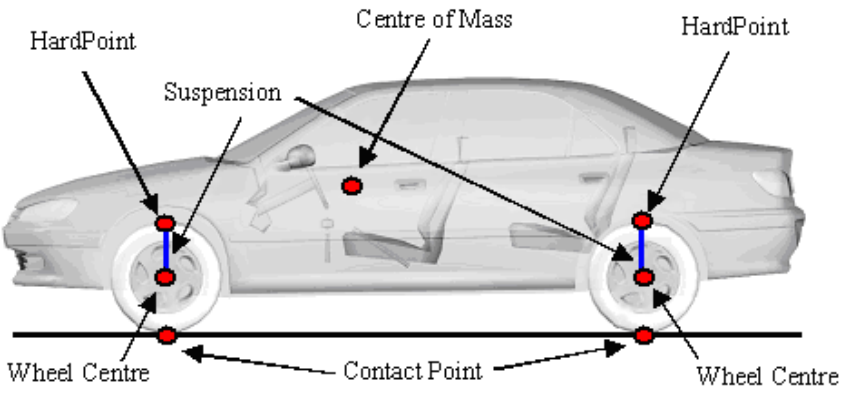

Figura 2.8. Suspensión del vehículo

2.4.1.3 Fricción

Cuando se toma una curva, el vehículo tiende a seguir la dirección actual. Sin embargo, la fricción entre las ruedas y el terreno fuerza al coche a seguir la dirección de la curva.

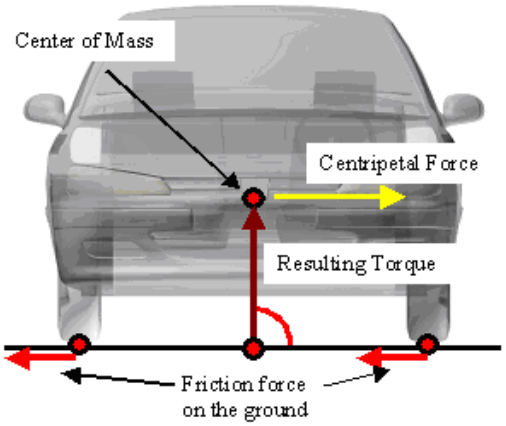

Figura 2.9. Fricción en el vehículo

Puesto que la fricción es aplicada sobre las ruedas y no sobre el centro de masa, el resultado es un balanceo del coche.

Existe un límite en el cual la fuerza de fricción no es suficiente y el coche empieza a deslizarse. Esta situación puede representarse mediante un círculo de fricción:

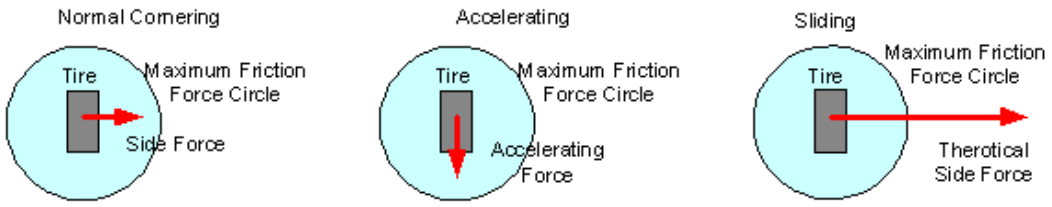

Figura 2.10. Círculo de fricción del vehículo

Cuando la fuerza que actúa sobre la rueda supera el límite del círculo de fricción tiene lugar el deslizamiento del vehículo.

Por este motivo hay que configurar adecuadamente el vehículo para lograr que sea suficientemente manejable.

2.4.1.4 Giro

El giro del volante se traduce en una rotación de las ruedas en el eje vertical. En la vida real existe una relación lineal entre la dirección del volante y el ángulo de giro de las ruedas. Sin embargo, a mayor velocidad girar el volante resulta más complicado. En Havok, la relación entre el giro de las ruedas y la velocidad viene dada por la siguiente figura:

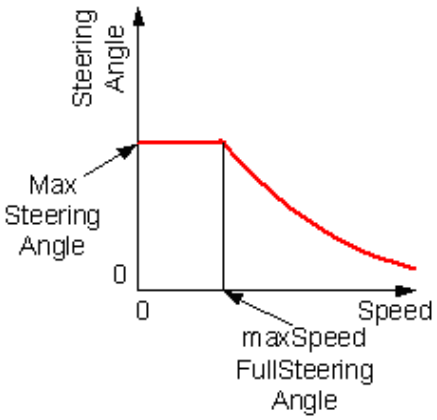

Figura 2.11. Relación entre giro y velocidad

Para obtener el comportamiento deseado del vehículo a alta velocidad hay que decidir el ángulo máximo de giro para velocidades bajas y jugar con el parámetro `maxSpeedFullSteeringAngle`.

2.4.1.5 Aerodinámica

El aire posee una cierta densidad, por lo que ejerce una influencia en un coche en movimiento. Estas influencias son conocidas como arrastre aerodinámico y sustentación aerodinámica. El arrastre crea una fuerza contraria al movimiento del coche, es el principal efecto aerodinámico en un vehículo y limita su velocidad punta.

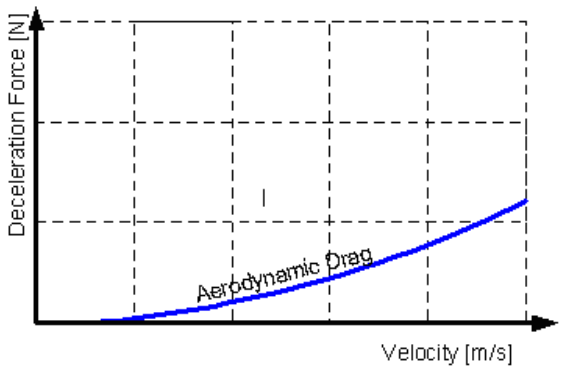

Figura 2.12. Aerodinámica del vehículo

2.4.1.6 Motor

El motor genera todas las fuerzas que permiten el movimiento del coche. En un motor común la fuerza neta entregada a las ruedas depende de la fuerza bruta del motor y la fricción del motor, que depende de las revoluciones por minuto (RPM).

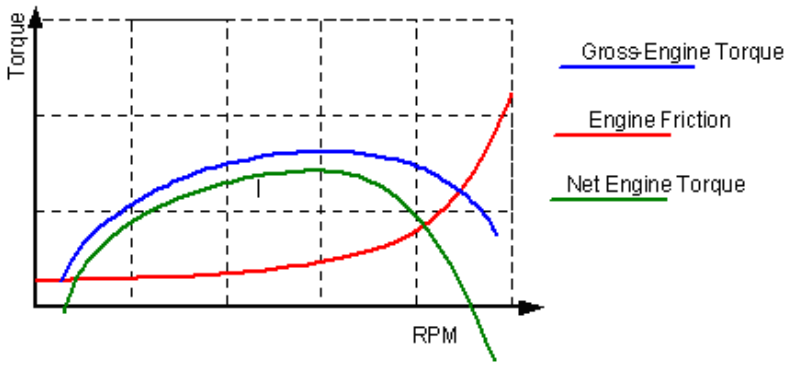

Figura 2.13. Fuerza del motor

Como resultado, el rendimiento del motor es óptimo dentro de un pequeño rango de RPM.

En Havok se utilizan los siguientes parámetros para simular el motor del vehículo:

- RPM/Fuerza mínima: Si el motor funciona por debajo de este punto, automáticamente se baja una marcha para mantener el motor dentro un rango de RPM adecuado.
- RPM/Fuerza óptima: En este punto el motor produce la fuerza máxima.
- RPM/Fuerza máxima: Define el tope de RPM del motor.

Con estos parámetros se puede simular el motor mediante el siguiente esquema:

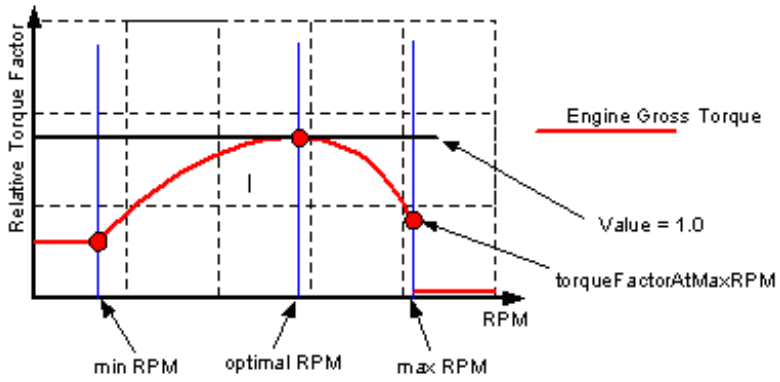

Figura 2.14. Fuerza del motor en Havok

2.4.1.7 Transmisión

Para mantener al motor trabajando en una región optima de RPM a diferentes velocidades, se usa la transmisión del vehículo. La transmisión utiliza las marchas para cambiar el ratio entre las revoluciones del motor y las revoluciones de las ruedas.

Una transmisión típica incluye el embrague, la caja de cambios y el diferencial. El embrague desconecta al motor de la caja de cambios cuando se cambia una marcha. La caja de cambios tiene varias marchas que trasladan la fuerza del motor al palier. El diferencial traslada y distribuye la fuerza desde el palier a las ruedas motrices.

Havok proporciona una transmisión automática, que incluye los siguientes parámetros:

- El ratio de cada marcha.
- El ratio del diferencial.
- Las revoluciones mínima y máxima cuando se debe cambiar de marcha.
- El tiempo que se tarda en cambiar una marcha.
- El ratio de distribución de las fuerzas entre las ruedas.

2.4.1.8 Frenos

Los frenos detienen el vehículo usando un mecanismo que convierte la energía cinética en calor a través de la fricción.

Havok multiplica el valor de entrada del pedal de freno por la fuerza máxima de frenada. La fuerza resultante se distribuye entre las ruedas.

2.4.1.9 Utilización de vehículos de Havok

Para utilizar un vehículo habrá que seguir los siguientes pasos:

- Crear el cuerpo rígido que represente el chasis del vehículo.
- Crear una instancia de *hkpVehicleInstance* y configurar sus componentes.
- Configurar y añadir al mundo físico la forma fantasma que representa las ruedas.
- Llamar al método `init()` de *hkpVehicleInstance* y añadir la instancia al mundo físico como una acción.

Una vez añadido el vehículo a la simulación física es necesario obtener la entrada del usuario y aplicársela al vehículo y actualizar la cámara para seguirlo.

2.4.2 El depurador visual de Havok

El depurador visual conecta a un PC (cliente) directamente al juego ejecutándose en una plataforma referida como servidor (PC o videoconsola) y muestra una versión gráfica de la simulación física de Havok. La posibilidad de visualizar la simulación física independientemente de nuestra representación gráfica facilita la tarea de encontrar y corregir errores tanto en el apartado físico como en nuestra solución para los gráficos.

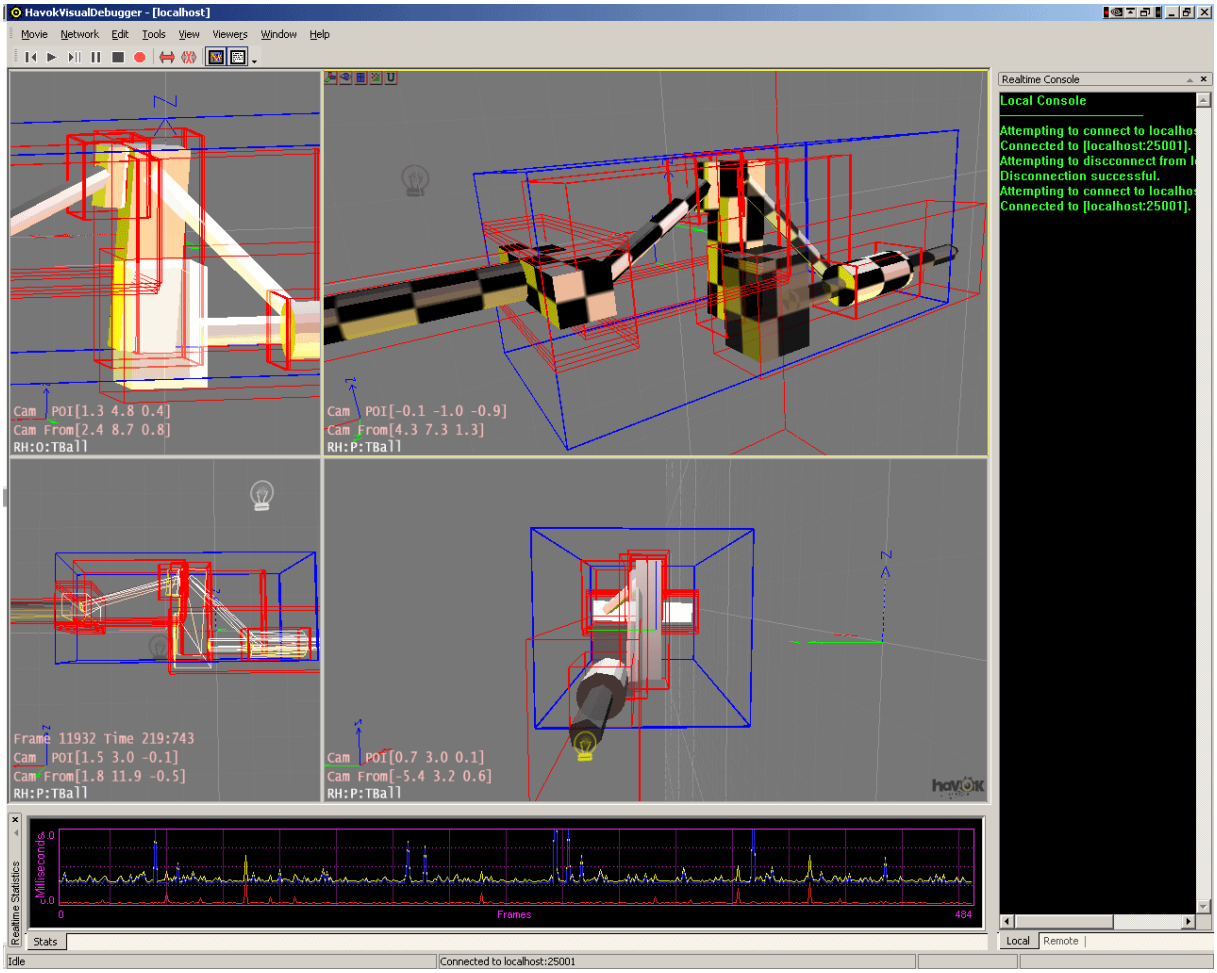

Figura 2.15. Depurador visual de Havok

2.5 Integración de gráficos y física

Cuando se utiliza un motor físico en un juego queremos crear la impresión de que la simulación y el dibujo de los objetos ocurren de manera continua. Sin embargo, en la práctica esto se traduce en la necesidad de realizar varias simulaciones y renderizaciones por segundo. Suponiendo que todo funciona correctamente, en cada uno de esos instantes tenemos que avanzar en el tiempo la simulación física, obtener los resultados y actualizar de manera acorde su representación gráfica.

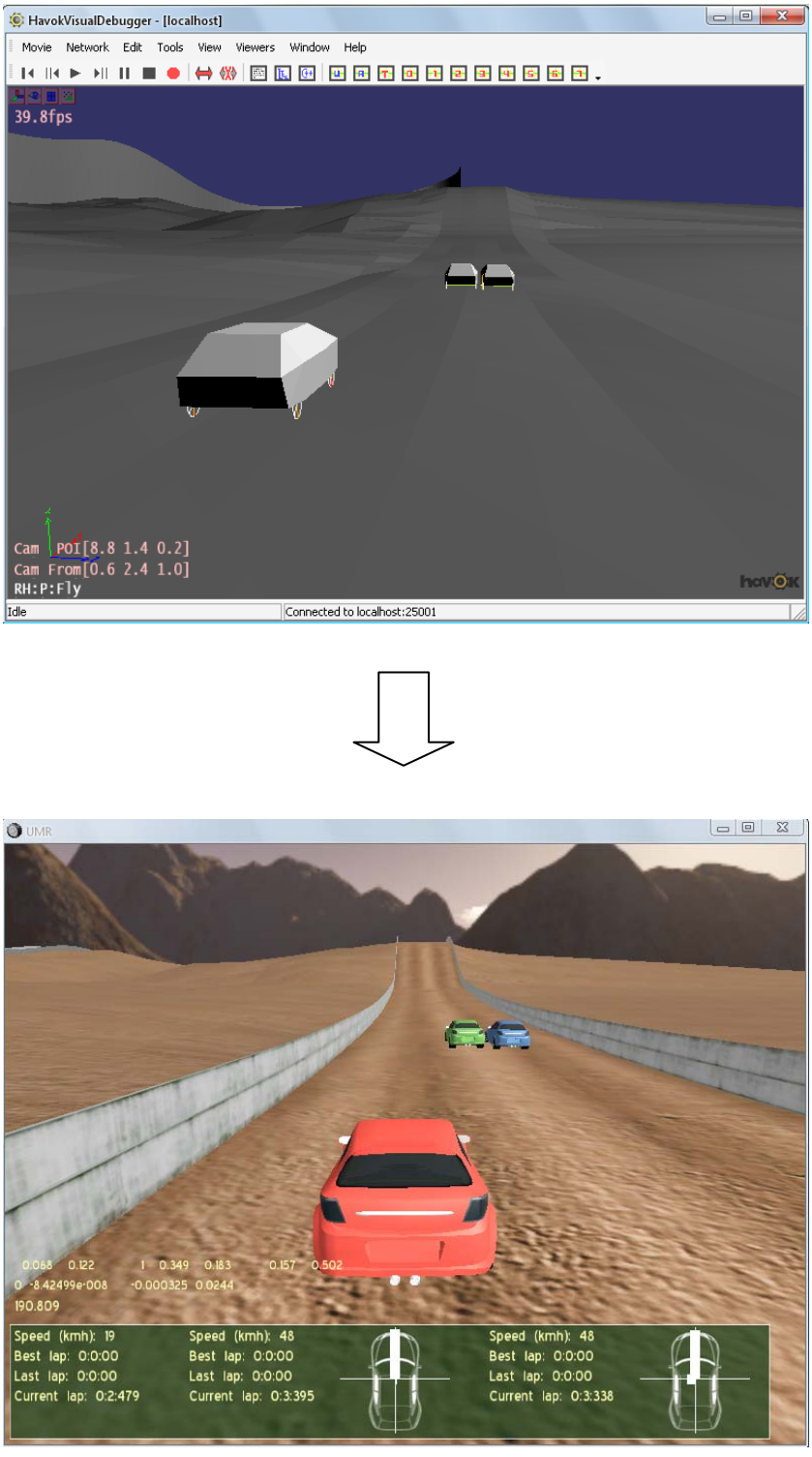

Figura 2.16. Integración de física y gráficos

Para conseguir una integración correcta y coherente entre el motor físico y el gráfico tendremos que partir de los mismos objetos. Es decir, el vehículo tiene que tener las mismas dimensiones tanto en la representación física como en la representación gráfica. Lo mismo

ocurre con el circuito. Para ello es necesario crear lo que se conoce como cauce de contenido que define el proceso que sigue un objeto desde su diseño hasta su utilización en el juego.

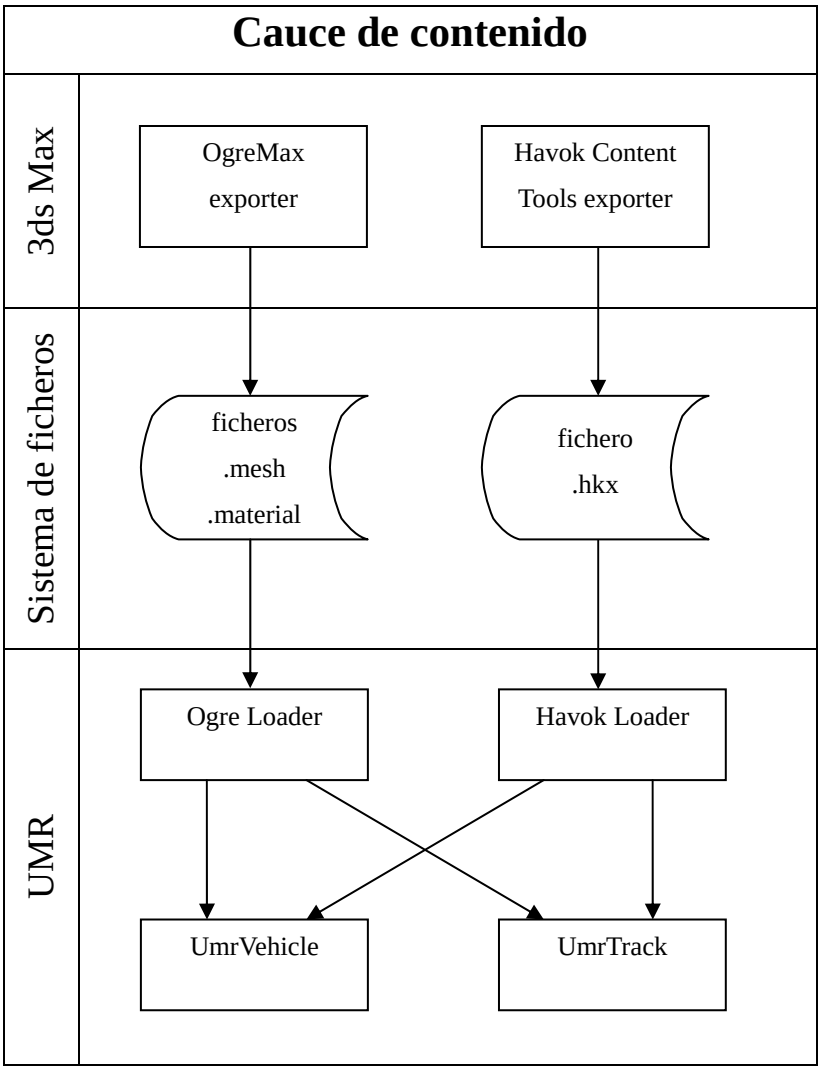

Figura 2.17. Cauce de contenido

Para obtener más información acerca del apartado gráfico se puede consultar los capítulos posteriores de análisis, diseño e implementación de la aplicación.

Capítulo 3

Aprendizaje automático

3.1 Aprendizaje automático en videojuegos

El aprendizaje automático es una rama de la inteligencia artificial cuyo objetivo es desarrollar técnicas que permitan a las computadoras aprender basándose en datos empíricos tales como sensores o bases de datos. Se trata de crear programas capaces de generalizar comoportamientos a partir de una información no estructurada, que en la mayoría de los casos es suministrada en forma de ejemplos.

Salvo en algunos ejemplos, se han lanzado pocos videojuegos comerciales que contuvieran algún nivel de aprendizaje y algunas de las razones incluyen:

- Hasta hace poco, la falta de precedentes en la aplicación exitosa de aprendizaje automático en videojuegos populares se traducía en que no era una tecnología madura con la consiguiente percepción de un alto riesgo en su uso.
- El aprendizaje automático resulta complicado de incluir en un juego debido a su relativa baja eficiencia comparado con otras técnicas explícitas y procedurales.

A pesar de esto, el aprendizaje automático puede ofrecer beneficios tanto a desarrolladores de videojuegos como a los jugadores en sí. A través del aprendizaje automático y con una supervisión mínima se pueden descubrir soluciones a problemas que son extremadamente complejos de resolver manualmente, evitando tener que utilizar un enorme conjunto de complejas reglas.

En general se distinguen tres paradigmas de aprendizaje automático:

- **Aprendizaje supervisado**. Produce una función que establece una correspondencia entre las entradas y las salidas deseadas.
- **Aprendizaje no supervisado**. El modelado se lleva a cabo sobre un conjunto de

entradas al sistema para realizar una agrupación (clustering).

- **Aprendizaje por refuerzo**. El algoritmo aprende observando el mundo que le rodea. Cada acción tiene un impacto en el entorno y este proporciona una realimentación que guía el aprendizaje.

Más concretamente se utilizan métodos tales como árboles de decisión, algoritmos genéticos o redes neuronales, de entre los cuales utilizaremos el último método por diversos motivos:

- Proporciona todos los beneficios del aprendizaje automático mediante una solución simple.
- Resulta apropiada para aprender complejas funciones y valores continuos.
- Es capaz de aprender incluso con datos imprecisos.
- Amplia bibliografía, ya que es una de las técnicas más utilizadas en videojuegos. Dispone además de un conjunto de herramientas para su diseño y utilización.

3.2 Desarrollo de la red neuronal

El uso de redes neuronales es un importante apartado del proyecto, por lo que se hace necesario abordar su desarrollo mediante un proceso iterativo que nos permita ir afinando el funcionamiento de dicha red neuronal. En nuestro caso dicho proceso se lleva a cabo mediante cuatro iteraciones, cada una de las cuales cubre el análisis, diseño e implementación correspondiente de la red neuronal.

3.2.1 Análisis red neuronal

El objetivo de la red neuronal es proporcionar un método para dotar a un agente inteligente de un comportamiento aproximado al de un jugador humano, por lo cual dicho agente deberá disponer de sus mismos medios para controlar el vehículo: el giro del volante y la utilización del acelerador y el freno.

Un aspecto importante a la hora de desarrollar el agente es la capacidad de realizar su trabajo sin información previa de un circuito, utilizando sólo información en tiempo real del estado del vehículo y su entorno. Para poder muestrear esta información de estado deberemos dotar al vehículo de una serie de sensores, los cuales se corresponderán con las entradas de la red

neuronal. Por lo tanto, la elección de unos sensores que aporten la mayor cantidad de información será un elemento clave.

Una vez conocidos los sensores podemos desarrollar un esquema para el sistema de inteligencia artificial que se ajuste al siguiente:

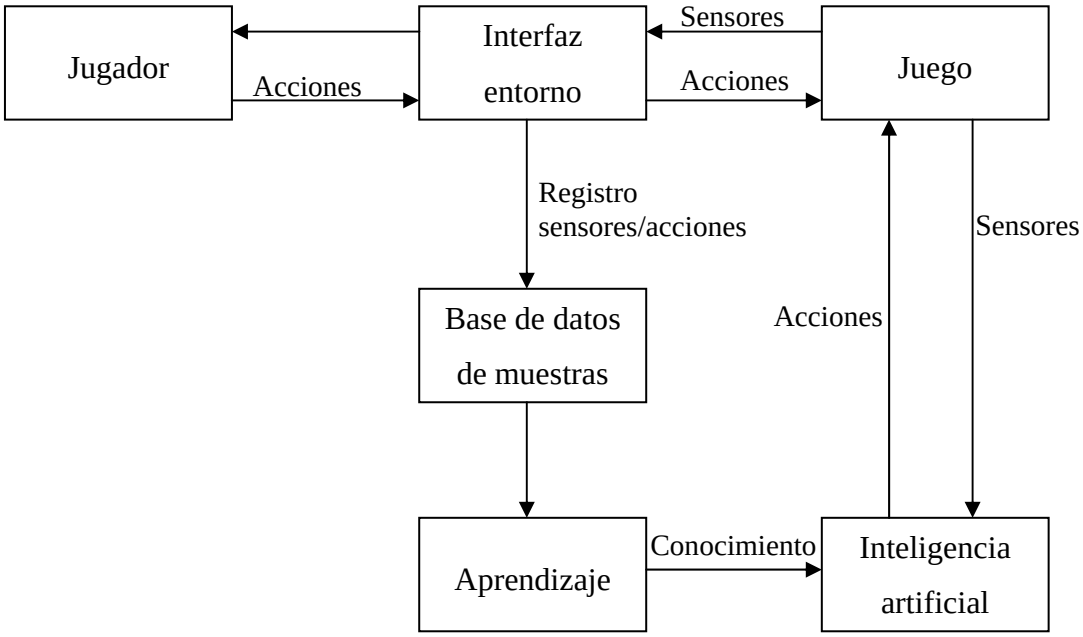

Figura 3.1. Visión general del sistema de inteligencia artificial

El diseño de la red neuronal nos debe aportar una visión general de la misma, que utilizaremos a la hora de realizar su implementación. Así mismo también tenemos que diseñar los sensores que incorporaremos al vehículo, los cuales podrán ser diferentes en iteraciones sucesivas, y que nos proporcionarán la información de estado del vehículo y su entorno.

3.2.3 Representación del entorno: Sensores

Una correcta representación del entorno a través de sensores es primordial a la hora de conseguir buenos resultados, por lo que elegir un buen diseño es trascendental.

Las salidas de la red neuronal nos proporcionan una información importante de cara a determinar el estado del vehículo, por lo que resulta razonable incluir el **giro** del volante, la **aceleración** y la frenada como sensores. Sería conveniente mantener esta información en tan

sólo dos sensores, por lo que la aceleración y la frenada se condensarán en una variable. Una manera de conseguir esto es mapear dichos sensores sobre un eje de coordenadas de dos dimensiones. De esta manera el giro a la izquierda se correspondería a valores negativos, mientras que el giro a la derecha resultaría en valores positivos. De igual forma, la presión del pedal de aceleración se relaciona con valores positivos, mientras que los valores negativos representan la presión del pedal de freno.

Figura 3.2. Mapeo de la aceleración y el giro

La **velocidad** actual del vehículo también proporciona información pertinente, de manera que se incluye en la lista de sensores.

Así mismo, y debido a la naturaleza de los circuitos que se pretenden desarrollar, mantener información del estado de **deslizamiento** del vehículo podría resultar provechoso.

Un sensor a tener en cuenta es la dirección del vehículo, es decir, si el vehículo sigue el **sentido correcto** del circuito. Será necesario estudiar si su inclusión mejora o empeora los resultados.

Una de las principales fuentes de información son los sensores utilizados para medir distancias. Dichos sensores tendrán la forma de **rayos** lanzados desde un punto concreto, relativo a la posición del vehículo, hasta una distancia prudencial siguiendo una línea recta.

El método más adecuado es lanzar los rayos formando un abanico, de forma que podamos muestrear la distancia a los bordes de la carretera de manera uniforme y simétrica.

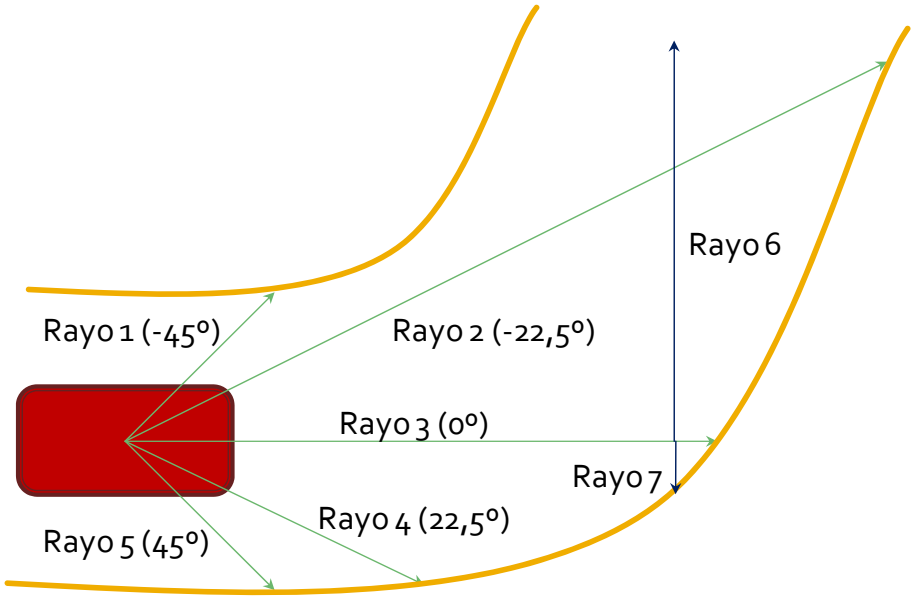

Figura 3.3. Sensores en forma de rayos

De esta forma tenemos 5 rayos, cada uno se lanza formando un ángulo determinado con la dirección del vehículo.

Un caso especial son los rayos 6 y 7 de la imagen anterior, los cuales se originan justo antes del próximo obstáculo y que permitirá, en su caso, detectar la dirección de la siguiente curva.

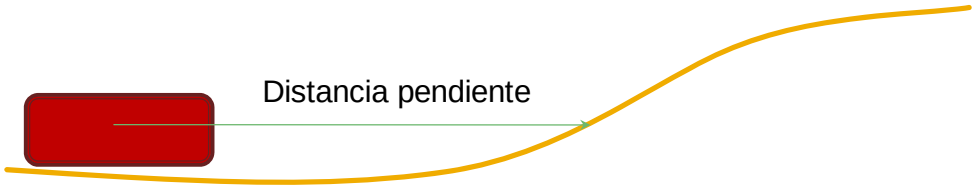

Figura 3.4. Sensor de distancia a pendiente

Así mismo puede resultar interesante tener información acerca de la **distancia a la próxima pendiente**, de manera que este factor pueda influenciar en el comportamiento final.

Un sensor que puede resultar más adecuado que el anterior es el grado de **inclinación** del terreno, por lo que es necesario realizar una comparación de ambos.

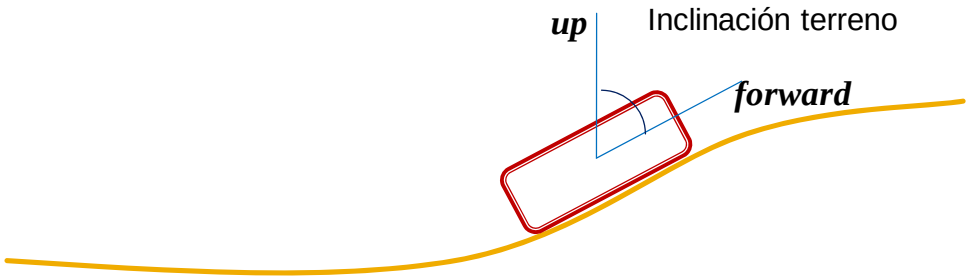

Figura 3.5. Sensor de inclinación del terreno

Cuando el vehículo se mueve a alta velocidad y requiere cambiar su dirección, el vector de dirección del vehículo resultará diferente al vector de velocidad, lo que llevará a aplicar una **diferencia** extra en la **dirección** de giro para contrarrestar la situación. Esto nos puede proporcionar información acerca del tipo de curva en la que se encuentra el vehículo (lenta o rápida), así como el estado de deslizamiento lateral del vehículo.

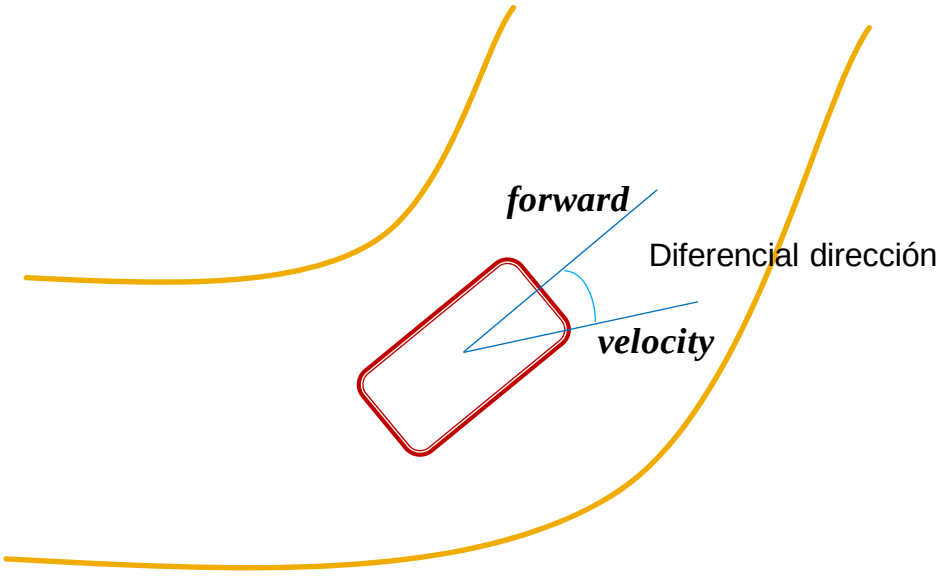

Figura 3.6. Sensor de diferencial de dirección

3.2.4 Diseño de la red neuronal

El proceso de diseño de la red neuronal conlleva elegir los sensores de entrada y las salidas de la red neuronal, tras lo cual podemos implementar la red neuronal en cuestión y realizar su entrenamiento para evaluar los resultados.

El tipo de red neuronal más adecuado es un perceptrón multicapa compuesto por una capa de unidades de entrada (sensores), otra capa de unidades de salida y un número determinado de capas intermedias de unidades de proceso.

3.2.4.1 Primera iteración

Para la primera aproximación optamos por alimentar a la red neuronal con la mayor información posible de forma que cada situación pueda quedar representada por el estado de los sensores.

La red neuronal quedó determinada por once variables de entrada y dos de salida, siendo las variables de entrada las siguientes:

- Giro previo.
- Aceleración previa.
- Velocidad.
- Deslizamiento del vehículo.
- Rayo 1 (-45º).
- Rayo 2 (-22.5º).
- Rayo 3 (0º).
- Rayo 4 (22.5º).
- Rayo 5 (45º).
- Distancia a pendiente.
- Dirección equivocada.

Lo que se corresponde a la siguiente arquitectura de la red:

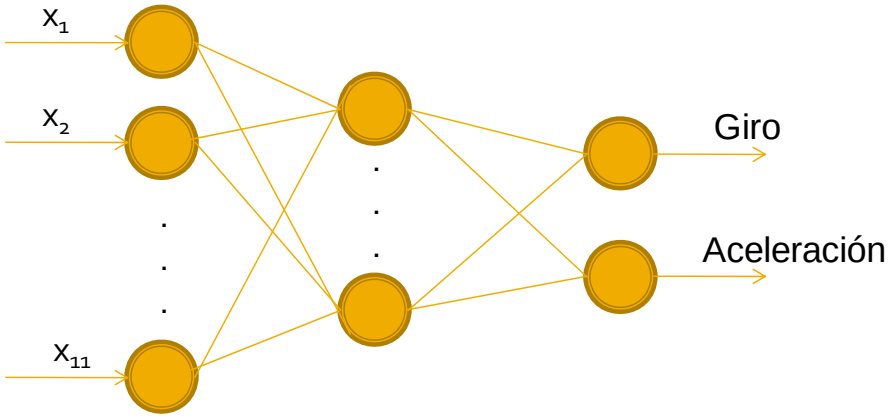

Figura 3.7. Primera red neuronal

3.2.4.2 Segunda iteración

La utilización de una sola red neuronal para dos funciones no resultó una buena decisión de diseño porque existireron interferencias al intentar aprender dos funciones distintas, lo que dificultó el aprendizaje. Por lo tanto se utiliza una aproximación diferente y se utilizó una red neuronal para cada salida con las mismas entradas que en la iteración anterior.

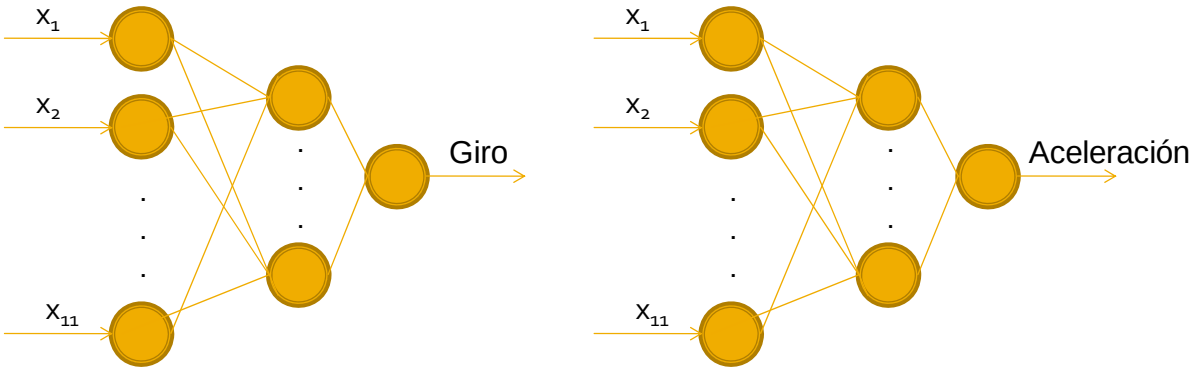

Figura 3.8. Segunda red neuronal

3.2.4.3 Tercera iteración

Haciendo un análisis de las posibles dependencias de las entradas y las salidas, podemos ver

que en las redes anteriores algunas entradas proporcionaban información innecesaria. Por lo tanto pensamos que reducir las entradas y utilizar sólo las necesarias proporcionaría resultados más adecuados.

Cuando un vehículo se conduce lentamente, con la velocidad justa en curva para no perder el control del coche, se puede prever que el giro del volante dependerá casi exclusivamente de la distancia a los bordes de la carretera, de forma que siempre se intenta conducir lo más alejado posible de dichos bordes, o lo que es lo mismo, conducir por el centro de la carretera. En esta tercera iteración se tuvo en cuenta este factor y se usó la información que teniamos sobre la distancia a la carretera a ambos lados del vehículo para diseñar la red neuronal encargada de aprender el giro del volante.

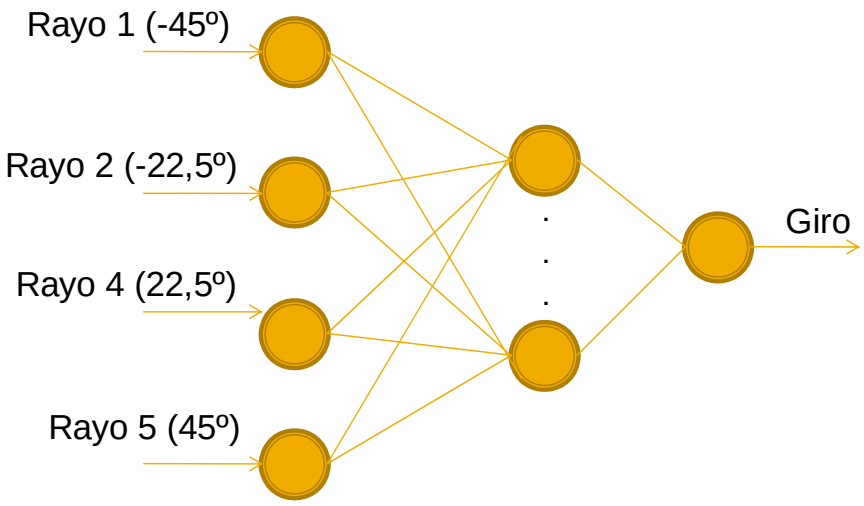

Figura 3.9. Tercera red neuronal – Giro

En el caso de la aceleración la situación cambió respecto al giro. En este caso, el modo habitual de proceder es acelerar al máximo cuando no hay obstáculos por delante. Cuando se detecta un obstáculo o una curva, lo habitual es frenar para tomar la curva con una velocidad prudente y mantener dicha velocidad hasta salir de la curva. En la detección de estas situaciones influyeron los sensores utilizados para la red neuronal que aprende la aceleración del vehículo.

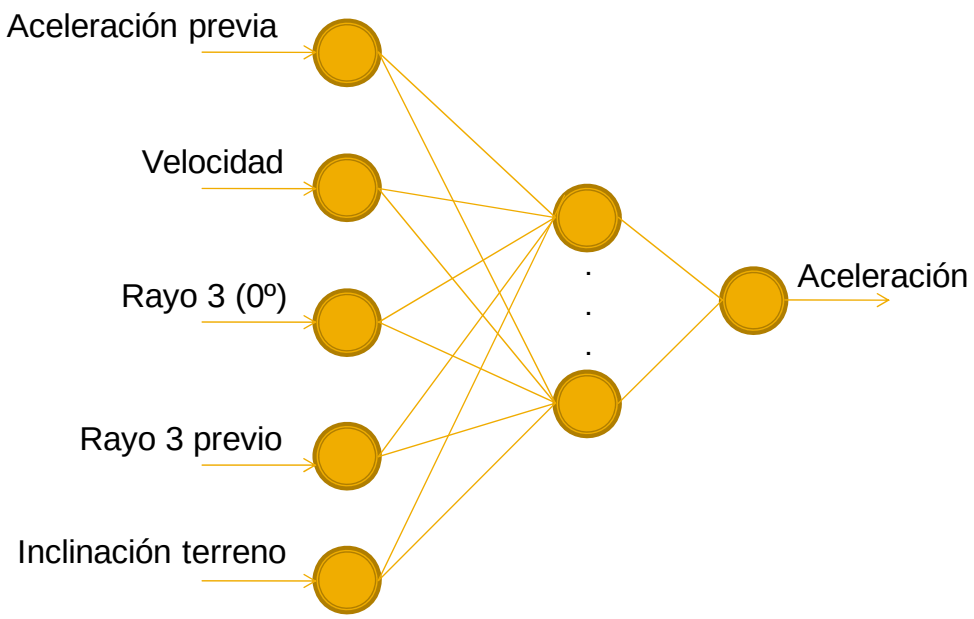

Figura 3.10. Tercera red neuronal – Aceleración

3.2.4.4 Cuarta iteración

Cuando conducimos un vehículo a alta velocidad, el número de factores que influyeron en las decisiones de conducción incrementan. En este caso la información utilizada en la iteración anterior resultó insuficiente, por lo que fue necesario diseñar dos nuevas redes neuronales para modelar las nuevas situaciones.

En una conducción a alta velocidad resultó adecuado mantener información respecto a dos momentos consecutivos. Este es el caso de los sensores de distancia o rayos. Incluir estos sensores aportó información valiosa acerca de los cambios en la posición del vehículo con respecto a los obstáculos, así como el ritmo o velocidad de dichos cambios.

Puesto que el vehículo tiene un comportamiento diferente a alta y baja velocidad, fue imprescindible incorporar la velocidad como entrada para las redes neuronales.

Así mismo la diferencia entre la dirección del vehículo y la dirección de su velocidad fue un importante factor a tener en cuenta. Esta diferencia era distinta de cero en curvas tomadas a alta velocidad, y nos indicó el deslizamiento lateral del vehículo, por lo que sin duda afectó al giro del volante.

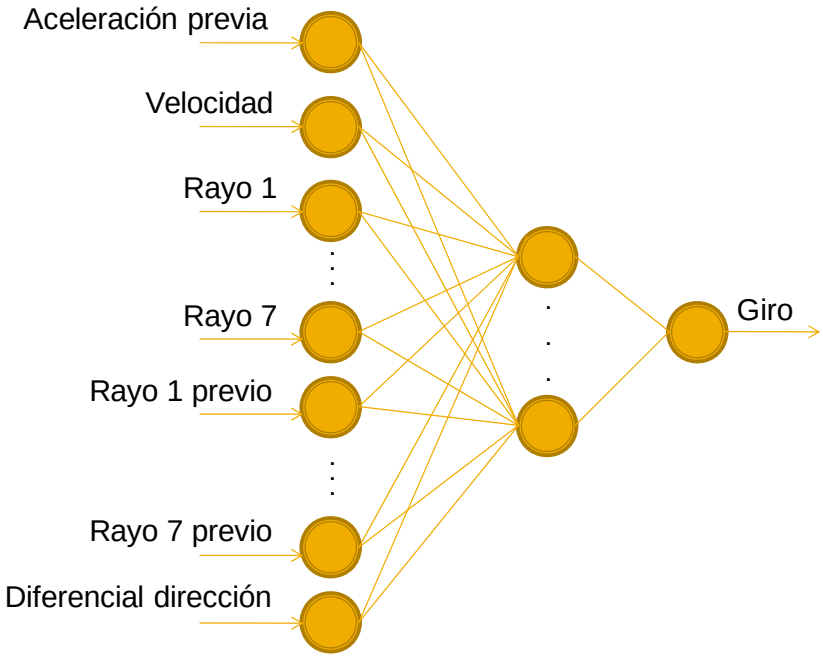

Figura 3.11. Cuarta red neuronal - Giro

En el modelado de la aceleración influyeron prácticamente los mismos factores, aunque en este caso el giro de volante intervino también al decidir el grado de aceleración.

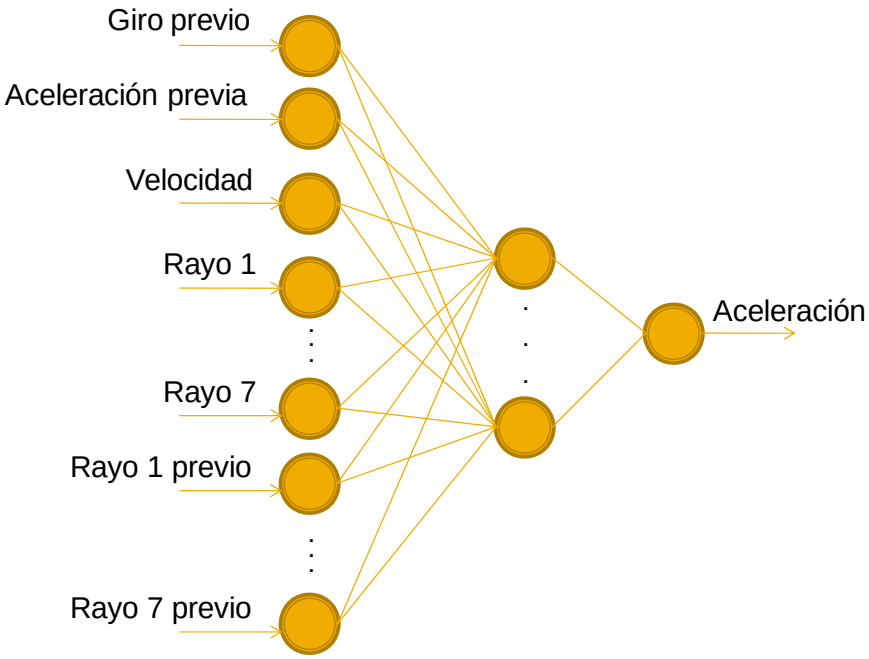

Figura 3.12. Cuarta red neuronal - Aceleración

En los siguientes capítulos se documenta el análisis, diseño e implementación de todo este proyecto software.

Capítulo 4

Requisitos de la aplicación

En este capítulo se documentan los requisitos y el análisis de la aplicación. Como en todo proyecto, al principio hay que establecer el ámbito del mismo. Una vez fijado el ámbito, se puede determinar de manera precisa qué se espera desarrollar. Esto se consigue creando una lista de requisitos que debe cumplir el software. Esta lista identifica específicamente la funcionalidad que se incluye dentro del ámbito establecido.

Como resultado de esta fase se obtendrá la definición del ámbito del proyecto, así como el catálogo de requisitos y los casos de uso del sistema.

4.1 Ámbito

El ámbito quedó establecido cuando definimos los objetivos del presente proyecto fin de carrera. Se pretende desarrollar un juego de carreras en 3D para la plataforma PC, concretamente para entornos Windows. Se permitirá a un jugador competir contra varios adversarios autónomos. Tanto la simulación física como la inteligencia artificial del juego deben ofrecer un compromiso entre realismo y diversión.

4.2 Catálogo de requisitos

4.2.1 Requisitos funcionales

RF1. El usuario podrá elegir la configuración gráfica de la aplicación.

RF2. El usuario podrá cerrar la aplicación en cualquier momento.

RF3. El usuario podrá correr con un vehículo en más de un circuito.

RF4. El vehículo se podrá controlar con el giro del volante, el acelerador y el freno.

RF5. El usuario podrá configurar las teclas usadas para cada acción.

RF6. El usuario podrá manejar el vehículo con el teclado o un joystick.

RF7. El usuario verá en todo momento la velocidad del vehículo.

RF8. El usuario verá en todo momento el tiempo de su mejor vuelta.

RF9. El usuario verá en todo momento el tiempo de su última vuelta.

RF10. El usuario verá en todo momento el tiempo de su vuelta actual.

RF11. El usuario podrá competir contra más de un vehículo.

RF12. El usuario verá en todo momento las acciones que realizan los adversarios.

RF13. El usuario podrá ver la velocidad de sus adversarios.

RF14. El usuario podrá ver los tiempos de sus adversarios.

RF15. Los vehículos reaparecerán en una posición segura cuando se queden bloqueados.

RF16. El vehículo podrá ser controlado por el usuario o por un agente inteligente.

4.2.2 Requisitos no funcionales

RNF1. La aplicación será multihebra.

RNF2. Los objetos de la escena serán tridimensionales.

RNF3. La cámara seguirá al vehículo del usuario.

RNF4. Los circuitos tendrán desniveles, saltos y curvas de distinta dificultad.

RNF5. Los circuitos serán de material deslizante.

RNF6. El comportamiento físico de los vehículos será realista.

RNF7. El vehículo será fácilmente manejable.

RNF8. Los agentes inteligentes estarán controlados por redes neuronales.

RNF9. La aplicación permitirá registrar datos de estado del vehículo.

RNF10. La aplicación estará desarrollada en C++.

RNF11. La aplicación utilizará el motor gráfico Ogre3D.

RNF12. La aplicación utilizará el motor físico Havok.

4.3 Diagramas de casos de uso

Entre los requisitos funcionales definidos se distinguen dos actores principales, el usuario de la aplicación y el vehículo del juego, que puede estar controlado bien por el usuario o bien por un agente inteligente. Por lo tanto se construye un diagrama de casos de uso diferente para cada actor.

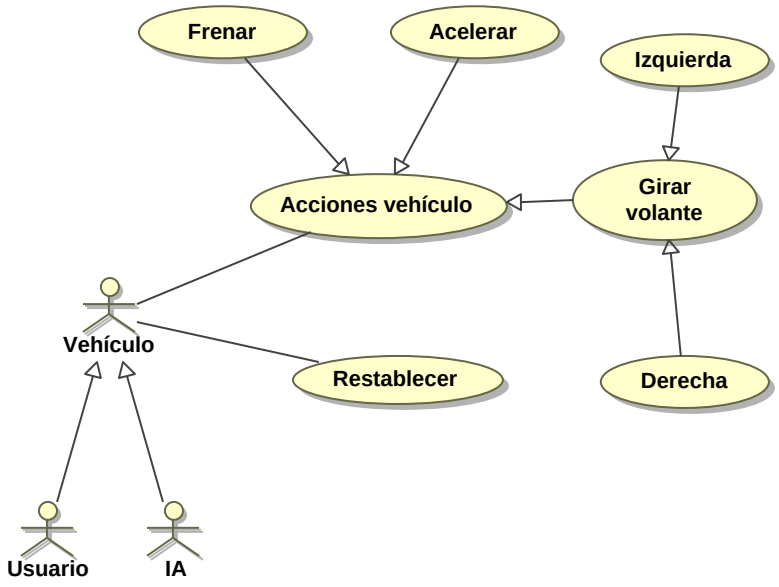

Figura 4.1. Diagrama de casos de uso del vehículo

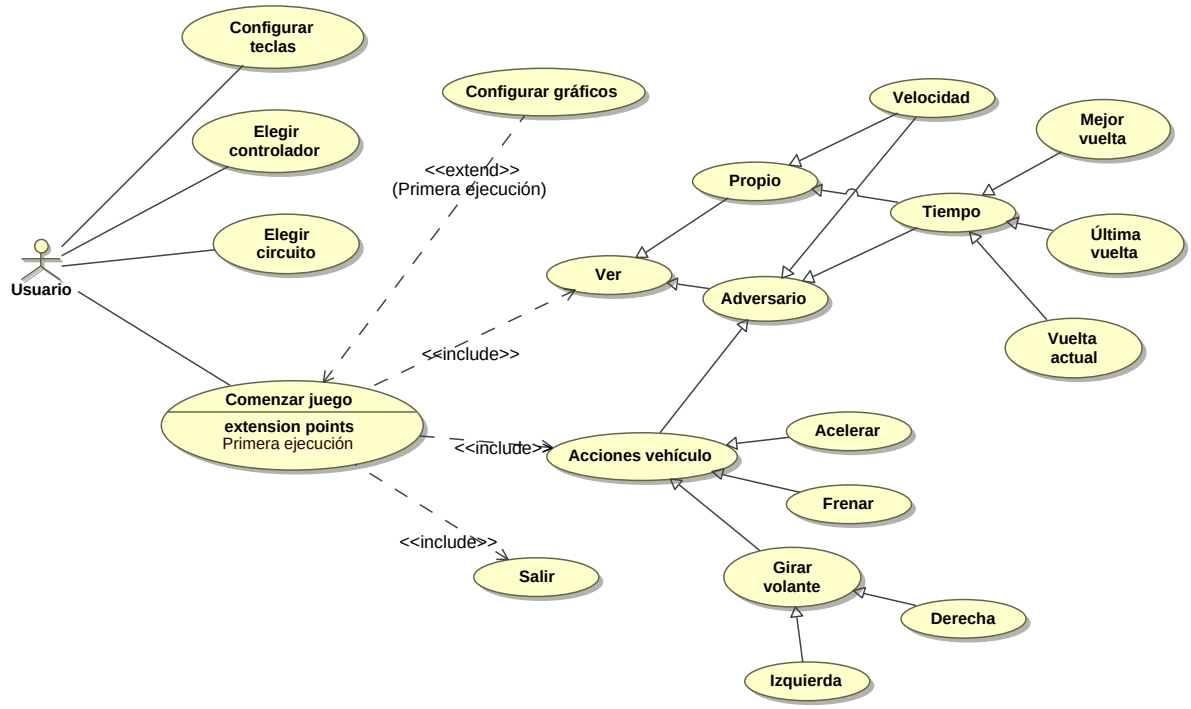

Figura 4.2. Diagrama de casos de uso del usuario

4.4 Casos de uso textuales

Los siguientes son los casos de uso más significativos del sistema:

Caso de uso	Comenzar juego
Actor	Usuario
Descripción	El usuario ejecuta el juego para comenzar a jugar
Precondiciones	El juego está instalado y listo para ejecutar.
Evento de disparo	El usuario inicia la aplicación.
Secuencia normal	1. El usuario inicia la aplicación. 2. El sistema carga los contenidos del juego. 3. El usuario realiza acciones sobre el vehículo. 4. El sistema actualiza el estado de los vehículos.

	5. Se vuelve al paso 3 hasta que el usuario finaliza el juego.
Excepciones	2a. Si es la primera vez que se ejecuta el juego o no existe un perfil gráfico seleccionado, el sistema muestra los perfiles gráficos disponibles. 2b. El usuario elige un perfil gráfico. 2c. El sistema carga los contenidos del juego.

Caso de uso	Ver velocidad del vehículo
Actor	Usuario
Descripción	El usuario ve en la interfaz del juego la velocidad del vehículo.
Precondiciones	El usuario ha iniciado el juego.
Evento de disparo	El sistema actualiza la interfaz periódicamente.
Secuencia normal	1. El sistema actualiza la interfaz y el estado de los vehículos. 2. El usuario consulta la velocidad del vehículo en la interfaz. 3. Se vuelve al paso 1 hasta que el usuario finaliza el juego.
Excepciones	

Caso de uso	Acelerar el vehículo
Actor	Usuario
Descripción	El usuario acelera el vehículo para aumentar la velocidad.
Precondiciones	El usuario ha iniciado el juego.
Evento de disparo	El usuario presiona la tecla asociada al acelerador.
Secuencia normal	1. El usuario presiona la tecla asociada al acelerador. 2. El sistema aumenta el nivel de aceleración del vehículo.

Excepciones	2a. Si el acelerador está presionado al máximo, el sistema mantiene la aceleración.

Caso de uso	Restablecer vehículo
Actor	Vehículo
Descripción	El vehículo se restablece en una posición segura.
Precondiciones	El vehículo está bloqueado.
Evento de disparo	El vehículo detecta que está bloqueado.
Secuencia normal	1. El vehículo detecta que está bloqueado. 2. El sistema restablece en una posición segura.
Excepciones	

Capítulo 5

Análisis y diseño de la aplicación

Para analizar y diseñar el proyecto partimos de la información recolectada durante la fase anterior. Esta información nos va a permitir definir una colección de entidades software que expresan la funcionalidad que los requisitos especifican. Además de crear un grupo de entidades, estas se organizan de forma que puedan trabajar juntas.

El diseño del proyecto implica planificar cómo construir el software de acuerdo a los requisitos. Es necesario examinar los elementos del sistema para encontrar maneras de agruparlos de acuerdo a las relaciones que existen entre ellos.

En esta fase obtendremos una versión preliminar de la arquitectura del sistema y exploraremos, desde el punto de vista del diseño, el motor gráfico Ogre3D que utilizaremos posteriormente en el proyecto. Así mismo, con los conocimientos adquiridos en fases anteriores, creamos los contenidos necesarios para la aplicación. Por último, definiremos el diseño de las clases del sistema.

5.1 Arquitectura

Con el fin de reducir la complejidad del proyecto, durante el diseño se hace uso de una conceptualización a alto nivel de los elementos del sistema. La arquitectura nos proporciona ese conjunto de conceptualizaciones de lo que el sistema será una vez finalizado el proyecto.

La arquitectura nos muestra el principio y el final del proceso de diseño. Marca el principio porque nos proporciona una serie de supuestos que guían el diseño y marca el final porque nos muestra una visión general del sistema como una creación arquitectónica.

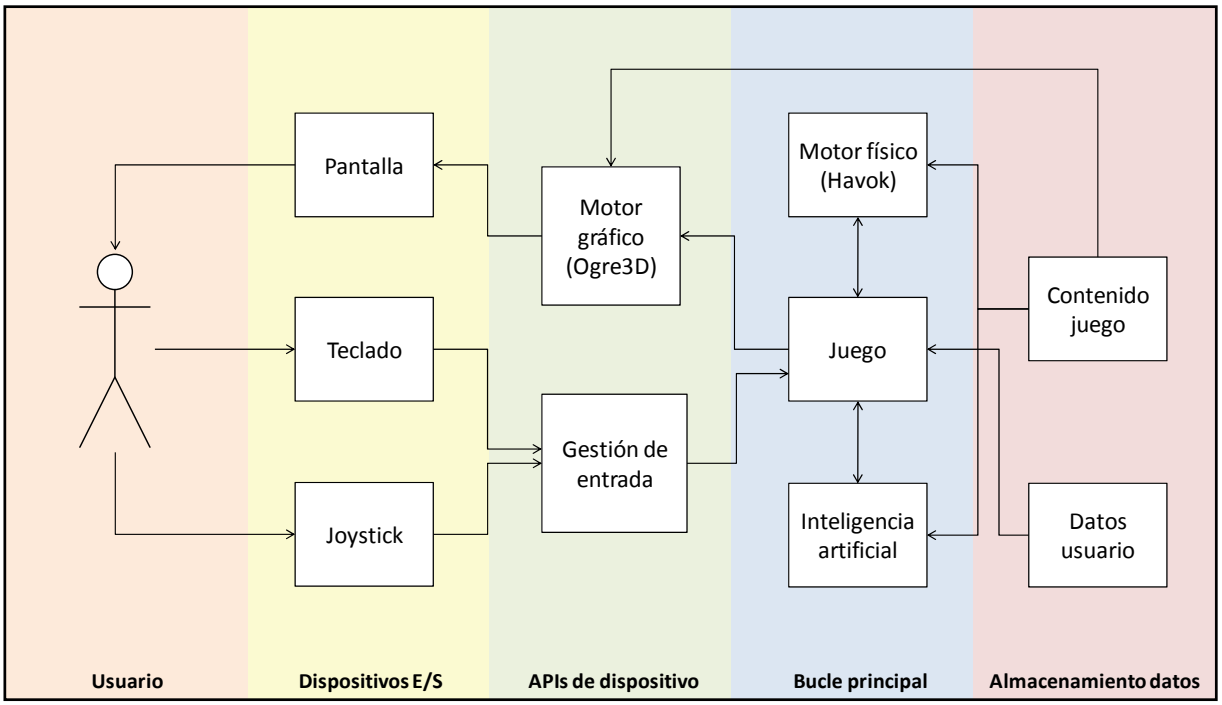

Figura 5.1. Arquitectura del sistema

En la arquitectura podemos distinguir varios componentes:

- **Motor gráfico**. Proporciona la capacidad de dibujar objetos 3D.
- **Gestión de entrada**. Permite utilizar dispositivos de entrada y transformar las entradas en acciones de control dentro del juego.
- **Motor físico**. Permite dotar a los vehículos de movimientos físicamente realistas.
- **Juego**. Contiene el bucle principal del juego y gestiona el estado del mismo.
- **Inteligencia artificial**. Gestiona el comportamiento de los agentes inteligentes del juego.
- **Contenido juego**. Representa el almacén de contenido estático del juego.
- **Datos usuario**. Representa el almacenamiento de datos de configuración asociados al usuario.

5.2 Ogre3D

OGRE (Object-Oriented Graphics Rendering Engine) es un motor gráfico 3D multiplataforma, estable, fiable, flexible, de código abierto y escrito en C++. Resulta apropiado para el desarrollo de aplicaciones gráficas de tiempo real.

Ogre3D está diseñado bajo la filosofía de la orientación a objetos, por lo que su interfaz es clara, intuitiva y fácil de usar. Su biblioteca de clases abstrae la complejidad de las capas inferiores, como OpenGL o Direct3D, y proporciona una interfaz basada en objetos más cercanos e intuitivos.

Entre las características más destacadas del diseño de Ogre3D podemos encontrar:

- Uso inteligente de los patrones de diseño más comunes.
- Desacoplamiento entre el grafo de escena y los contenidos de la escena.
- Arquitectura de plugins.
- Soporte de renderizado utilizando aceleración hardware.
- Arquitectura de renderizado flexible.
- Robusto sistema de materiales.
- Formato nativo y optimizado para geometría 3D.
- Múltiples tipos de animación.
- Composición de efectos de post-proceso.
- Sistema de gestión de recursos extensible.

El siguiente diagrama presenta una visión general del diseño Ogre3D, el cual tendremos que integrar más tarde en el juego:

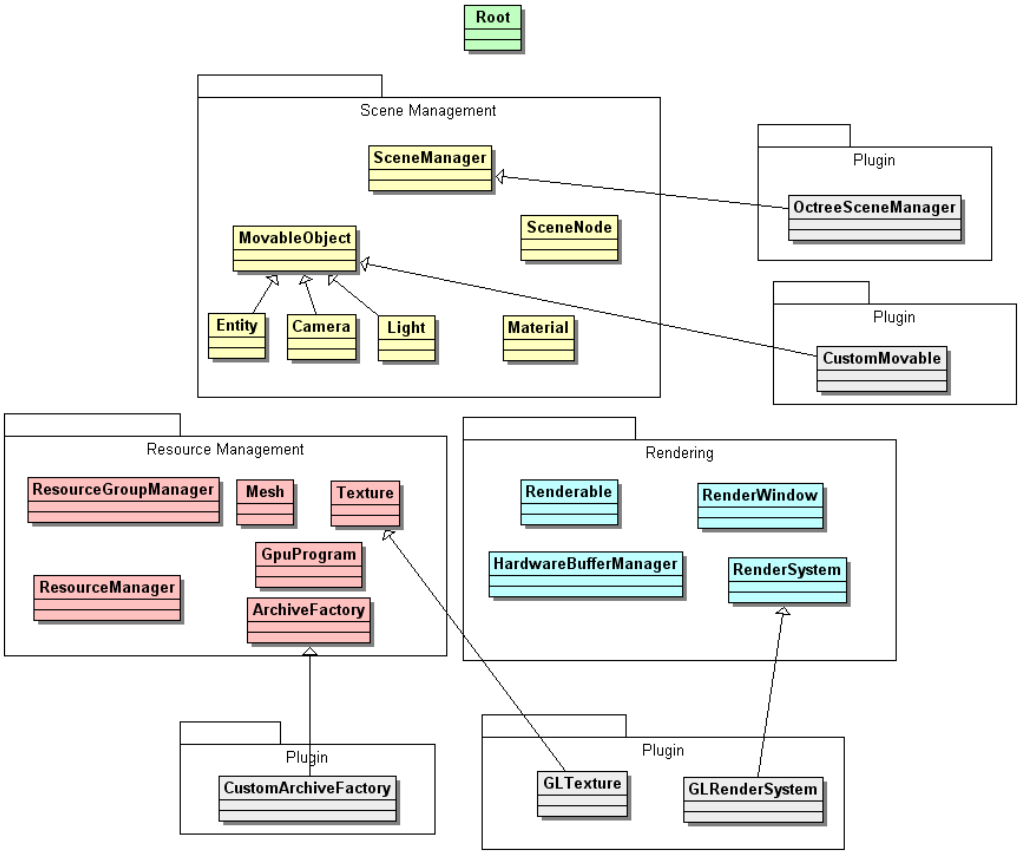

Figura 5.2. Diseño de Ogre3D

A continuación describiremos los elementos más comunes que tendremos que integrar y con los cuales deberemos interactuar en la mayor parte de las ocasiones.

5.2.1 Root

Es el principal punto de acceso al sistema. Utiliza el patrón de diseño Façade y proporciona acceso a los distintos subsistemas de Ogre3D.

5.2.2 Resource Management

Cualquier cosa que se necesite Ogre3D para dibujar una escena es conocida como recurso, los cuales están gestionados por *ResourceGroupManager*. Este objeto se encargará de localizar recursos e inicializarlos.

Entre los tipos de recursos conocidos y que vamos a utilizar, podemos encontrar:

- *Mesh*. Representa un objeto 3D. Utiliza un formato nativo, y típicamente estará

almacenado en un fichero con la extensión .mesh.

- *Material*. Define el aspecto de un objeto 3D. Normalmente están referenciados por un fichero .mesh y están contenidos en ficheros de script con extensión .material.
- *Texture*. Representa una textura 2D en cualquier formato reconocido por Ogre3D.
- *Overlay*. Permiten la posibilidad de superponer elementos 2D en la escena final. En la práctica están almacenados en scripts con extensión .overlay.
- *Font*. Representa la definición de fuente que Ogre3D utiliza en Overlays. Se almacenan en ficheros de script con extensión .font.

5.2.3 Scene Management

En Ogre3D el grafo de la escena está completamente desacoplado de su contenido. Sólo se opera con el grafo de escena a nivel de interfaz. A dicha interfaz sólo le interesa la estructura del grafo. Los nodos no contienen ninguna funcionalidad de acceso o de gestión.

La funcionalidad de gestión y acceso a contenidos se delega en los objetos *Renderable*.

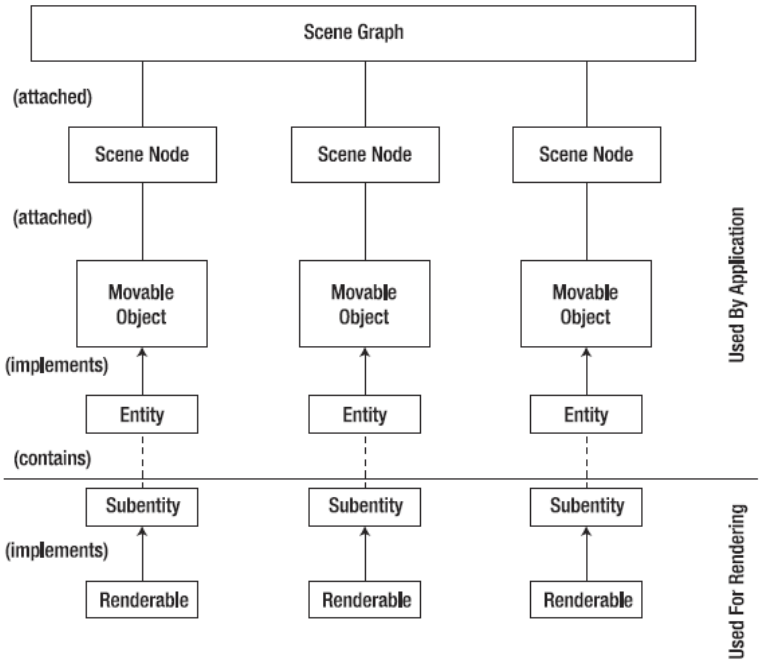

Figura 5.3. Relación entre el grafo de escena y su contenido

Cualquier implementación del grafo de escena se deriva de *SceneManager*, con el cual tendremos que interactuar de manera habitual y que además es la raíz para los nodos de la escena.

5.2.4 Rendering

En este apartado se encuentran las clases que se relacionan con los elementos del cauce de renderizado a más bajo nivel. Se utilizan para llevar toda la información de alto nivel de la escena a pantalla.

Dentro de este subsistema, la clase con la que trabajaremos más frecuentemente es *RenderWindow*, la cual representará el destino donde renderizar la escena y la pantalla principal de nuestra aplicación.

5.3 Diseño del contenido

Como se puede observar en la arquitectura del sistema [Figura 5.1], la única información que recibe el usuario proviene del apartado gráfico, por lo que representa un aspecto importante de cara al usuario final. Es necesario crear un entorno visual adecuado que ayude a la inmersión del usuario.

A su vez, es preciso cubrir los requisitos hallados en la fase de análisis, de los cuales podemos extraer los contenidos necesarios para el juego:

- Vehículo.
- Circuito.
- Interfaz de usuario.

El modelado 3D no es uno de los objetivos del proyecto y el dominio de una herramienta de modelado 3D requeriría un tiempo considerable, por lo que recurriremos a modelos diseñados previamente para el juego rFactor [3]. Estos modelos se encuentran en un formato nativo del juego, por lo que tendremos que valernos de diversas herramientas para incorporarlos a nuestro juego:

- Autodesk 3ds Max [4].

- rFactor Max Plugin [5].
- rFactor gMotor Mas Utility [5].
- OgreMax [6].

Una vez instaladas dichas herramientas deberemos localizar los archivos `.mas` que contienen los modelos 3D del vehículo y el circuito que deseemos utilizar.

5.3.1 Vehículo

El modelo 3D del vehículo se encuentra almacenado en distintos archivos. Cada uno se corresponde a una parte del coche y hará referencia a diferentes materiales y texturas. Por lo tanto nuestro objetivo es combinar todas las partes para formar un único objeto que represente el chasis del vehículo, además de los modelos de las ruedas.

En primer lugar, una vez elegido el vehículo que queremos utilizar, deberemos localizar su archivo `.mas` y extraer los componentes de dicho vehículo utilizando la herramienta gMotor Mas Utility proporcionada por los autores de rFactor.

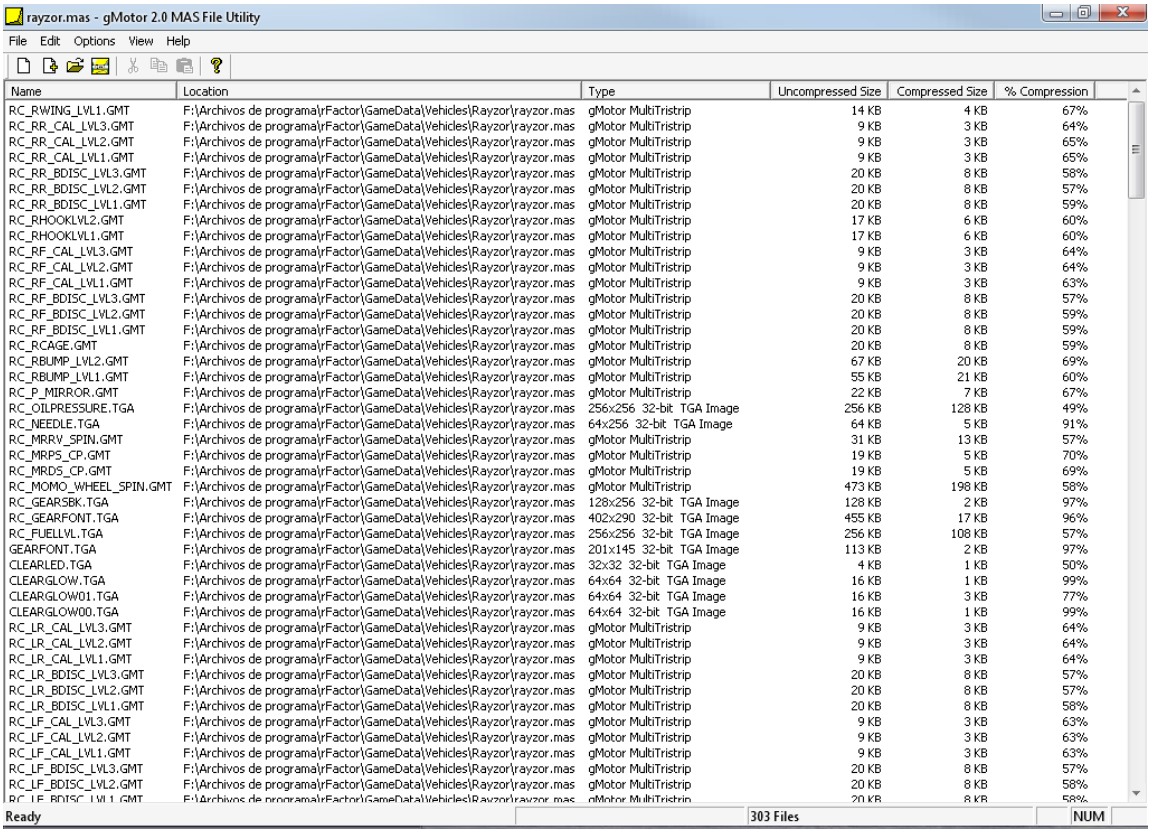

Figura 5.4. Componentes originales del vehículo

Una vez extraídos, tendremos que identificar los componentes principales, almacenados en ficheros .gmt, e importarlos a 3ds Max a través de MAXScript usando el correspondiente plugin (GMT 2 Importer).

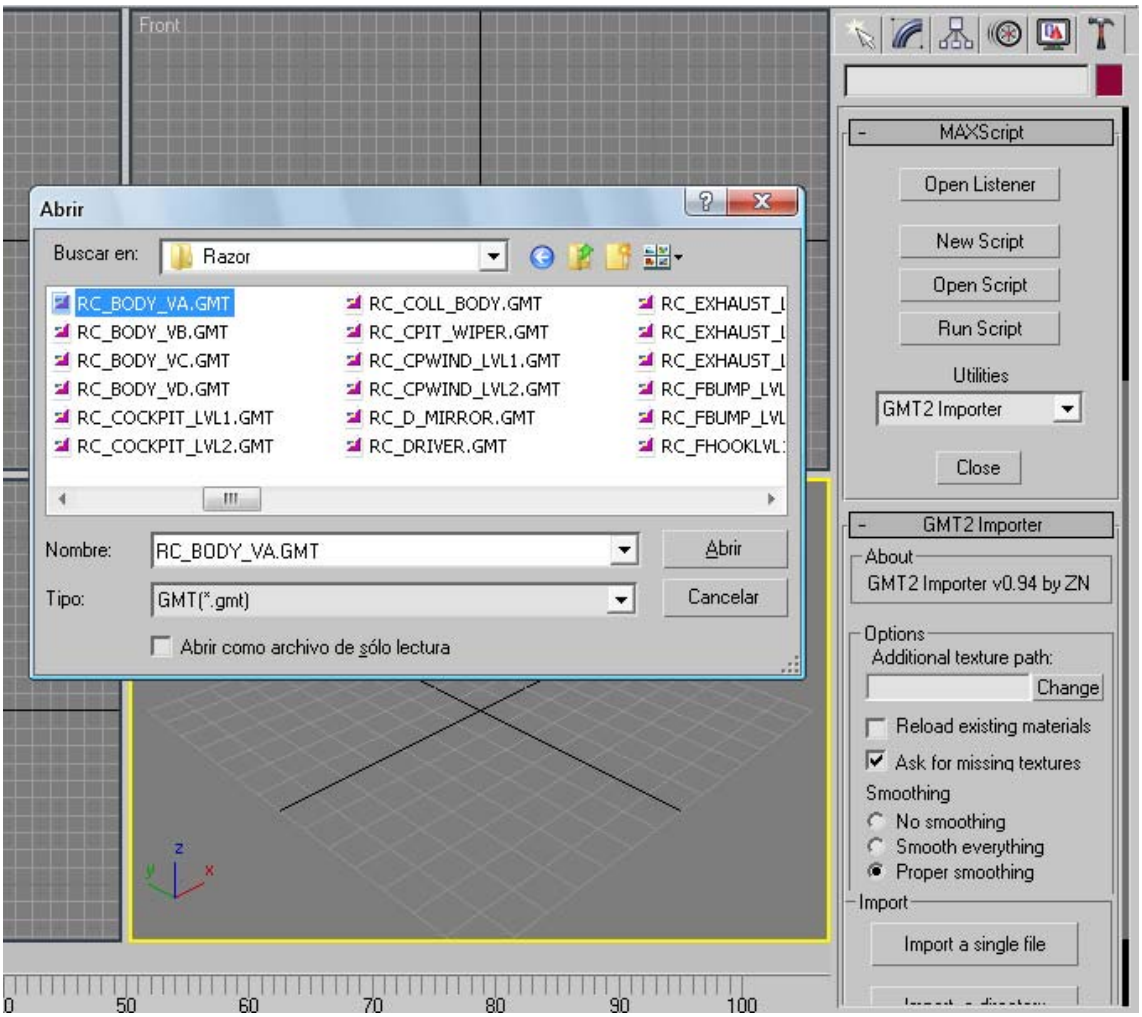

Figura 5.5. Importación de ficheros .gmt

El modelo importado tiene asociado una serie de materiales con un formato no estándar, por lo que tendremos que realizar una reasignación de materiales de forma que resulte apropiado para ser usado posteriormente en nuestro juego.

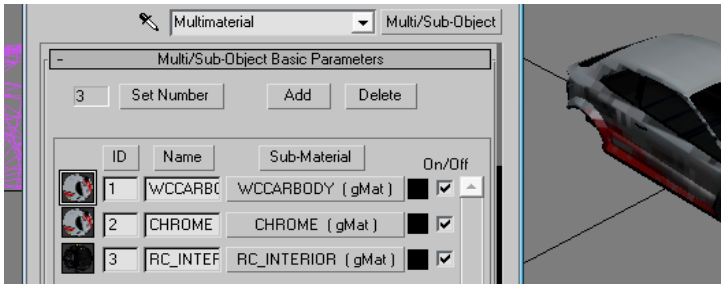

Figura 5.6. Materiales nativos

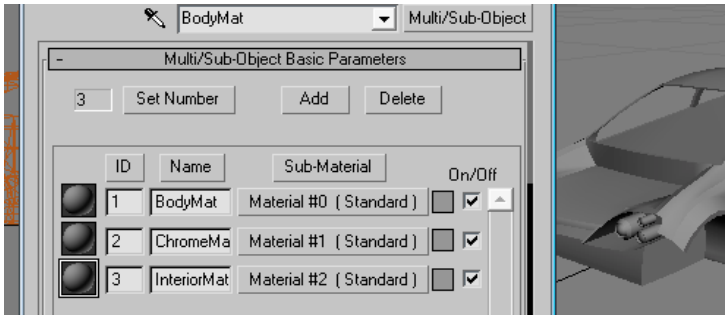

Figura 5.7. Materiales estándar

El mismo proceso se repite con cada componente del coche para finalmente fusionarlos todos y obtener el modelo definitivo de nuestro vehículo.

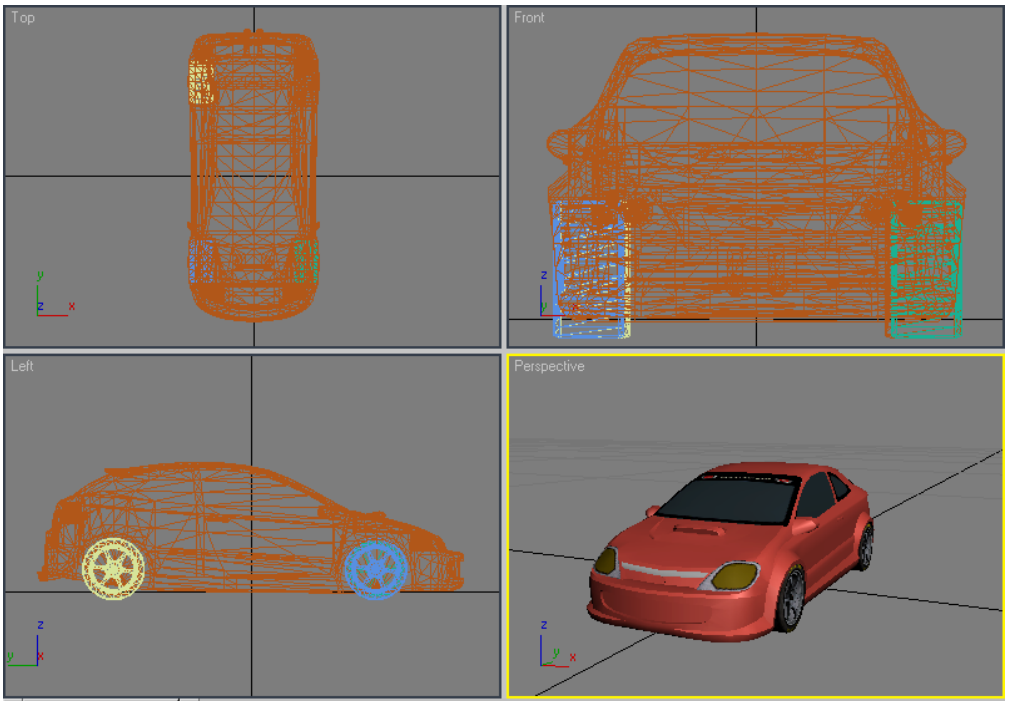

Figura 5.8. Modelo 3D del vehículo

Con los pasos anteriores podemos importar un objeto de rFactor a un objeto en Autodesk 3ds Max. Ahora necesitamos exportar dicho objeto a un formato entendible por nuestro motor gráfico, Ogre3D. Para ello haremos uso del plugin OgreMax, el cual nos facilitará los correspondientes archivos `.mesh` y `.material` reconocidos por Ogre3D.

5.3.2 Circuito

Siguiendo los requisitos establecidos para el tipo de circuito llevamos a cabo la elección de uno de los circuitos de rFactor y realizamos el mismo proceso que seguimos para importar y exportar el vehículo, con lo cual obtendríamos dos circuitos utilizables por nuestro juego, uno con un recorrido siguiendo el sentido de las agujas del reloj y otro con el recorrido inverso.

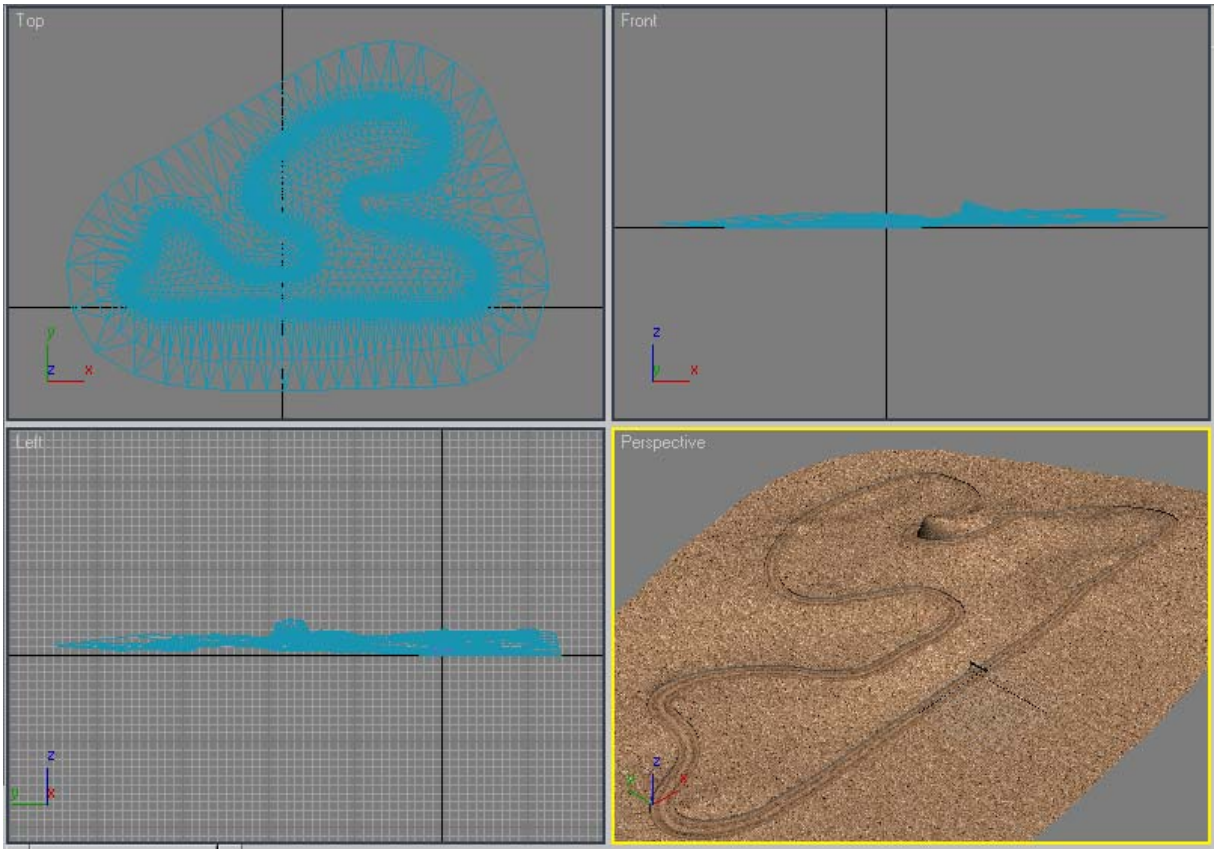

Figura 5.9. Modelo 3D del circuito

5.4 Diseño de clases

A lo largo del proyecto el diseño de clases ha ido cambiado tras las sucesivas revisiones, por lo que resulta apropiado mostrar sólo el diseño final de la jerarquía de clases.

Para una presentación más clara del sistema, las clases las agruparemos por la funcionalidad asociada y omitiremos sus atributos y operaciones, que se verán en el apartado de implementación.

Como norma y para mantener una nomenclatura propia, al nombre de las clases se les añadirá el prefijo *Umr*, el cual proviene de UMa Racing.

5.4.1 Base común

Utilizar una superclase común a todas las demás clases resulta útil a la hora de añadir funcionalidad que comparten todas. Esta funcionalidad puede estar dedicada a tareas tales como el mantenimiento de referencias de punteros o la gestión de la asignación de memoria.

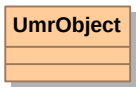

Figura 5.10. Superclase común UmrObject

5.4.2 Gestión de entrada

En este apartado se representan las clases encargadas de recoger entrada del usuario, bien a través del teclado o bien a través de un joystick instalado en el sistema.

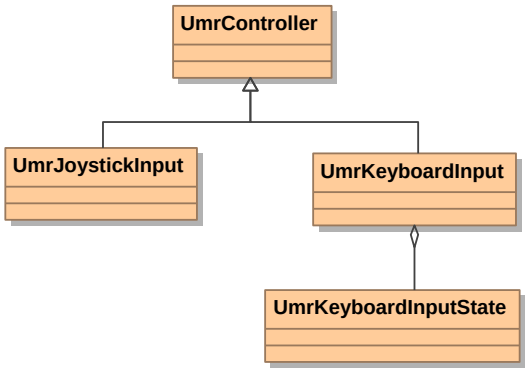

Figura 5.11. Clases para la gestión de la entrada

La clase *UmrController* nos proporcionará una interfaz común para cada controlador que pueda utilizar el usuario, mientras que *UmrJoystickInput* y *UmrKeyboardInput* nos proporcionará una implementación específica apropiada al dispositivo en concreto.

La funcionalidad aportada por estas clases es la capacidad de transformar las pulsaciones de teclas en acciones del vehículo, giro del volante o aceleración. Así mismo debe permitir la actualización periódica del estado de estas acciones.

5.4.3 Gestión de gráficos

Las clases que mayor relación guardan con el motor gráfico Ogre3D se encargan de la gestión de la ventana de renderizado (*UmrWindow*) y la gestión de la escena (*UmrScene*).

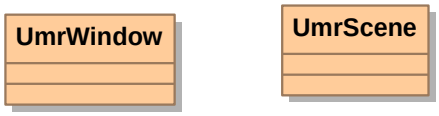

Figura 5.12. Clases para la gestión de gráficos

La clase *UmrWindow* requiere funcionalidad para crear la ventana donde podamos dibujar la escena, así como la posibilidad de configurarla y destruirla cuando sea necesario.

Por otra parte la clase *UmrScene* nos debe permitir crear un escenario a través de Ogre3D, además de poder actualizarlo y desecharlo según sea conveniente.

5.4.4 Gestión de físicas

En este caso tan sólo necesitamos una clase que se encapsule el motor físico Havok y que se encargue de gestionarlo.

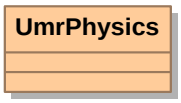

Figura 5.13. Clase para la gestión de la física

La clase *UmrPhysics* debe ser capaz de inicializar el sistema Havok, crear un mundo físico, actualizarlo y destruirlo.

5.4.5 Inteligencia artificial

El sistema de inteligencia artificial tiene que estar fuertemente ligado a las clases de control puesto que su objetivo final es proporcionar valores de salida que permitan controlar el vehículo. Por este motivo, el controlador del agente inteligente se diseña como una subclase de *UmrController*.

Como resultado de las iteraciones realizadas en el desarrollo de las redes neuronales tenemos dos controladores funcionales, uno de ellos apropiado para conducción prudente (*UmrAIControlSlow*) y otro para conducción rápida (*UmrAIControlFast*).

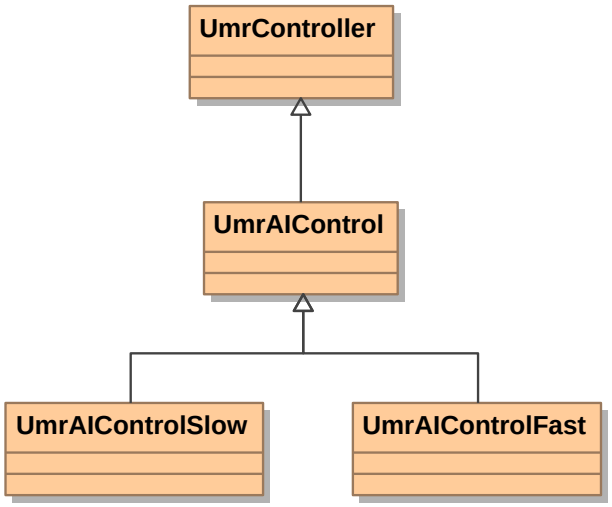

Figura 5.14. Clases para la inteligencia artificial

5.4.6 Juego

El primer aspecto a tener en cuenta al diseñar la jerarquía de clases del juego es que deseamos tener un sistema multihebra en el cual la actualización de la física se realice en paralelo a la actualización de los gráficos. Para cumplir este requisito debemos diseñar un conjunto de clases que nos permitan realizar la ejecución mediante hebras y la sincronización entre ellas.

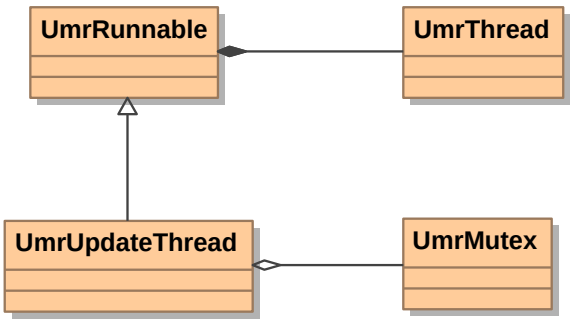

Figura 5.15. Clases para la ejecución multihebra

La clase *UmrThread* proporciona una encapsulación de las hebras del sistema operativo. Contiene funcionalidad para crear una hebra, suspenderla, reactivarla, destruirla y esperar a su destrucción.

La clase que contendrá el código que queremos ejecutar en paralelo será aquella que implemente la interfaz *UmrRunnable*. Mientras que *UmrThread* nos proporciona un mecanismo para ejecutar una hebra en paralelo, a través de *UmrRunnable* podremos utilizar dicho mecanismo indicando qué queremos ejecutar.

UmrUpdateThread será la clase específica donde la actualización de la física y del estado de los controladores tendrá lugar. Esta actualización debe estar sincronizada con la de la representación gráfica para tener un sistema coherente, por lo que necesitamos además la clase *UmrMutex* para realizar dicha sincronización.

La clase encargada de la gestión del juego y de ejecutar el bucle principal es *UmrGame*. Esta clase estará a cargo de iniciar el juego y los demás subsistemas, además de ejecutar la actualización de los gráficos en paralelo con la actualización de la física.

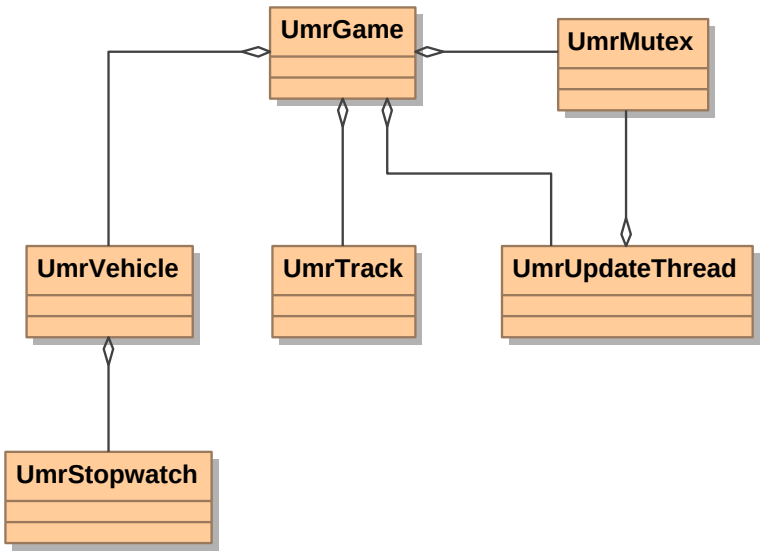

Figura 5.16. Clases principales del juego

UmrVehicle tiene dos funciones principales. Por una parte es un contenedor para los datos de configuración del vehículo. Por otra parte gestiona la carga de contenido del vehículo y su actualización para mantener la coherencia con el estado del chasis en el mundo físico. Además proporciona funcionalidad para registrar los datos provenientes de los sensores.

Puesto que es necesario llevar el control del tiempo por vuelta del vehículo se requiere añadir la clase *UmrStopwatch* diseñada para tal destino.

UmrTrack es el contenedor de la información gráfica y física del circuito. Como tal su propósito es gestionar la carga y configuración tanto del circuito como de los sectores en que éste está dividido.

Capítulo 6

Implementación de la aplicación

En esta fase se lleva a cabo la idea inicial que hemos ido definiendo en fases anteriores. La implementación puede entenderse como una traducción del diseño a un formato que sea comprensible para la maquina.

Este proyecto se ha desarrollado utilizando un proceso iterativo e incremental, de forma que al final de cada iteración tenemos un prototipo funcional sobre el cual podemos verificar el correcto funcionamiento. Cada versión sucesiva implementa un subconjunto de la funcionalidad requerida o un componente del juego concreto, por lo que al final obtenemos como resultado la aplicación completa.

Concretamente los componentes desarrollados e incorporados en cada paso son:

- Gestión de memoria y múltiples hebras.
- Gestión de gráficos.
- Gestión de física.
- Gestión de entrada.
- Inteligencia artificial.

En primer lugar veremos las distintas bibliotecas utilizadas en el desarrollo del juego, para más tarde explorar la implementación del proyecto siguiendo el orden en que se introdujeron los componentes.

6.1 Ogre3D

Una vez que entendemos como está diseñado Ogre3D y como sus diferentes componentes trabajan juntos, podemos empezar a escribir programas que utilicen su API.

6.1.1 Inicialización

El primer paso en cualquier aplicación que utilice Ogre3D es crear una instancia de *Root*. Su constructor puede tener varios argumentos en forma de cadenas de texto:

```
Root *root = new Root();
Root *root = new Root("plugins.cfg");
Root *root = new Root("plugins.cfg", "ogre.cfg");
Root *root = new Root("plugins.cfg", "ogre.cfg", "ogre.log");
```

Listado 1. Inicialización de Ogre3D

A partir de la segunda línea, los constructores contienen los nombres por defecto para los archivos de configuración, los mismos que se asumen en el primer constructor.

6.1.1.1 Plugins

Un plugin en Ogre3D es cualquier módulo (archivo `.dll` en Windows) que implementa una de sus interfaces, tales como *SceneManager* o *RenderSystem*. `plugins.cfg` contiene una lista de plugins que se cargarán al comenzar la aplicación.

```
# Define plugin folder
PluginFolder=.

# Define plugins
Plugin=RenderSystem_Direct3D9
Plugin=RenderSystem_GL
Plugin=Plugin_OctreeSceneManager
```

Listado 2. Contenido de plugins.cfg

La directiva `PluginFolder` indica a Ogre3D donde buscar los plugins listados en el archivo. El resto de líneas contienen los plugins que se quieren cargar.

6.1.1.2 Configuración

Ogre3D proporciona un medio simple para establecer la configuración gráfica a través de un diálogo de configuración.

```
Root *root = new Root();
if (!root->restoreConfig())
    root->showConfigDialog();
```

Listado 3. Configuración de Ogre3D

El código anterior intenta cargar la configuración gráfica desde el fichero `ogre.cfg` por

defecto, o en su caso, desde el fichero indicado en el constructor de *Root*. Si no existe el fichero, muestra el diálogo de configuración que lo creará.

Figura 6.1. Diálogo de configuración de Ogre3D

6.1.1.3 Registro

Ogre3D también proporciona medios para registrar excepciones y mensajes de diagnóstico. Esto es útil para obtener información detallada cuando ocurre un error en la aplicación. El registro que Ogre3D genera contiene mensajes de inicialización del sistema, estado, capacidades del hardware gráfico, etc. Todos estos mensajes se guardan en el fichero indicado en el constructor, `ogre.log` por defecto.

6.1.2 Ventana de renderización

Una vez seleccionado el sistema de renderizado (OpenGL o Direct3D) podemos crear una ventana donde se dibujará la escena.

```
RenderWindow *window = root->initialise(true, "My Render Window");
```

Listado 4. Creación de una ventana en Ogre3D

Además de la ventana, Ogre3D necesita una cámara y una región de visualización o viewport donde dibujar.

```
Camera *cam = sceneMgr->createCamera("MainCamera");
cam->setPosition(Vector3(-6,1.5f,0));
cam->setNearClipDistance(5);
cam->setFarClipDistance(1000);
cam->setAspectRatio(Real(1.333333));

Viewport *vp = window->addViewport(camera);
vp->setBackgroundColour(ColourValue(0, 0, 0));
```

Listado 5. Cámara y región de visualización en Ogre3D

6.1.3 Gestor de escena

Para organizar los elementos que se van a representar en la escena es necesario usar una estructura tal como un grafo de escena. Con este fin, en Ogre3D se utilizan los gestores de escena, los cuales siguen un esquema de plugins que implementan tipos concretos de gestores de escena.

```
SceneManager* sceneMgr =
    root->createSceneManager(ST_GENERIC, "MySceneManager");
```

Listado 6. Creación de un gestor de escena en Ogre3D

Cada tipo de gestor de escena puede ser apropiado para una situación u otra:

- `ST_GENERIC`. Aporta la funcionalidad mínima y no está optimizado para un contenido o estructura de la escena en particular. Es útil para escenas simples.
- `ST_INTERIOR`. Gestor de escena optimizado para escenas de interiores y altamente pobladas.
- `ST_EXTERIOR_CLOSE`. Gestor de escena optimizado para escenas de exteriores con visibilidad cercana o media.
- `ST_EXTERIOR_REAL_FAR`. Gestor de escena apropiado para paisajes grandes que requieren gran distancia de visibilidad.

6.1.4 Bucle de renderización

Típicamente una aplicación gráfica dibujará instantánea tras instantánea sin cesar hasta que se desee finalizar. Por lo tanto tendremos que incluir el método de Ogre3D para renderizar una instantánea en el bucle principal de la aplicación.

```
while (!salir)
{
    // procesar entrada

    // procesar física

    // actualizar escena

    // dibujar escena
    root->renderOneFrame();
    // comprobar si debemos salir
    // Nota: MensajeEnCola() es totalmente ficticio y
    // se usa aquí sólo con propósitos ilustrativos
    if (MensajeEnCola() == QUIT)
    {
        salir = true;
    }
}
```

Listado 7. Bucle principal con Ogre3D

6.2 Boost

Boost[7] es un conjunto de bibliotecas C++ ampliamente revisadas, algunas de las cuales están incluidas en el nuevo estándar de C++. Dentro del espectro de bibliotecas que ofrece, utilizaremos la biblioteca para punteros inteligentes, Boost Smart Pointers[8].

Los punteros inteligentes son objetos que almacenan punteros a objetos almacenados dinámicamente. Su comportamiento es similar a los punteros de C++, excepto que liberan automáticamente la memoria del objeto referenciado en el momento apropiado. Los punteros inteligentes son especialmente útiles para asegurar la correcta destrucción de objetos dinámicos y para gestionar objetos dinámicos con múltiples dueños.

6.3 NedMalloc

NedMalloc[9] es un asignador de memoria muy rápido, escalable y apropiado para múltiples hebras que produce una baja fragmentación de memoria. Para utilizarlo tenemos que reemplazar el asignador habitual de C++.

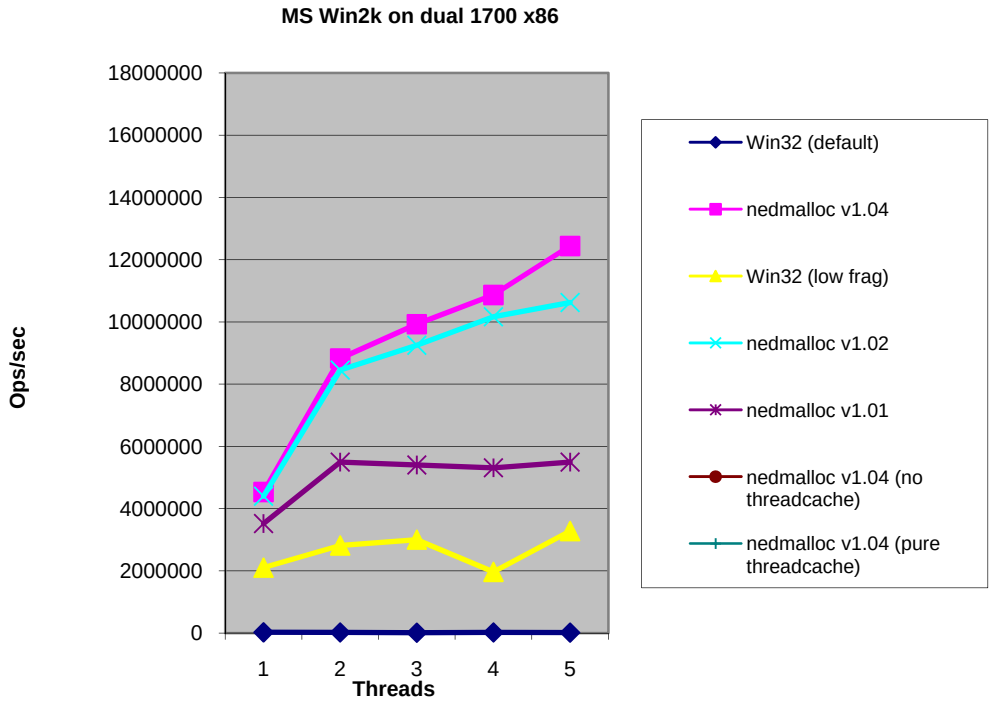

Figura 6.2. Asignadores de memoria

6.4 DirectInput

DirectInput[10] es una biblioteca de Microsoft que permite recoger datos del usuario a través de dispositivos de entrada tales como el ratón, el teclado, y joysticks u otros controladores para juegos.

Posee ciertas ventajas sobre el método habitual de recoger eventos de Windows, ya que es capaz de recoger datos incluso cuando la aplicación no está en primer plano y permite utilizar cualquier tipo de dispositivo de entrada.

Al igual que otras APIs de DirectX, DirectInput sigue la filosofía de componentes de Microsoft (COM – Component Object Model). El programador declara punteros a objetos y luego llama a una función para obtener una instancia de dicho objeto. Cada objeto proporciona un conjunto de métodos que pueden ser usados para distintas tareas. Los dos tipos de objetos COM de DirectInput en los que estamos interesados son:

- **DirectInput**. Su tipo es *IDirectInput8* y proporciona una interfaz para crear otros

objetos.

- **DirectInputDevice**. Su tipo es *IDirectInputDevice8* y representa a un dispositivo de entrada.

Habitualmente los pasos para utilizar un dispositivo en una aplicación son los siguientes:

- Crear un objeto DirectInput.
- Buscar los dispositivos si es necesario.
- Crear un objeto DirectInputDevice para cada dispositivo que se quiera utilizar.
- Configurar el dispositivo.
- Adquirir el dispositivo para su uso.
- Recoger datos y procesarlos.
- Liberar DirectInput al finalizar la aplicación.

6.5 Flood

Flood[2] es una exhaustiva biblioteca de clases que implementa un perceptrón multicapa en el lenguaje de programación C++. Incluye varios métodos de optimización, de entrenamiento y otras utilidades que lo hacen apropiado para la resolución de problemas tales como reconocimiento de patrones, modelado de datos, predicción de series temporales o diseño óptimo de formas.

Dentro del conjunto de clases de Flood las que utilizaremos para nuestros propósitos son:

- *InputTargetDataSet*. Representa un conjunto de datos en la forma entrada-objetivo para problemas de modelado de datos.
- *MultilayerPerceptron*. Representa un perceptrón multicapa.
- *NormalizedSquaredError*. Representa el método del error cuadrático mínimo para la optimización de la función objetivo.
- *QuasiNewtonMethod*. Representa el algoritmo de entrenamiento de quasi-Newton para el perceptrón multicapa.

6.6 Implementación de componentes

6.6.1 Bucle del juego

En su curso normal funcionamiento el juego sigue el siguiente diagrama:

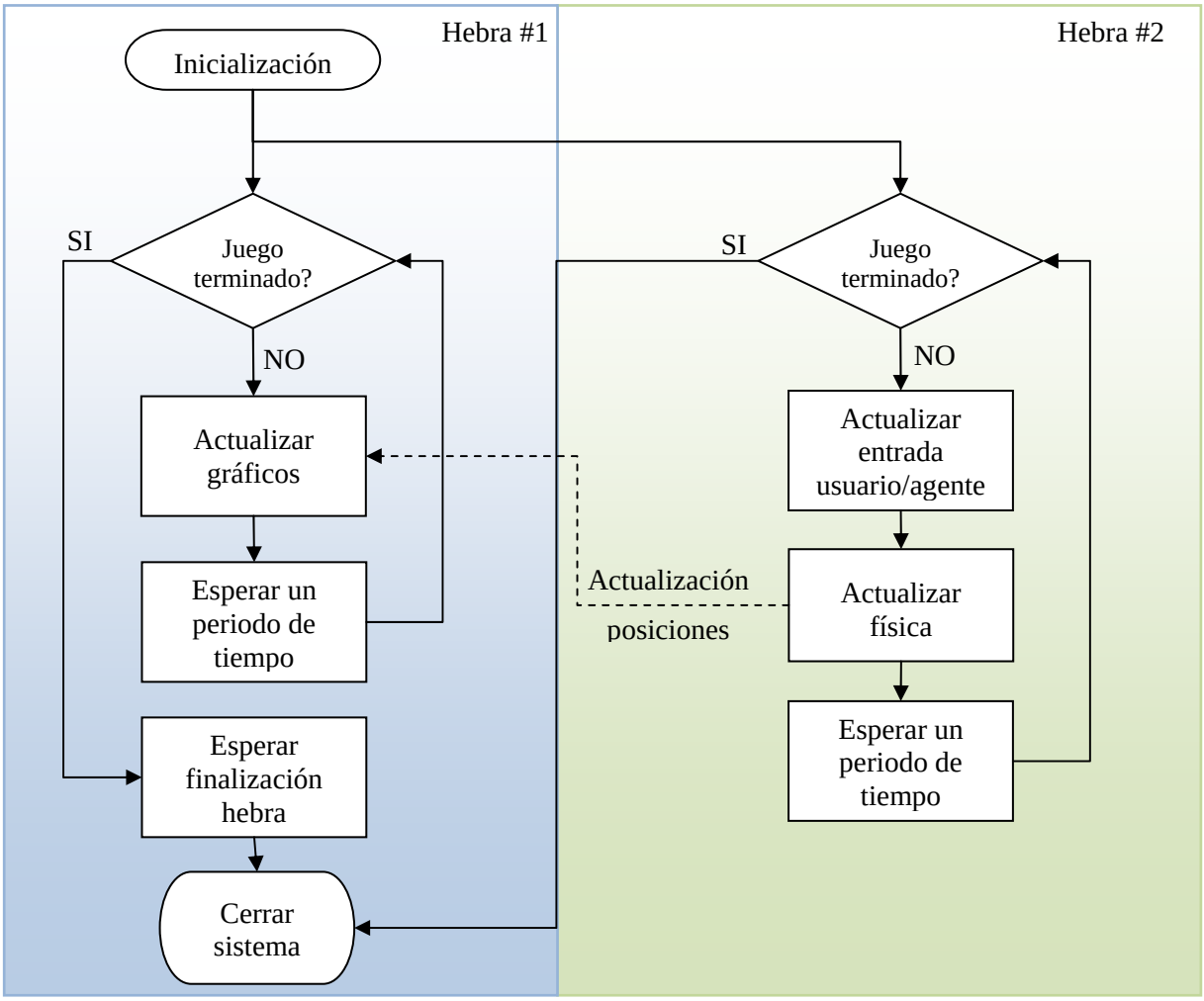

Figura 6.3. Funcionamiento normal del juego

6.6.2 Gestión de memoria

El primer sistema a desarrollar es la gestión de memoria, ya que todos los demás componentes harán uso de él. Para este propósito nos valdremos del mecanismo de punteros inteligentes de Boost y el asignador de memoria NedMalloc.

Para reemplazar el asignador de memoria habitual implementamos la clase *UmrObject* de la siguiente manera:

```
#define MAXTHREADSINPOOL 4

#ifdef new
#undef new
#endif
#ifdef delete
#undef delete
#endif

/**
    Base class for all game classes.
    Overrides memory allocation and dealocation
    operators to use the NedMalloc allocator.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrObject
{
public:
    explicit UmrObject(void);
    ~UmrObject(void);

    void* operator new(size_t sz);
    void* operator new(size_t sz, void* ptr);
    void* operator new[](size_t sz);
    void operator delete(void* ptr);
    void operator delete(void* ptr, void*);
    void operator delete[](void* ptr);
};
```

Listado 8. Declaración de UmrObject

Para cada clase de la aplicación se usa la directiva del compilador `__declspec(align(#))`, que nos permite alinear el almacenamiento en memoria de una clase mejorando el rendimiento. En nuestro caso alineamos los datos en bloques de 16 bytes.

De la declaración de *UmrObject* podemos destacar dos detalles:

- El constructor se declara como explícito para asegurarnos de que no se llama automáticamente en ningún momento.
- Se redefinen los operadores de C++ new y delete.

```
/**
    New operator. Allocates a new block of memory.
    @param
        sz Size of the memory block, in bytes.
*/
void* UmrObject::operator new(size_t sz)
{
    return nedalloc::nedmemalign(MEM_ALIGNMENT, sz);
}

/**
    Placement new operator. Avoid allocating new memory.
```

```
    @param
        ptr Pointer to a memory block to place the object.
*/
void* UmrObject::operator new(size_t sz, void *ptr)
{
    return ptr;
}

/**
    New operator for arrays.
*/
void* UmrObject::operator new[](size_t sz)
{
    return nedalloc::nedmemalign(MEM_ALIGNMENT, sz);
}

/**
    Deallocates a block of memory.
    @param
        ptr Pointer to a memory block previously allocated.
*/
void UmrObject::operator delete(void *ptr)
{
    if (!ptr) return;
    nedalloc::nedfree(ptr);
}

/**
    Deallocates a block of memory previously allocated.
*/
void UmrObject::operator delete(void *ptr, void *)
{
    if (!ptr) return;
    nedalloc::nedfree(ptr);
}

/**
    Deallocates a block of memory previously allocated.
*/
void UmrObject::operator delete[](void *ptr)
{
    if (!ptr) return;
    nedalloc::nedfree(ptr);
}
```

Listado 9. Definición de UmrObject

Como podemos observar, todas las peticiones de memoria que hacemos al asignador NedMalloc están alineadas en bloques de `MEM_ALIGNMENT` (16 bytes).

Una vez solucionado el problema de la asignación de memoria, tenemos que utilizar un sistema que nos evite las pérdidas de memoria que ocurren por ejemplo al perder la referencia a un objeto cuya memoria fue reservada con anterioridad. Para esto utilizamos uno de los tipos de puntero inteligente que proporciona Boost, concretamente el tipo *shared_ptr*. Este

tipo de objeto está diseñado para objetos con más de un dueño y que pueden estar referenciados por múltiples punteros.

Los punteros inteligentes de Boost están diseñados para poder ser utilizados de manera transparente al programador. Si queremos definir un puntero inteligente a una clase dada tan sólo tendremos que escribir lo siguiente:

```
typedef boost::shared_ptr<NombreClase> NombreClasePtr;
```

Listado 10. Definición de un puntero inteligente

6.6.3 Gestión de hebras

La API de Windows nos proporciona un mecanismo para la ejecución de hebras, pero nuestra intención es encapsular esta funcionalidad de forma que tengamos un conjunto de clases que nos proporcionen una abstracción de las capas inferiores.

En primer lugar desarrollamos la clase *UmrThread* que nos encapsulará las funciones de Windows para la gestión de hebras:

```
/**
    Thread class.
    Thin encapsulation of the Windows thread API.
    The thread will start in suspended state.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrThread : public UmrObject
{
public:
    UmrThread(unsigned (__stdcall * pFun) (void* arg), void* pArg);
    ~UmrThread(void);
    void resume();
    void suspend();
    void waitForDeath();
protected:
    /// Windows thread handle
    HANDLE m_handle;    // 4 bytes

    /// Thread id
    unsigned m_tid;        // 4 bytes
};
```

Listado 11. Declaración de UmrThread

```
/**
    Constructor.
    @param
        pFun Pointer to the function to be executed by the thread.
        pArg Pointer to the arguments of the function.
*/
```

```
UmrThread::UmrThread(
    unsigned (__stdcall *pFun) (void* arg), void* pArg)
{
    m_handle = (HANDLE)_beginthreadex(
                    NULL, 0, pFun, pArg, CREATE_SUSPENDED, &m_tid);
}

/**
    Destructor.
*/
UmrThread::~UmrThread(void)
{
    CloseHandle(m_handle);
}

/**
    Decrements a thread's suspend count. When the suspend count is
    decremented to zero, the execution of the thread is resumed.
*/
void UmrThread::resume()
{
    ResumeThread(m_handle);
}

/**
    Suspends the thread.
*/
void UmrThread::suspend()
{
    SuspendThread(m_handle);
}

/**
    Waits for this thread to die.
*/
void UmrThread::waitForDeath()
{
    WaitForSingleObject(m_handle, 1000);
}
```

Listado 12. Definición de UmrThread

Del código anterior hay que mencionar que cuando se crea una hebra, ésta se encontrará en suspensión para asegurarnos de que la ejecución no comienza hasta que se termine la construcción del objeto y se lo indiquemos.

Mientras que *UmrThread* nos proporciona una encapsulación de la API de Windows, *UmrRunnable* nos aporta una interfaz para ejecutar código en una hebra.

```
/**
      Runnable class.
      Abstract class that should be implemented by any class
      whose instances are intended to be executed by a thread.
*/
__declspec(align(MEM_ALIGNMENT))
```

```
class UmrRunnable : public UmrObject
{
public:
	UmrRunnable(void);
	virtual ~UmrRunnable(void) {}
	void start();
	void join();
	void suspend();
	void resume();

protected:
	virtual unsigned run () = 0;

	static unsigned __stdcall threadEntry (void *pArg);

	/// True when signaled to end
	bool m_isDying;

	/// Execution thread
	UmrThread m_thread;
};

typedef boost::shared_ptr<UmrRunnable> UmrRunnablePtr;
```

Listado 13. Declaración de UmrRunnable

```
/**
    Constructor.
*/
UmrRunnable::UmrRunnable(void) :
    m_isDying(false),
// 'this' : used in base member initializer list
#pragma warning(disable: 4355)
    m_thread(threadEntry, this)
{
}

/**
    Destructor.
*/
void UmrRunnable::join()
{
    m_isDying = true;
    m_thread.waitForDeath();
}

/**
    Executes the thread function.
    @param
        pArg Pointer to the UmrRunnable class to be executed.
*/
unsigned __stdcall UmrRunnable::threadEntry(void *pArg)
{
    UmrRunnable *pRunnable = (UmrRunnable *)pArg;
    return pRunnable->run();
}

/**
```

```
    Starts executing the thread.
*/
void UmrRunnable::start()
{
    m_thread.resume();
}

/**
    Suspends the execution of the thread.
*/
void UmrRunnable::suspend()
{
    m_thread.suspend();
}

/**
    Resumes the execution of the thread.
*/
void UmrRunnable::resume()
{
    m_thread.resume();
}
```

Listado 14. Definición de UmrRunnable

El constructor de *UmrRunnable* inicializa la hebra pasándole un puntero al método `run()` y un puntero a sí mismo, para lo que tenemos que deshabilitar el aviso que da el compilador al utilizar '`this`' en el constructor. Puesto que la hebra se inicia en estado de suspensión, el método `start()` tiene que activarla para comenzar la ejecución.

Puesto que el método `threadEntry()` es estático tenemos que pasarle la referencia a *UmrRunnable* como un puntero para tener acceso al método `run()`.

El método virtual `run()` será el que contendrá el código a ejecutar en la hebra, por lo tanto cualquier clase que queramos que se ejecute en una hebra tan sólo tendrá que implementar este método.

Una vez que tenemos el mecanismo para la ejecución de hebras, necesitamos una forma de sincronizarlas entre sí. Con este objetivo desarrollamos la clase *UmrMutex*, que nos permite ejecutar porciones de código en exclusión mutua. Al igual que *UmrThread*, encapsulamos las funciones de la API de Windows para poder tener acceso a su funcionalidad mediante una abstracción orientada a objetos.

```
#define SPIN_COUNT    4000

/**
    Mutex class.
```

```
    Provides synchronization between threads to avoid the
    simultaneous use of a common resource.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrMutex : public UmrObject
{
public:
    UmrMutex(void);
    explicit UmrMutex(DWORD spinCount);
    ~UmrMutex(void);

    void acquire();
    bool tryAcquire();
    void release();
protected:
    /// Windows synchronization object
    CRITICAL_SECTION m_criticalSection;
};

typedef boost::shared_ptr<UmrMutex> UmrMutexPtr;
```

Listado 15. Declaración de UmrMutex

```
/**
    Constructor.
*/
UmrMutex::UmrMutex(void)
{
    InitializeCriticalSectionAndSpinCount(
        &m_criticalSection, SPIN_COUNT);
}

/**
    Constructor.
    @param
        spinCount Spin count for the critical section.
*/
UmrMutex::UmrMutex(DWORD spinCount)
{
    InitializeCriticalSectionAndSpinCount(
        &m_criticalSection, spinCount);
}

/**
    Destructor.
*/
UmrMutex::~UmrMutex(void)
{
    DeleteCriticalSection(&m_criticalSection);
}

/**
    Waits for ownership of the mutex object. The method
    returns when the calling thread is granted ownership.
*/
void UmrMutex::acquire()
{
    EnterCriticalSection(&m_criticalSection);
```

```
}

/**
    Attempts to acquire the mutex without blocking.
    If the call is successful, the calling thread
    takes ownership of the mutex.
*/
bool UmrMutex::tryAcquire()
{
    if (TryEnterCriticalSection(&m_criticalSection)) {
        return true;
    } else {
        return false;
    }
}

/**
    Releases the ownership of the mutex.
*/
void UmrMutex::release()
{
    LeaveCriticalSection(&m_criticalSection);
}
```

Listado 16. Definición de UmrMutex

Para ejecutar la actualización de la entrada del usuario y la física tenemos que desarrollar una subclase de *UmrRunnable* que se encargue de ello.

```
/**
    Update thread class.
    Thread implementation used to update vehicles' input
    and perform physics simulation steps.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrUpdateThread : public UmrRunnable
{
public:
    UmrUpdateThread(UmrMutexPtr &mutex);
    ~UmrUpdateThread(void);

    const bool stopped() const;
    void setVehicles(UmrVehiclePtr (&vehicles)[NCARS]);
protected:
    unsigned run ();

    /// True when the player wants to quit the game
    LONG m_quit;

    /// Reference to physics system
    UmrPhysicsPtr m_pPhysics;

    /// Mutex used to gain exclusive access to vehicles
    UmrMutexPtr    m_pMutex;

    /// Vehicles of the game
    UmrVehiclePtr m_vehicles[NCARS];
```

```
};

typedef boost::shared_ptr<UmrUpdateThread> UmrUpdateThreadPtr;
```

Listado 17. Declaración de UmrUpdateThread

Para comprender la definición de UmrUpdateThread necesitamos conocer varios aspectos no vistos todavía, por lo que posponemos su introducción para apartados posteriores.

6.6.4 Gestión de gráficos

En este apartado veremos las clases dedicadas a las tareas gráficas así como cualquier funcionalidad gráfica que proporcionan las clases del proyecto.

6.6.4.1 Ventana gráfica

La clase *UmrWindow* está únicamente dedicada a la gestión de la ventana gráfica, para lo que hace uso de la funcionalidad de Ogre3D presentada en apartados anteriores, proporcionando una encapsulación que podemos utilizar de manera conveniente en nuestra aplicación.

```
using namespace Ogre;

/**
    Window class.
    Manages Ogre3D window creation, configuration and destruction.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrWindow : public UmrObject
{
public:
    UmrWindow(RootPtr &root);
    ~UmrWindow(void);

    bool            loadGraphicsConfiguration ();
    bool            initialize ();
    void            destroy ();
    const HWND        getWindowHandle () const;
    RenderWindow*    getRenderWindow ();
    const float         getLastFps() const;
    void            setViewport(Camera* camera);
protected:
    /// Root class for Ogre3D system
    RootPtr m_pRoot;

    /// Target rendering window
    RenderWindow* m_pWindow;

    /// Window title
    String m_sTitle;

    /// Window handle
```

```
    HWND m_hWnd;
};

typedef boost::shared_ptr<UmrWindow> UmrWindowPtr;
```

Listado 18. Declaración de UmrWindow

```
/**
    Constructor.
    @param
        root Reference to the Ogre3D root class.
*/
UmrWindow::UmrWindow(RootPtr &root) : m_pRoot(root)
{
    m_sTitle = Ogre::String(TITLE);
}

/**
    Destructor.
*/
UmrWindow::~UmrWindow(void)
{
}

/**
    Loads rendering options from a previously created
    configuration file or shows a dialog to choose the settings.
*/
bool UmrWindow::loadGraphicsConfiguration()
{
    if (!m_pRoot->restoreConfig()) {
        if (!m_pRoot->showConfigDialog()) {
            return false;
        }
    }
    return true;
}

/**
    Initializes the render window.
*/
bool UmrWindow::initialize()
{
    m_pWindow = m_pRoot->initialise(true, m_sTitle);

    // set window handle
    m_hWnd = FindWindow("OgreD3D9Wnd", TITLE);
    if (!IsWindow(m_hWnd))
        m_hWnd = FindWindow("OgreGLWindow", TITLE);
    if (!IsWindow(m_hWnd))
        return false;

    // set window icon
    char buf[1024];
    GetModuleFileName(0, (LPCH)&buf, 1024);
    HINSTANCE hInstance = GetModuleHandle(buf);
    HICON hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APP_ICON));
    if (hIcon) {
```

```
        SendMessage(m_hWnd, WM_SETICON, 1, (LPARAM)hIcon);
        SendMessage(m_hWnd, WM_SETICON, 0, (LPARAM)hIcon);
    }

    return true;
}

/**
    Destroys the render window.
*/
void UmrWindow::destroy()
{
    m_pWindow->removeAllViewports();
    m_pRoot->detachRenderTarget(m_pWindow);
}

/**
    Gets the window handle.
*/
const HWND UmrWindow::getWindowHandle() const
{
    return m_hWnd;
}

/**
    Gets a pointer to the render window.
*/
RenderWindow* UmrWindow::getRenderWindow()
{
    return m_pWindow;
}

/**
    Gets the number of frames per second (FPS).
*/
const float UmrWindow::getLastFps() const
{
    return m_pWindow->getLastFPS();
}

/**
    Sets the viewport for the render window.
    @param
        camera Pointer to the scene camera.
*/
void UmrWindow::setViewport(Ogre::Camera *camera)
{
    Viewport *vp = m_pWindow->addViewport(camera);
    vp->setBackgroundColour(ColourValue(0,0,0));
    // Alter the camera aspect ratio to match the viewport
    camera->setAspectRatio(
        Real(vp->getActualWidth()) / Real(vp->getActualHeight()));
}
```

Listado 19. Definición de UmrWindow

6.6.4.2 Escena

En la clase *UmrScene* agrupamos la funcionalidad dedicada a la gestión de la escena. Nos

permite crear el escenario gráfico donde tendrá lugar el juego, destruirlo al finalizar la aplicación y actualizarlo, por lo que además de utilizar la API de Ogre3D también necesitamos de la ayuda de Havok para actualizar la posición de la cámara en cada instante.

```
/**
      Scene class.
      Manages Ogre3D scene creation and updating.
      Object loading into the scene is delegated to
      the Vehicle and Track classes.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrScene : public UmrObject
{
public:
      UmrScene(RootPtr &root);
      ~UmrScene(void);

      bool createScene (UmrVehiclePtr &vehicle);
      void destroyScene ();
      Camera * getCamera();
      void updateScene ();
protected:
      void createFollowCamera ();
      void updateCamera ();

      /// Root class for Ogre3D system
      RootPtr m_pRoot;

      /// Camera from which the scene will be rendered
      Camera* m_pCamera;

      /// Ogre3D scene manager. Manages the organization
      /// and rendering of the scene.
      SceneManager* m_pSceneMgr;

      /// Vehicle followed by the scene camera
      UmrVehiclePtr m_pFollowVehicle;

      /// Havok camera helper. Camera attached to the vehicle
      hkp1dAngularFollowCam m_followCamera;

      /// Havok physics world
      hkpWorld* m_pWorld;
};

typedef boost::shared_ptr<UmrScene> UmrScenePtr;
```

Listado 20. Declaración de UmrScene

```
/**
    Constructor.
    @param
        root Reference to the Ogre3D root class.
*/
UmrScene::UmrScene(RootPtr &root) : m_pRoot(root)
```

```
{
    // create the scene manager
    //m_pSceneMgr=root->createSceneManager(ST_GENERIC, "SMInstance");
    m_pSceneMgr =
        root->createSceneManager(ST_EXTERIOR_CLOSE, "SMInstance");

    // create the camera
    m_pCamera = m_pSceneMgr->createCamera("PlayerCam");
    // Position the camera
    m_pCamera->setPosition(Vector3(-6,1.5f,0));
    // Look back along -Z
    m_pCamera->lookAt(Vector3(2,0,0));
    m_pCamera->setNearClipDistance(1);
    m_pCamera->setFarClipDistance(2000);
    m_pCamera->setFOVy(Radian(60.0f * M_PI / 180.0f));
}

/**
    Destructor.
*/
UmrScene::~UmrScene(void)
{
    m_pSceneMgr->destroyCamera("PlayerCam");
    m_pRoot->destroySceneManager(m_pSceneMgr);
}

/**
    Creates the elements of the graphic scene.
    @param
        vehicle Reference to the vehicle to be followed by the camera.
*/
bool UmrScene::createScene(UmrVehiclePtr &vehicle)
{
    m_pFollowVehicle = vehicle;
    createFollowCamera();

    m_pSceneMgr->setAmbientLight(ColourValue(0.8f,0.8f,0.8f));

    // Create a directional light
    Light *l = m_pSceneMgr->createLight("MainLight");
    Ogre::Vector3 dir(1,-1,-1);
    dir.normalise();
    l->setType(Ogre::Light::LT_DIRECTIONAL);
    l->setDirection(dir);
    l->setDiffuseColour(1, 1, 1);
    l->setSpecularColour(1, 1, 1);

    m_pSceneMgr->setSkyBox(true, "Sky1", 1000);

    return true;
}

/**
    Destroys the scene.
*/
void UmrScene::destroyScene()
{
    m_pSceneMgr->destroyLight("MainLight");
}
```

```
/**
    Gets a pointer to the camera of the scene.
*/
Camera * UmrScene::getCamera()
{
    return m_pCamera;
}

/**
    Creates a camera that follows a vehicle.
*/
void UmrScene::createFollowCamera()
{
    hkp1dAngularFollowCamCinfo cinfo;

    cinfo.m_yawSignCorrection = 1.0f;
    cinfo.m_upDirWS.set(0.0f, 1.0f, 0.0f);
    cinfo.m_rigidBodyForwardDir.set(1.0f, 0.0f, 0.0f);

    // Depending on the velocity of the car, the parameters actually
    // used are calculated by interpolating between two parameter sets
    cinfo.m_set[0].m_velocity = 10.0f;
    cinfo.m_set[1].m_velocity = 80.0f;
    cinfo.m_set[0].m_speedInfluenceOnCameraDirection = 1.0f;
    cinfo.m_set[1].m_speedInfluenceOnCameraDirection = 1.0f;
    cinfo.m_set[0].m_angularRelaxation = 3.0f;
    cinfo.m_set[1].m_angularRelaxation = 6.0f;

    cinfo.m_set[0].m_positionUS.set( -6.0f, 2.0f, 0.0f);
    cinfo.m_set[1].m_positionUS.set( -9.0f, 2.5f, 0.0f);

    cinfo.m_set[0].m_lookAtUS.set ( 2.0f, 0.0f, 0.0f );
    cinfo.m_set[1].m_lookAtUS.set ( 2.0f, 0.0f, 0.0f );

    cinfo.m_set[0].m_fov = 60.0f;
    cinfo.m_set[1].m_fov = 60.0f;

    m_followCamera.reinitialize( cinfo );
}

/**
    Updates the camera, based on where the vehicle is.
*/
void UmrScene::updateCamera()
{
    const hkpRigidBody *vehicleChassis =
        m_pFollowVehicle->getChassis();
    // Vehicle specific camera settings
    hkp1dAngularFollowCam::CameraInput in;
    {
        hkpWorld *world = vehicleChassis->getWorld();
        hkReal time = world->getCurrentTime();
        vehicleChassis->approxTransformAt( time, in.m_fromTrans );

        in.m_linearVelocity = vehicleChassis->getLinearVelocity();
        in.m_angularVelocity = vehicleChassis->getAngularVelocity();
        in.m_deltaTime = TIME_STEP_S;
    }
```

```
    hkp1dAngularFollowCam::CameraOutput out;
    {
        m_followCamera.calculateCamera( in, out );
    }

    // General camera settings
    Vector3 position(
        out.m_positionWS(0),
        out.m_positionWS(1),
        out.m_positionWS(2));
    Vector3 lookAt(
        out.m_lookAtWS(0),
        out.m_lookAtWS(1),
        out.m_lookAtWS(2));
    //float fov = out.m_fov * M_PI / 180.0f;
    m_pCamera->setPosition(position);
    m_pCamera->lookAt(lookAt);
    //m_pCamera->setFOVy(Radian(fov));
}

/**
    Updates the camera of the scene.
*/
void UmrScene::updateScene()
{
    updateCamera();
}
```

Listado 21. Definición de UmrScene

El constructor de la clase se encarga de crear los elementos mínimos para iniciar la aplicación: un gestor de escena de Ogre3D apropiado para exteriores y la cámara que se utilizará a lo largo del juego.

Los demás elementos, tales como la luz de la escena o el fondo de la escena se construyen en el método `createScene()`. Además hacemos uso de la clase *hkp1dAngularFollowCam* de Havok, que nos facilitará la tarea de actualización de la posición de la cámara con respecto al vehículo.

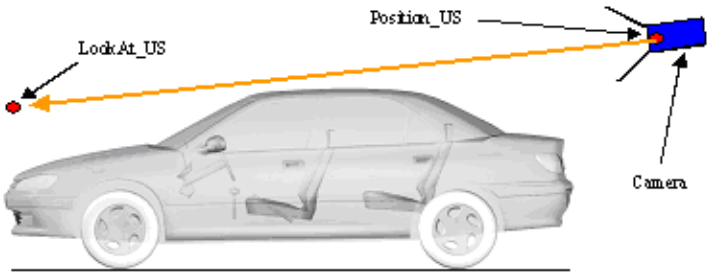

Figura 6.4. Posición de la cámara

Dicha clase nos proporciona una serie de parámetros divididos en dos conjuntos, de forma que la cámara tendrá un comportamiento diferente a baja y a alta velocidad. En puntos intermedios realizará una interpolación lineal tal y como se indica en la figura:

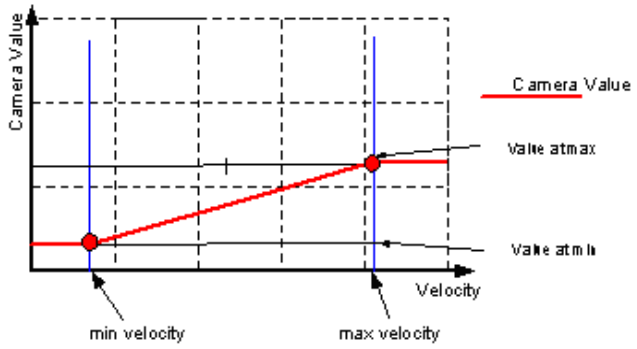

Figura 6.5. Comportamiento de la cámara a distintas velocidades

Desde *UmrScene* la única actualización que se lleva a cabo es la de la cámara, ya que la actualización gráfica del vehículo se delega en su propia clase *UmrVehicle*.

Para actualizar la cámara obtenemos la ayuda de Havok para calcular la nueva posición de la cámara. Con la nueva posición actualizamos la posición y la dirección de la cámara de Ogre3D.

6.6.4.3 Carga de contenidos gráficos

La carga de contenidos gráficos la realiza el objeto que necesita dicho contenido, es decir, *UmrVehicle* carga sus propios contenidos a la escena de Ogre3D, al igual que *UmrTrack*.

```
/**
      Loads graphic data and configuration from a file.
      @param
            cf Configuration file.
*/
void UmrVehicle::loadGraphics(const ConfigFile &cf)
{
      Vector3 vec;
      Entity *ent;
      SubEntity *sub;
      Root *root = Root::getSingletonPtr();
      SceneManager *sceneMgr = root->getSceneManager("SMInstance");
      SceneNode *rootNode = sceneMgr->getRootSceneNode();
      String id = StringConverter::toString(m_id);

      ent = sceneMgr->createEntity(m_sName + "_body" + id,
            cf.getSetting("Body", "Graphics"));
```

```
        if (m_id > 1) {
              sub = ent->getSubEntity(0);
              sub->setMaterialName("BodyMat" + id);
        }
        vec = StringConverter::parseVector3(
              cf.getSetting("BodyPos", "Graphics"));
        m_snBody = rootNode->createChildSceneNode(
              m_sName + "_body" + id, vec);
        m_snBody->attachObject(ent);

        ent = sceneMgr->createEntity(m_sName + "_tire_lf" + id,
              cf.getSetting("TireLeftFront", "Graphics"));
        vec = StringConverter::parseVector3(
              cf.getSetting("TireLeftFrontPos", "Graphics"));
        m_snTire_lf = m_snBody->createChildSceneNode(
              m_sName + "_tire_lf" + id, vec);
        m_snTire_lf->attachObject(ent);

        ent = sceneMgr->createEntity(m_sName + "_tire_lr" + id,
              cf.getSetting("TireLeftRear", "Graphics"));
        vec = StringConverter::parseVector3(
              cf.getSetting("TireLeftRearPos", "Graphics"));
        m_snTire_lr = m_snBody->createChildSceneNode(
              m_sName + "_tire_lr" + id, vec);
        m_snTire_lr->attachObject(ent);

        ent = sceneMgr->createEntity(m_sName + "_tire_rf" + id,
              cf.getSetting("TireRightFront", "Graphics"));
        vec = StringConverter::parseVector3(
              cf.getSetting("TireRightFrontPos", "Graphics"));
        m_snTire_rf = m_snBody->createChildSceneNode(
              m_sName + "_tire_rf" + id, vec);
        m_snTire_rf->attachObject(ent);

        ent = sceneMgr->createEntity(m_sName + "_tire_rr" + id,
              cf.getSetting("TireRightRear", "Graphics"));
        vec = StringConverter::parseVector3(
              cf.getSetting("TireRightRearPos", "Graphics"));
        m_snTire_rr = m_snBody->createChildSceneNode(
              m_sName + "_tire_rr" + id, vec);
        m_snTire_rr->attachObject(ent);
}
```

Listado 22. Carga de los contenidos gráficos del vehículo

Mediante la clase *ConfigFile* de Ogre3D podemos acceder de forma sencilla a configuraciones guardadas en un fichero mediante el método `getSetting()`. En nuestro caso debemos acceder a valores de configuración para el vehículo previamente almacenados en `razor.cfg`:

```
[Graphics]
Body=RZ_BODY.mesh
BodyPos=0 0 0
TireLeftFront=RZ_TR_LF.mesh
TireLeftFrontPos=0.70237 0.209205 1.13749
```

```
TireLeftRear=RZ_TR_LR.mesh
TireLeftRearPos=0.689354 0.209197 -1.25623
TireRightFront=RZ_TR_RF.mesh
TireRightFrontPos=-0.69412 0.209201 1.13749
TireRightRear=RZ_TR_RR.mesh
TireRightRearPos=-0.681105 0.209201 -1.25623
```

Listado 23. Configuración gráfica del vehículo

El vehículo está compuesto por cinco componentes, el chasis y las cuatro ruedas, por lo que tenemos que obtener el fichero `.mesh` y la posición de cada uno de ellos.

Una vez obtenidos los datos necesarios, tan sólo tenemos que crear una entidad de Ogre3D y añadirla al grafo de la escena.

La misma operación la tenemos que realizar para *UmrTrack*:

```
/**
    Loads graphic data and configuration from a file.
    @param
        cf Configuration file.
*/
void UmrTrack::loadGraphics(const ConfigFile &cf)
{
    Vector3 vec;
    Entity *ent;
    Root *root = Root::getSingletonPtr();
    SceneManager *sceneMgr = root->getSceneManager("SMInstance");
    SceneNode *rootNode = sceneMgr->getRootSceneNode();

    ent = sceneMgr->createEntity(m_sName + "_track",
        cf.getSetting("Track", "Graphics"));
    vec = StringConverter::parseVector3(
        cf.getSetting("TrackPos", "Graphics"));
    m_snTrack =
        rootNode->createChildSceneNode(m_sName + "_track", vec);
    m_snTrack->attachObject(ent);

    ent = sceneMgr->createEntity(m_sName + "_outfield",
        cf.getSetting("Outfield", "Graphics"));
    vec = StringConverter::parseVector3(
        cf.getSetting("OutfieldPos", "Graphics"));
    m_snOutfield =
        rootNode->createChildSceneNode(m_sName + "_outfield", vec);
    m_snOutfield->attachObject(ent);
}
```

Listado 24. Carga de los contenidos gráficos del circuito

En el caso del circuito tan sólo está dividido en dos componentes, la carretera y el terreno de los alrededores.

6.6.4.4 Actualización de contenidos gráficos

Aparte de la actualización de la cámara, también es necesario actualizar el vehículo y la interfaz de usuario:

```
/**
    Updates graphic rotation and position of the vehicle
    to match those of the physic world.
*/
void UmrVehicle::updateGraphics()
{
    // Get front wheels turning rotation
    float wheelRot =
        -MAX_STEERING *
        m_pController->getXPosition() *
        (HK_REAL_PI / 180.0f);
    Ogre::Quaternion wRot(
        Ogre::Radian(wheelRot), Ogre::Vector3(0,1,0));
    hkVector4 pos(m_pChassis->getPosition());
    Ogre::Vector3 oPos(pos(0), pos(1) - 0.3f, pos(2));

    // Update body
    hkVector4 up(0, 1, 0);
    hkQuaternion rot(m_pChassis->getRotation());
    hkQuaternion q(up, M_PI_2);
    rot.mul(q);
    Ogre::Quaternion oRot(rot(3), rot(0), rot(1), rot(2));
    m_snBody->setPosition(oPos);
    m_snBody->setOrientation(oRot);

    // Update left front wheel
    m_pVehicle->calcCurrentPositionAndRotation(m_pChassis,
        m_pVehicle->m_suspension, 0, pos, rot);
    oPos = m_snTire_lf->getPosition();
    oRot = Ogre::Quaternion(rot(3), rot(2), 0, 0);
    oRot = wRot * oRot;
    m_snTire_lf->setOrientation(oRot);
    m_snTire_lf->setPosition(oPos.x, pos(3) + 0.2f, oPos.z);

    // Update right front wheel
    m_pVehicle->calcCurrentPositionAndRotation(m_pChassis,
        m_pVehicle->m_suspension, 1, pos, rot);
    oRot = Ogre::Quaternion(rot(3), rot(2), 0, 0);
    oRot = wRot * oRot;
    oPos = m_snTire_rf->getPosition();
    m_snTire_rf->setOrientation(oRot);
    m_snTire_rf->setPosition(oPos.x, pos(3) + 0.2f, oPos.z);

    // Update left rear wheel
    m_pVehicle->calcCurrentPositionAndRotation(m_pChassis,
            m_pVehicle->m_suspension, 2, pos, rot);
    oRot = Ogre::Quaternion(rot(3), rot(2), 0, 0);
    oPos = m_snTire_lr->getPosition();
    m_snTire_lr->setOrientation(oRot);
    m_snTire_lr->setPosition(oPos.x, pos(3) + 0.2f, oPos.z);

    // Update right rear wheel
```

```
    m_pVehicle->calcCurrentPositionAndRotation(m_pChassis,
        m_pVehicle->m_suspension, 3, pos, rot);
    oRot = Ogre::Quaternion(rot(3), rot(2), 0, 0);
    oPos = m_snTire_rr->getPosition();
    m_snTire_rr->setOrientation(oRot);
    m_snTire_rr->setPosition(oPos.x, pos(3) + 0.2f, oPos.z);
}
```

Listado 25. Actualización gráfica del vehículo

Para actualizar el vehículo lo primero que hacemos es obtener el giro del volante y la rotación asociada a dicho giro. Con este valor podemos actualizar la rotación con respecto al eje vertical de las ruedas delanteras.

Mediante el motor físico obtenemos las posiciones y rotaciones tanto del chasis del vehículo como de las ruedas y con esta información actualizamos los elementos gráficos del vehículo.

Para dibujar la interfaz donde se muestra la información relativa al estado del vehículo utilizamos los Overlays de Ogre3D. Elementos 2D que se dibujan sobre el resto de elementos de la escena. Se pueden programar directamente sobre scripts de Ogre3D y tan sólo es necesario cargarlo al inicio de la aplicación.

Para cada información que queramos trasladar a la interfaz necesitamos crear un elemento 2D, en nuestro caso tenemos para cada vehículo la velocidad actual, el tiempo de la mejor vuelta, el tiempo de la última vuelta y el tiempo de la vuelta actual. Para el vehículo manejado por el usuario también indicamos si se encuentra en modo de entrenamiento, mientras que para los vehículos manejados por un agente inteligente indicamos sobre el contorno de un coche las acciones que llevan a cabo. Así mismo también se muestra información de depurado para ver el estado de ciertos sensores del vehículo.

```
/**
      Updates vehicle status information on the HUD.
*/
void UmrVehicle::updateStats()
{
      static String strSpeed = "Speed (kmh): ";
      static String strBestLap = "Best lap: ";
      static String strLastLap = "Last lap: ";
      static String strCurLap = "Current lap: ";
      static String strTrain = "Training";
      String id = StringConverter::toString(m_id);

      OverlayManager* overlayMgr = OverlayManager::getSingletonPtr();
      OverlayElement* speed =
            overlayMgr->getOverlayElement("Speed"+id);
      OverlayElement* bestLap =
```

```
            overlayMgr->getOverlayElement("BestLap"+id);
      OverlayElement* lastLap =
            overlayMgr->getOverlayElement("LastLap"+id);
      OverlayElement* curLap =
            overlayMgr->getOverlayElement("CurrentLap"+id);

      speed->setCaption(
            strSpeed + StringConverter::toString((int)m_speed));
      bestLap->setCaption(strBestLap + m_pStopwatch->getBestTime());
      lastLap->setCaption(strLastLap + m_pStopwatch->getLastTime());
      curLap->setCaption(strCurLap + m_pStopwatch->getCurrentTime());

      if (m_id > 1) {
            float xpos = m_pController->getXPosition();
            float ypos = m_pController->getYPosition();
            float width = hkMath::fabs(50.0f * xpos);
            float height = hkMath::fabs(50.0f * ypos);
            float left = 385.0f + (m_id-2) * 300.0f;
            float top = 55.0f;
            if (xpos < 0)
                  left -= width;
            if (ypos < 0)
                  top -= height;
            OverlayElement* turn =
                  overlayMgr->getOverlayElement("TurnPanel"+id);
            turn->setWidth(width);
            turn->setLeft(left);
            OverlayElement* accel =
                  overlayMgr->getOverlayElement("AccelPanel"+id);
            accel->setHeight(height);
            accel->setTop(top);
      } else {
            OverlayElement* train =
                  overlayMgr->getOverlayElement("Training");
            train->setCaption(
                  (m_pController->getTrainNNetwork()) ? strTrain : "");
//#ifdef _DEBUG
            OverlayElement* debugText1 =
                  overlayMgr->getOverlayElement("DebugText1");
            OverlayElement* debugText2 =
                  overlayMgr->getOverlayElement("DebugText2");
            String sXpos =
                  StringConverter::toString(m_pController->getXPosition());
            String sYpos =
                  StringConverter::toString(m_pController->getYPosition());

            String dir = StringConverter::toString(m_dirDiff, 3, 6);
            String tilt = StringConverter::toString(m_tilt, 3, 6);
            String ray0 = StringConverter::toString(m_rays[0], 3, 6);
            String ray1 = StringConverter::toString(m_rays[1], 3, 6);
            String ray2 = StringConverter::toString(m_rays[2], 3, 6);
            String ray3 = StringConverter::toString(m_rays[3], 3, 6);
            String ray4 = StringConverter::toString(m_rays[4], 3, 6);
            String ray5 = StringConverter::toString(m_rays[5], 3, 6);
            String ray6 = StringConverter::toString(m_rays[6], 3, 6);
            debugText1->setCaption(
                  sXpos + " " + sYpos + "   " + dir + " " + tilt);
            debugText2->setCaption(
                  ray0 + " " + ray1 + " " + ray2 + " " + ray3 + " " +
```

```
                ray4 + "    " + ray5 + " " + ray6);
//#endif
      }
}
```

Listado 26. Actualización de la interfaz

A través del gestor de Overlays de Ogre3D accedemos a los elementos de la interfaz que tenemos que actualizar. Estos elementos son en su mayoría cadenas de texto, por lo que la actualización consiste en concatenar el nombre de la información junto con su valor actual.

Para cada vehículo manejado por un agente representamos sus acciones sobre un eje de coordenadas. El giro del volante y la aceleración se muestran como una barra dibujada sobre los ejes X e Y respectivamente. La actualización consiste en actualizar las posiciones y el tamaño de dichas barras siguiendo el esquema de posicionamiento definido en el fichero `StatsPanel.overlay`.

6.6.4.5 Actualización de la escena

La actualización de la escena se lleva a cabo periódicamente y se encuentra dentro del bucle principal. El proceso que seguimos es: en el momento apropiado se actualiza la escena (actualización de la cámara) y seguidamente actualizamos cada uno de los vehículos.

```
    while(!quit) {
        if (window->isClosed() || m_updateThread->stopped()) break;

        /*if (!window->isActive()) {
            m_updateThread->suspend();
            WaitMessage();
            m_updateThread->resume();
        }*/

        elapsedTime = timer->getMilliseconds() - lastTime;
        if (elapsedTime > 8) {
            // Gain exclusive access to vehicles
            m_pMutex->acquire();
                // Update camera and vehicles transform
                m_pScene->updateScene();
                for (int i = 0; i < NCARS; i++)    {
                    m_vehicles[i]->updateGraphics();
                }
            m_pMutex->release();
            lastTime = timer->getMilliseconds();
        }

        fps->setCaption(
            StringConverter::toString(m_pWindow->getLastFps()));

        m_pRoot->renderOneFrame();
```

```
        // Handle windows messages
        Ogre::WindowEventUtilities::messagePump();
    }
```

Listado 27. Actualización de la escena

6.6.5 Gestión de física

Al igual que ocurre con la gestión de los gráficos, la funcionalidad relacionada con el procesamiento físico se encuentra repartida entre varias clases. Cada clase se encarga de llevar a cabo las funciones que le corresponden.

6.6.5.1 Gestión motor físico

Existe una clase cuyo único cometido es inicializar y configurar el entorno necesario para llevar a cabo la simulación física, *UmrPhysics*. Se puede identificar como un gestor central para la física, ya que inicia y finaliza el motor físico Havok y se encarga de avanzar la simulación física y realizar la actualización del depurador visual.

```
#define DEBUG_PHYSICS 1

class UmrPhysics;
typedef boost::shared_ptr<UmrPhysics> UmrPhysicsPtr;

/**
    Physics class.
    Manages the initialization and shut down of the
    Havok physics system and performs simulation steps.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrPhysics : public UmrObject
{
public:
    ~UmrPhysics(void);

    void initThread();
    void quitThread();
    void quit();
    void stepPhysics();
    static UmrPhysicsPtr& getInstance();
    hkpWorld* getWorld();
protected:
    UmrPhysics(void);

    static UmrPhysicsPtr s_instance;
    hkpWorld*            m_world;
    hkPoolMemory*        m_memoryMgr;
    //hkStlDebugMemory *m_memoryMgr;
    hkThreadMemory*      m_threadMemory;
    char*                m_stackBuffer;
    hkpPhysicsContext*   m_physicsCtx;
```

```
    hkVisualDebugger*    m_vdb;
};

static void errorReport(const char *str, void *errorOutputObject);
```

Listado 28. Declaración de UmrPhysics

```
UmrPhysicsPtr UmrPhysics::s_instance;

/**
    Constructor
*/
UmrPhysics::UmrPhysics(void)
{
    Ogre::LogManager* logMgr = Ogre::LogManager::getSingletonPtr();

    logMgr->logMessage("UmrPhysics: Initializing havok engine");

    m_memoryMgr = new hkPoolMemory();
    //m_memoryMgr = new hkStlDebugMemory();
    m_threadMemory = new hkThreadMemory(m_memoryMgr);

    hkBaseSystem::init(m_memoryMgr, m_threadMemory, errorReport);
    m_memoryMgr->removeReference();

    // Initialize the stack area to 100k
    {
        int stackSize = 0x100000;
        m_stackBuffer =
            hkAllocate<char>(stackSize, HK_MEMORY_CLASS_BASE);
        hkThreadMemory::getInstance().setStackArea(
            m_stackBuffer, stackSize);
    }

    // Create the world
    {
        hkpWorldCinfo info;
        info.m_simulationType =
            hkpWorldCinfo::SIMULATION_TYPE_DISCRETE;
        info.m_gravity.set(0, -9.8f, 0);
        info.setBroadPhaseWorldSize(1000.0f);
        info.setupSolverInfo(
            hkpWorldCinfo::SOLVER_TYPE_4ITERS_MEDIUM);
        m_world = new hkpWorld(info);
    }

    // Pre-allocate some larger memory blocks.
    // These are used by the physics system
    // when in multithreaded mode.
    // The amount and size of these depends on your physics usage.
    // Larger simulation islands will use larger memory blocks.
    {
        hkMemory::getInstance().preAllocateRuntimeBlock(
            512000, HK_MEMORY_CLASS_BASE);
        hkMemory::getInstance().preAllocateRuntimeBlock(
```

```
            256000, HK_MEMORY_CLASS_BASE);
        hkMemory::getInstance().preAllocateRuntimeBlock(
            128000, HK_MEMORY_CLASS_BASE);
        hkMemory::getInstance().preAllocateRuntimeBlock(
            64000, HK_MEMORY_CLASS_BASE);
        hkMemory::getInstance().preAllocateRuntimeBlock(
            32000, HK_MEMORY_CLASS_BASE);
        hkMemory::getInstance().preAllocateRuntimeBlock(
            16000, HK_MEMORY_CLASS_BASE);
        hkMemory::getInstance().preAllocateRuntimeBlock(
            16000, HK_MEMORY_CLASS_BASE);
    }

    // Register all agents
    hkpAgentRegisterUtil::registerAllAgents(
        m_world->getCollisionDispatcher());

    {
        hkpGroupFilter* filter = new hkpGroupFilter();

        // disable all collisions by default
        filter->disableCollisionsUsingBitfield(
            0xfffffffe, 0xfffffffe);

        // Enable collision filter for vehicles
        for (int i = 0; i < NCARS; i++) {
            filter->enableCollisionsUsingBitfield(
                1 << (CAR_FILTER + i),
                    (1 << TRACK_FILTER) |
                    (1 << OUTFIELD_FILTER) |
                    (1 << SECTOR_FILTER));
        }

        // Enable collision between vehicles
        filter->enableCollisionsUsingBitfield(1<<CAR_FILTER,
                (1 << (CAR_FILTER+1))|
                (1 << (CAR_FILTER+2)));
        filter->enableCollisionsUsingBitfield(1<<CAR_FILTER+1,
            (1 << (CAR_FILTER+2)));

        // Enable collision filter for ray casting
        filter->enableCollisionsUsingBitfield(1<<RAY_FILTER,
            (1 << OUTFIELD_FILTER) |
            (1 << CAR_FILTER));

        m_world->setCollisionFilter(filter);
        filter->removeReference();
    }

#ifdef DEBUG_PHYSICS
    // Instantiate a context so that the VDB will
    // know the physics worlds it can visualize
    m_physicsCtx = new hkpPhysicsContext();
    m_physicsCtx->addWorld(m_world);

    // Register all processes to expose to the VDB
    hkpPhysicsContext::registerAllPhysicsProcesses();

    // Create the Visual Debugger server
```

```
    hkArray<hkProcessContext*> contexts;
    contexts.pushBack(m_physicsCtx);
    m_vdb = new hkVisualDebugger(contexts);
    m_vdb->serve();
#endif
}

/**
    Destructor.
*/
UmrPhysics::~UmrPhysics(void)
{
}

/**
    Gets the instance of this class.
*/
UmrPhysicsPtr& UmrPhysics::getInstance()
{
    if (!s_instance)
        s_instance = UmrPhysicsPtr(new UmrPhysics());
    return s_instance;
}

/**
    Gets the physics world.
*/
hkpWorld* UmrPhysics::getWorld()
{
    return m_world;
}

/**
    Initializes thread for physics simulation.
*/
void UmrPhysics::initThread()
{
    // Need to call this for each active thread which uses Havok
    hkBaseSystem::initThread(m_threadMemory);
}

/**
    Clears resources used by a thread.
*/
void UmrPhysics::quitThread()
{
    hkBaseSystem::clearThreadResources();
}

/**
    Shut down physic system.
*/
void UmrPhysics::quit()
{
    Ogre::LogManager* logMgr = Ogre::LogManager::getSingletonPtr();

    logMgr->logMessage("UmrPhysics: Quitting havok engine");

    // Deallocate stack area
```

```
    m_threadMemory->setStackArea(0, 0);
    hkDeallocate(m_stackBuffer);

    //Deallocate runtime blocks
    hkMemory::getInstance().freeRuntimeBlocks();
    m_memoryMgr->removeReference();
    m_world->removeReference();
#ifdef DEBUG_PHYSICS
    m_physicsCtx->removeReference();
    m_vdb->removeReference();
#endif
    hkBaseSystem::quit();
}

/**
    Error report function for physics system.
*/
static void errorReport(const char *str, void *errorOutputObject)
{
    Ogre::LogManager *logMgr;

    logMgr = Ogre::LogManager::getSingletonPtr();
    if (logMgr != NULL) {
        // Redirect Havok messages to Ogre3D log file.
        logMgr->logMessage(Ogre::String(str));
    }
}

/**
    Performs a step in the physics system.
*/
void UmrPhysics::stepPhysics()
{
    m_world->stepDeltaTime(TIME_STEP_S);
#ifdef DEBUG_PHYSICS
    // Step the VDB display
    m_vdb->step(TIME_STEP_S);
#endif
}
```

Listado 29. Definición de UmrPhysics

En el constructor se lleva a cabo la configuración de Havok. A través de los comentarios en el código podemos saber el propósito de cada región de código, pero a grandes rasgos lo que se hace es:

- Crear el gestor de memoria de Havok e iniciar el sistema y su zona de memoria.
- Crear el mundo donde tendrá lugar la simulación física.
- Pre asignar bloques de memoria para el sistema.
- Registrar los filtros de colisión.
- Inicializar el depurador visual.

6.6.5.2 Carga de contenidos físicos

La carga de contenidos físicos desde fichero se realiza en las clases *UmrTrack* y *UmrVehicle*.

6.6.5.2.1 Carga del circuito

```
/**
	Loads physics data and configuration from a file.
	@param
		cf Configuration file.
*/
void UmrTrack::loadPhysics(const ConfigFile &cf)
{
	String path;
	hkBinaryPackfileReader* reader;
	hkIstream* infile;
	hkRootLevelContainer* container;
	hkpPhysicsData* physicsData;
	Ogre::LogManager* logMgr = Ogre::LogManager::getSingletonPtr();
	Vector3 vec;

	path = m_sPath + cf.getSetting("Track", "Physics");
	logMgr->logMessage("UmrTrack: Loading " + path);

	// Load the data
	infile = new hkIstream(path.c_str());
	reader = new hkBinaryPackfileReader();
	reader->loadEntireFile(infile->getStreamReader());
	m_pTrackLoadedData = reader->getPackfileData();
	infile->removeReference();
	m_pTrackLoadedData->addReference();
	reader->removeReference();

	// Get the top level object in the file,
	// which we know is a hkRootLevelContainer.
	container =
		m_pTrackLoadedData->getContents<hkRootLevelContainer>();
	// Get the physics data
	physicsData = static_cast<hkpPhysicsData*>(
		container->findObjectByType(hkpPhysicsDataClass.getName()));
	m_pTrackRigidBody = physicsData->findRigidBodyByName("road");

	path = m_sPath + cf.getSetting("Outfield", "Physics");
	logMgr->logMessage("UmrTrack: Loading " + path);

	// Load the data
	infile = new hkIstream(path.c_str());
	reader = new hkBinaryPackfileReader();
	reader->loadEntireFile(infile->getStreamReader());
	m_pOutfieldLoadedData = reader->getPackfileData();
	infile->removeReference();
	m_pOutfieldLoadedData->addReference();
	reader->removeReference();

	// Get the top level object in the file,
	// which we know is a hkRootLevelContainer.
	container =
```

```
        m_pOutfieldLoadedData->getContents<hkRootLevelContainer>();
    // Get the physics data
    physicsData = static_cast<hkpPhysicsData*>(
        container->findObjectByType(hkpPhysicsDataClass.getName()));
    m_pOutfieldRigidBody = physicsData->findRigidBodyByName("gravel");

    // Create last sector end
    {
        hkpRigidBodyCinfo boxInfo;
        vec = StringConverter::parseVector3(
                    cf.getSetting("LastSectorSize", "Physics"));
        hkVector4 boxSize(vec[0], vec[1], vec[2]);
        hkpShape* boxShape = new hkpBoxShape(boxSize, 0);
        boxInfo.m_motionType = hkpMotion::MOTION_FIXED;
        boxInfo.m_collisionFilterInfo =
            hkpGroupFilter::calcFilterInfo(SECTOR_FILTER);
        vec = StringConverter::parseVector3(
                    cf.getSetting("LastSectorPosition", "Physics"));
        boxInfo.m_position.set(vec[0], vec[1], vec[2]);

        UmrLastSectorEnd* myPhantomShape =
            new UmrLastSectorEnd(m_vehicles);
        hkpBvShape* bvShape =
            new hkpBvShape(boxShape, myPhantomShape);
        boxShape->removeReference();
        myPhantomShape->removeReference();

        boxInfo.m_shape = bvShape;

        m_pFinish = new hkpRigidBody(boxInfo);
        bvShape->removeReference();
    }

    // Create first sector end
    {
        hkpRigidBodyCinfo boxInfo;
        vec = StringConverter::parseVector3(
                    cf.getSetting("Sector1Size", "Physics"));
        hkVector4 boxSize(vec[0], vec[1], vec[2]);
        hkpShape* boxShape = new hkpBoxShape(boxSize, 0);
        boxInfo.m_motionType = hkpMotion::MOTION_FIXED;
        boxInfo.m_collisionFilterInfo =
            hkpGroupFilter::calcFilterInfo(SECTOR_FILTER);
        vec = StringConverter::parseVector3(
                    cf.getSetting("Sector1Position", "Physics"));
        boxInfo.m_position.set(vec[0], vec[1], vec[2]);

        UmrFirstSectorEnd* myPhantomShape =
            new UmrFirstSectorEnd(m_vehicles);
        hkpBvShape* bvShape =
            new hkpBvShape(boxShape, myPhantomShape);
        boxShape->removeReference();
        myPhantomShape->removeReference();

        boxInfo.m_shape = bvShape;

        m_pSector1 = new hkpRigidBody(boxInfo);
        bvShape->removeReference();
    }
```

```
	// Create second sector end
	{
		hkpRigidBodyCinfo boxInfo;
		vec = StringConverter::parseVector3(
				cf.getSetting("Sector2Size", "Physics"));
		hkVector4 boxSize(vec[0], vec[1], vec[2]);
		hkpShape* boxShape = new hkpBoxShape(boxSize, 0);
		boxInfo.m_motionType = hkpMotion::MOTION_FIXED;
		boxInfo.m_collisionFilterInfo =
			hkpGroupFilter::calcFilterInfo(SECTOR_FILTER);
		vec = StringConverter::parseVector3(
				cf.getSetting("Sector2Position", "Physics"));
		boxInfo.m_position.set(vec[0], vec[1], vec[2]);

		UmrSecondSectorEnd* myPhantomShape =
			new UmrSecondSectorEnd(m_vehicles);
		hkpBvShape* bvShape =
			new hkpBvShape(boxShape, myPhantomShape);
		boxShape->removeReference();
		myPhantomShape->removeReference();

		boxInfo.m_shape = bvShape;

		m_pSector2 = new hkpRigidBody(boxInfo);
		bvShape->removeReference();
	}

	// Set vehicles' initial position
	{
		vec = StringConverter::parseVector3(
				cf.getSetting("StartPosition", "Physics"));
		bool reverse = StringConverter::parseBool(
					cf.getSetting("Reverse", "Physics"));
		for (int i = 0; i < NCARS; i++)
			m_vehicles[i]->setStartPosition(vec, reverse);
	}
}
```

Listado 30. Carga del contenido físico del circuito

Para obtener los cuerpos rígidos que representan al circuito y que diseñamos anteriormente mediante Autodesk 3ds Max hacemos uso de las facilidades de Havok para leer archivos .hkx. Con cada componente del circuito realizamos la misma operación: leer los datos empaquetados en el fichero, obtener el sistema físico que Havok Content Tools serializó en el fichero y acceder al cuerpo rígido almacenado en el dicho sistema físico.

Mención especial merece la carga de los sectores del circuito que nos van a permitir controlar el tiempo por vuelta de los vehículos. Cada sector está implementado como un cuerpo rígido fijo, pero la forma que lo representa es especial, es una forma fantasma la cual puede ejecutar un código dado cada vez que se detecta el solapamiento con otro cuerpo.

```
/**
    Custom shape used to detect when a
    vehicle crosses the sector end.
*/
class UmrSectorEnd : public hkpPhantomCallbackShape
{
public:
    UmrSectorEnd(UmrVehiclePtr (&vehicles)[NCARS])
    {
        for (int i = 0; i < NCARS; i++) {
            m_vehicles[i] = vehicles[i];
        }
    }

    /**
        This callback is called when the phantom shape
        starts intersecting with another shape.
    */
    virtual void phantomEnterEvent(const hkpCollidable* collidableA,
                                   const hkpCollidable* collidableB,
                                   const hkpCollisionInput& env) = 0;
    /**
        This callback is called when the phantom shape
        stops intersecting with another shape.
    */
    virtual void phantomLeaveEvent(const hkpCollidable* collidableA,
                                   const hkpCollidable* collidableB) = 0;
protected:
    UmrVehiclePtr    m_vehicles[NCARS];
};

/**
    Detects when a vehicle finishes the first sector.
*/
class UmrFirstSectorEnd : public UmrSectorEnd
{
public:
    UmrFirstSectorEnd(UmrVehiclePtr (&vehicles)[NCARS]) :
      UmrSectorEnd(vehicles) {}

    virtual void phantomEnterEvent(
        const hkpCollidable* collidableA,
        const hkpCollidable* collidableB,
        const hkpCollisionInput& env)
    {
        hkpRigidBody* owner = hkGetRigidBody(collidableB);
        // Get vehicle's id
        hkUlong id = owner->getUserData();
        m_vehicles[id-1]->firstSectorFinished();
    }

    virtual void phantomLeaveEvent(
        const hkpCollidable* collidableA,
        const hkpCollidable* collidableB) {}
};

/**
    Detects when a vehicle finishes the second sector.
*/
```

```
class UmrSecondSectorEnd : public UmrSectorEnd
{
public:
    UmrSecondSectorEnd(UmrVehiclePtr (&vehicles)[NCARS]) :
      UmrSectorEnd(vehicles) {}

    virtual void phantomEnterEvent(
        const hkpCollidable* collidableA,
        const hkpCollidable* collidableB,
        const hkpCollisionInput& env)
    {
        hkpRigidBody* owner = hkGetRigidBody(collidableB);
        hkUlong id = owner->getUserData();
        m_vehicles[id-1]->secondSectorFinished();
    }

    virtual void phantomLeaveEvent(
        const hkpCollidable* collidableA,
        const hkpCollidable* collidableB) {}
};

/**
    Detects when a vehicle finishes the last sector.
*/
class UmrLastSectorEnd : public UmrSectorEnd
{
public:
    UmrLastSectorEnd(UmrVehiclePtr (&vehicles)[NCARS]) :
      UmrSectorEnd(vehicles) {}

    virtual void phantomEnterEvent(
        const hkpCollidable* collidableA,
        const hkpCollidable* collidableB,
        const hkpCollisionInput& env)
    {
        hkpRigidBody* owner = hkGetRigidBody(collidableB);
        hkUlong id = owner->getUserData();
        m_vehicles[id-1]->lastSectorFinished();
    }

    virtual void phantomLeaveEvent(
        const hkpCollidable* collidableA,
        const hkpCollidable* collidableB) {}
};
```

Listado 31. Sectores del circuito

El funcionamiento de cada sector es prácticamente el mismo. Cuando Havok detecta que un vehículo se solapa por primera vez con un sector llama al método `phantomEnterEvent()`. En este método obtenemos el identificador del vehículo con el que se produce el solapamiento y, a través del array de vehículos almacenado en el objeto del sector, llamamos al método apropiado para indicar que se ha cruzado dicho sector.

6.6.5.2.2 Carga del vehículo

```
/**
    Sets up physics data for the vehicle.
*/
void UmrVehicle::loadPhysics(const ConfigFile &cf)
{
    /*
    String path;
    hkBinaryPackfileReader reader;
    hkpPhysicsData* physicsData;
    Ogre::LogManager* logMgr = Ogre::LogManager::getSingletonPtr();

    path = m_sPath + cf.getSetting("Packfile", "Physics");
    logMgr->logMessage("UmrVehicle: Loading " + path);

    // Load the data
    hkIstream infile(path.c_str());
    reader.loadEntireFile(infile.getStreamReader());
    m_pLoadedData = reader.getPackfileData();
    m_pLoadedData->addReference();

    // Get the top level object in the file,
    // which we know is a hkRootLevelContainer
    hkRootLevelContainer* container =
        m_pLoadedData->getContents<hkRootLevelContainer>();
    // Get the physics data
    physicsData = static_cast<hkpPhysicsData*>(
        container->findObjectByType(hkpPhysicsDataClass.getName()));
    m_pChassis = physicsData->findRigidBodyByName("RZ_COLL_BODY");
    */
    // Create rigid body manually
    hkReal xSize = 1.75f;
    hkReal ySize = 0.3f;
    hkReal zSize = 0.85f;

    hkReal xBumper = 1.9f;
    hkReal yBumper = 0.2f;
    hkReal zBumper = 0.80f;

    hkReal xRoofFront = 0.4f;
    hkReal xRoofBack = -1.2f;
    hkReal yRoof = ySize + 0.45f;
    hkReal zRoof = 0.6f;

    hkReal xDoorFront = xRoofFront;
    hkReal xDoorBack = xRoofBack;
    hkReal yDoor = ySize;
    hkReal zDoor = zSize + 0.1f;

    int numVertices = 22;

    // 16 = 4 (size of "each float group", 3 for x,y,z, 1 for padding)
    //      * 4 (size of float).
    int stride = sizeof(float) * 4;

    float vertices[] = {
        xSize, ySize, zSize, 0.0f,        // v0
```

```
    xSize, ySize, -zSize, 0.0f,         // v1
    xSize, -ySize, zSize, 0.0f,         // v2
    xSize, -ySize, -zSize, 0.0f,     // v3
    -xSize, -ySize, zSize, 0.0f,     // v4
    -xSize, -ySize, -zSize, 0.0f,     // v5

    xBumper, yBumper, zBumper, 0.0f,     // v6
    xBumper, yBumper, -zBumper, 0.0f,     // v7
    -xBumper, yBumper, zBumper, 0.0f,     // v8
    -xBumper, yBumper, -zBumper, 0.0f,     // v9

    xRoofFront, yRoof, zRoof, 0.0f,         // v10
    xRoofFront, yRoof, -zRoof, 0.0f,     // v11
    xRoofBack, yRoof, zRoof, 0.0f,         // v12
    xRoofBack, yRoof, -zRoof, 0.0f,         // v13

    xDoorFront, yDoor, zDoor, 0.0f,         // v14
    xDoorFront, yDoor, -zDoor, 0.0f,     // v15
    xDoorFront, -yDoor, zDoor, 0.0f,     // v16
    xDoorFront, -yDoor, -zDoor, 0.0f,     // v17

    xDoorBack, yDoor, zDoor, 0.0f,         // v18
    xDoorBack, yDoor, -zDoor, 0.0f,         // v19
    xDoorBack, -yDoor, zDoor, 0.0f,         // v20
    xDoorBack, -yDoor, -zDoor, 0.0f,     // v21
};

//
// SHAPE CONSTRUCTION.
//
hkpConvexVerticesShape* chassisShape;
hkArray<hkVector4> planeEquations;
hkGeometry geom;
{
    hkStridedVertices stridedVerts;
    {
        stridedVerts.m_numVertices = numVertices;
        stridedVerts.m_striding = stride;
        stridedVerts.m_vertices = vertices;
    }

    hkGeometryUtility::createConvexGeometry(
        stridedVerts, geom, planeEquations);

    {
        stridedVerts.m_numVertices = geom.m_vertices.getSize();
        stridedVerts.m_striding = sizeof(hkVector4);
        stridedVerts.m_vertices = &(geom.m_vertices[0](0));
    }

    chassisShape =
        new hkpConvexVerticesShape(stridedVerts, planeEquations);
}

chassisShape->setRadius(0.1f);

{
    hkpRigidBodyCinfo chassisInfo;
```

```
        chassisInfo.m_mass = 1300.0f;
        chassisInfo.m_shape = chassisShape;
        chassisInfo.m_restitution = 0.4f;
        chassisInfo.m_friction = 0.5f;

        // The chassis MUST have m_motionType
        // hkpMotion::MOTION_BOX_INERTIA to correctly simulate
        // vehicle roll, pitch and yaw.
        chassisInfo.m_motionType = hkpMotion::MOTION_BOX_INERTIA;
        chassisInfo.m_position.set(0, 0, 0);
        chassisInfo.m_collisionFilterInfo =
            hkpGroupFilter::calcFilterInfo(CAR_FILTER+m_id-1);
        hkpInertiaTensorComputer::setShapeVolumeMassProperties(
            chassisInfo.m_shape,
            chassisInfo.m_mass,
            chassisInfo);

        m_pChassis = new hkpRigidBody(chassisInfo);
        m_pChassis->setUserData(m_id);

        // No longer need reference to shape
        // as the hkpRigidBody holds one.
        chassisShape->removeReference();
    }
    m_lastValidTrans = m_pChassis->getTransform();
    m_lastValidDir = hkVector4(1, 0, 0);
}
```

Listado 32. Carga del contenido físico del vehículo

Por problemas de compatibilidad regional de Windows con Havok Content Tools y Autodesk 3ds Max, la carga del contenido físico del vehículo se tiene que hacer de forma manual. Para ello configuramos las dimensiones de la forma que representa al chasis del vehículo y construimos su cuerpo rígido asociado. Así mismo almacenamos en dicho cuerpo rígido información de identificación del vehículo que nos resultará útil en distintas situaciones.

6.6.5.3 Construcción del vehículo

Para poder introducir el vehículo en la simulación física, primero debemos construirlo y configurar todos sus componentes y parámetros.

```
/**
    Creates vehicles components and adds it to the physics world.
    @param
        world Pointer to the physics world.
*/
void UmrVehicle::buildVehicle(hkpWorld* world)
{
    world->addEntity(m_pChassis);

    m_pVehicle = new hkpVehicleInstance(m_pChassis);

    m_pVehicle->m_data    =
```

```
        new hkpVehicleData;
    m_pVehicle->m_driverInput =
        new hkpVehicleDefaultAnalogDriverInput;
    m_pVehicle->m_steering =
        new hkpVehicleDefaultSteering;
    m_pVehicle->m_engine =
        new hkpVehicleDefaultEngine;
    m_pVehicle->m_transmission =
        new hkpVehicleDefaultTransmission;
    m_pVehicle->m_brake =
        new hkpVehicleDefaultBrake;
    m_pVehicle->m_suspension =
        new hkpVehicleDefaultSuspension;
    m_pVehicle->m_aerodynamics =
        new hkpVehicleDefaultAerodynamics;
    m_pVehicle->m_velocityDamper =
        new hkpVehicleDefaultVelocityDamper;
    m_pVehicle->m_wheelCollide =
        new hkpVehicleRaycastWheelCollide;
    m_pVehicle->m_deviceStatus =
        new hkpVehicleDriverInputAnalogStatus;

    setupVehicleData(world, *(m_pVehicle->m_data));

    // Initialise the tyremarks controller with 128 tyremark points
    m_pVehicle->m_tyreMarks    =
        new hkpTyremarksInfo(*(m_pVehicle->m_data), 128);

    // Setup components configuration
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultAnalogDriverInput* >(
            m_pVehicle->m_driverInput));
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultSteering*>(
            m_pVehicle->m_steering));
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultEngine*>(
            m_pVehicle->m_engine));
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultTransmission*>(
            m_pVehicle->m_transmission));
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultBrake*>(
            m_pVehicle->m_brake));
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultSuspension*>(
            m_pVehicle->m_suspension));
    setupComponent(
        *(m_pVehicle->m_data),
        *static_cast<hkpVehicleDefaultAerodynamics*>(
            m_pVehicle->m_aerodynamics));
    setupComponent(
        *(m_pVehicle->m_data),
```

```
            *static_cast<hkpVehicleDefaultVelocityDamper*>(
                m_pVehicle->m_velocityDamper));

        // The wheel collide component performs collision detection.
        // To do this, it needs to create an aabbPhantom from the
        // vehicle information that has been set here already.
        setupWheelCollide(
            world,
            *m_pVehicle,
            *static_cast<hkpVehicleRaycastWheelCollide*>(
                m_pVehicle->m_wheelCollide));

        /*setupTyremarks(
            *(m_pVehicle->m_data),
            *static_cast<hkpTyremarksInfo*>(m_pVehicle->m_tyreMarks));*/

        // Don't forget to call init!
        // (This function is necessary to set up derived data)
        m_pVehicle->init();

        // The phantom for collision detection needs
        // to be explicitly added to the world
        world->addPhantom(
            (hkpPhantom*)(static_cast<hkpVehicleRaycastWheelCollide*>(
                m_pVehicle->m_wheelCollide)->m_phantom));

        world->addAction(m_pVehicle);

        m_pController->setInputStatus(
            (hkpVehicleDriverInputAnalogStatus*)(
                m_pVehicle->m_deviceStatus));

        m_pStopwatch = UmrStopwatchPtr(new UmrStopwatch());
    }
```

Listado 33. Construcción del vehículo

Para poder utilizar el vehículo tenemos que crear y configurar sus componentes, inicializar el vehículo mediante su método `init()` y añadir tanto el vehículo como los cuerpos que representan a sus ruedas al mundo físico. Adicionalmente también tenemos que establecer el enlace entre el controlador del vehículo de Havok y nuestro controlador y crear el cronometro para medir el tiempo por vuelta.

La instancia que utilizamos de *hkpVehicleInstance* contiene una serie de parámetros estáticos que no cambian a lo largo de la ejecución y que tenemos que rellenar para que el vehículo funcione correctamente:

```
/**
    Sets up vehicle data for the physics simulation.
    @param
        world Pointer to the physics world.
    @param
```

```
        data Reference to the class to store the
            vehicle's static information.
*/
void UmrVehicle::setupVehicleData(
    hkpWorld* world, hkpVehicleData& data)
{
    //
    // The vehicleData contains information about the chassis.
    //

    data.m_gravity = world->getGravity();

    hkVector4 up(0, 1, 0);
    hkVector4 forward(1, 0, 0);
    hkVector4 right(0, 0, 1);
    // The coordinates of the chassis system,
    // used for steering the vehicle.
    data.m_chassisOrientation.setCols(up, forward, right);

    // Specifies how the effect of dynamic load distribution is
    // averaged with static load distribution.
    data.m_frictionEqualizer = 0.5f;

    // Change this value to clip the normal in suspension force
    // calculations. In particular, when mounting a curb, the
    // raycast vehicle calculations can produce large impulses as
    // the rays hit the face of the step, causing the
    // vehicle to spin around its up-axis.
    data.m_normalClippingAngle = 0.2f;

    // Inertia tensor for each axis is calculated by using :
    // (1 / chassis_mass) * (torque(axis)Factor / chassisUnitInertia)
    data.m_torqueRollFactor = 0.625f;
    data.m_torquePitchFactor = 0.5f;
    data.m_torqueYawFactor = 0.4f;

    // Rotation intertia
    data.m_chassisUnitInertiaYaw = 1.0f;
    data.m_chassisUnitInertiaRoll = 1.0f;
    data.m_chassisUnitInertiaPitch = 1.0f;

    // Adds or removes torque around the yaw axis based on the
    // current steering angle. This will affect steering.
    data.m_extraTorqueFactor = -0.5f;

    data.m_maxVelocityForPositionalFriction = 180.0f;

    //
    // Wheel specifications
    //
    data.m_numWheels = 4;

    data.m_wheelParams.setSize( data.m_numWheels );

    // The axle the wheel is on
    data.m_wheelParams[0].m_axle = 0;
    data.m_wheelParams[1].m_axle = 0;
    data.m_wheelParams[2].m_axle = 1;
    data.m_wheelParams[3].m_axle = 1;
```

```
    // Wheel friction coefficient
    data.m_wheelParams[0].m_friction = 1.5f;
    data.m_wheelParams[1].m_friction = 1.5f;
    data.m_wheelParams[2].m_friction = 1.5f;
    data.m_wheelParams[3].m_friction = 1.5f;

    // The slip angle of the tyre for a car
    // where the cornering forces are one G
    data.m_wheelParams[0].m_slipAngle = 0.0f;
    data.m_wheelParams[1].m_slipAngle = 0.0f;
    data.m_wheelParams[2].m_slipAngle = 0.0f;
    data.m_wheelParams[3].m_slipAngle = 0.0f;

    for (int i = 0; i < data.m_numWheels; i++) {
        // This value is also used to calculate
        // the m_primaryTransmissionRatio.
        data.m_wheelParams[i].m_radius = 0.25f;
        data.m_wheelParams[i].m_width = 0.3f;
        data.m_wheelParams[i].m_mass = 10.0f;

        // An extra velocity dependent friction factor.
        // This factor allows us to increase the friction
        // if the car slides.
        data.m_wheelParams[i].m_viscosityFriction = 0.25f;

        // Clips the final friction
        data.m_wheelParams[i].m_maxFriction =
            2.0f * data.m_wheelParams[i].m_friction;

        data.m_wheelParams[i].m_forceFeedbackMultiplier = 0.1f;
        data.m_wheelParams[i].m_maxContactBodyAcceleration =
            hkReal(data.m_gravity.length3()) * 2;
    }
}
```

Listado 34. Configuración de parámetros del vehículo

Además de los parámetros anteriores tenemos que configurar individualmente cada componente del vehículo:

```
/**
    Sets up the driver input configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultAnalogDriverInput& driverInput)
{
    // We also use an analog "driver input" class to
    // help converting user input to vehicle behavior.

    // The initial slope. Used for small steering angles.
    driverInput.m_initialSlope = 0.7f;

    // The input value, up to which the m_initialSlope is valid.
    driverInput.m_slopeChangePointX = 0.8f;
```

```
    // The deadZone of the joystick.
    driverInput.m_deadZone = 0.0f;

    // If true, the car will start reversing when
    // the brake is applied and the car is stopped.
    driverInput.m_autoReverse = true;
}

/**
    Sets up the vehicle steering configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultSteering& steering)
{
    steering.m_doesWheelSteer.setSize(data.m_numWheels);

    // degrees
    steering.m_maxSteeringAngle = MAX_STEERING * (HK_REAL_PI / 180);

    // The steering angle decreases linearly based
    // on your overall max speed of the vehicle.
    steering.m_maxSpeedFullSteeringAngle = 70.0f * (1.605f / 3.6f);
    steering.m_doesWheelSteer[0] = true;
    steering.m_doesWheelSteer[1] = true;
    steering.m_doesWheelSteer[2] = false;
    steering.m_doesWheelSteer[3] = false;
}

/**
    Sets up the vehicle engine configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultEngine& engine)
{
    // The maximum gross torque the engine
    // can supply at the optimum RPM.
    engine.m_maxTorque = 600.0f;

    engine.m_minRPM = 1000.0f;
    // The optimum RPM, where the gross
    // torque of the engine is maximal.
    engine.m_optRPM = 5500.0f;

    // This value is also used to calculate
    // the m_primaryTransmissionRatio.
    engine.m_maxRPM = 7500.0f;

    // Defines the gross torque at the min/max rpm as
    // a factor to the torque at optimal RPM.
    engine.m_torqueFactorAtMinRPM = 0.8f;
    engine.m_torqueFactorAtMaxRPM = 0.8f;

    // Defines the engine resistance torque at the min/opt/max rpm
    // as a factor to the torque at optimal RPM.
    engine.m_resistanceFactorAtMinRPM = 0.05f;
    engine.m_resistanceFactorAtOptRPM = 0.1f;
    engine.m_resistanceFactorAtMaxRPM = 0.3f;
```

```
}

/**
    Sets up the vehicle transmission configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultTransmission& transmission)
{
    int numGears = 4;

    transmission.m_gearsRatio.setSize( numGears );
    transmission.m_wheelsTorqueRatio.setSize( data.m_numWheels );

    // The RPM of the engine the transmission shifts down and up
    transmission.m_downshiftRPM = 3500.0f;
    transmission.m_upshiftRPM = 6500.0f;

    // The time needed [seconds] to shift a gear
    transmission.m_clutchDelayTime = 0.0f;

    // The back gear ratio
    transmission.m_reverseGearRatio = 1.0f;

    // The ratio of the forward gears
    transmission.m_gearsRatio[0] = 2.0f;
    transmission.m_gearsRatio[1] = 1.5f;
    transmission.m_gearsRatio[2] = 1.0f;
    transmission.m_gearsRatio[3] = 0.75f;

    // The transmission ratio for every wheel
    transmission.m_wheelsTorqueRatio[0] = 0.5f;
    transmission.m_wheelsTorqueRatio[1] = 0.5f;
    transmission.m_wheelsTorqueRatio[2] = 0.0f;
    transmission.m_wheelsTorqueRatio[3] = 0.0f;

    const hkReal vehicleTopSpeed = 180.0f;
    const hkReal wheelRadius = 0.25f;
    const hkReal maxEngineRpm = 7500.0f;
    // An extra factor to the gear ratio
    transmission.m_primaryTransmissionRatio =
    hkpVehicleDefaultTransmission::calculatePrimaryTransmissionRatio(
            vehicleTopSpeed,
            wheelRadius,
            maxEngineRpm,
            transmission.m_gearsRatio[numGears - 1]);
}

/**
    Sets up the vehicle brake configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultBrake& brake)
{
    brake.m_wheelBrakingProperties.setSize( data.m_numWheels );

    const float bt = 1500.0f;
    // The maximum torque the wheel can apply when braking
```

```
    brake.m_wheelBrakingProperties[0].m_maxBreakingTorque = bt;
    brake.m_wheelBrakingProperties[1].m_maxBreakingTorque = bt;
    brake.m_wheelBrakingProperties[2].m_maxBreakingTorque = bt;
    brake.m_wheelBrakingProperties[3].m_maxBreakingTorque = bt;

    // Handbrake is attached to rear wheels only
    brake.m_wheelBrakingProperties[0].m_isConnectedToHandbrake =false;
    brake.m_wheelBrakingProperties[1].m_isConnectedToHandbrake =false;
    brake.m_wheelBrakingProperties[2].m_isConnectedToHandbrake = true;
    brake.m_wheelBrakingProperties[3].m_isConnectedToHandbrake = true;

    // The minimum amount of braking from the driver
    // that could cause the wheel to block.
    brake.m_wheelBrakingProperties[0].m_minPedalInputToBlock = 0.9f;
    brake.m_wheelBrakingProperties[1].m_minPedalInputToBlock = 0.9f;
    brake.m_wheelBrakingProperties[2].m_minPedalInputToBlock = 0.9f;
    brake.m_wheelBrakingProperties[3].m_minPedalInputToBlock = 0.9f;

    // The time (in secs) after which, if the user
    // applies enough brake input wheel will block.
    brake.m_wheelsMinTimeToBlock = 1000.0f;
}

/**
    Sets up the vehicle suspension configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultSuspension& suspension)
{
    suspension.m_wheelParams.setSize( data.m_numWheels );
    suspension.m_wheelSpringParams.setSize( data.m_numWheels );

    // The suspension length at rest i.e. the maximum distance
    // from the hardpoint to the wheel center.
    suspension.m_wheelParams[0].m_length = 0.5f;
    suspension.m_wheelParams[1].m_length = 0.5f;
    suspension.m_wheelParams[2].m_length = 0.5f;
    suspension.m_wheelParams[3].m_length = 0.5f;

    const float str = 50.0f;
    // The strength [N/m] of the suspension at each wheel
    suspension.m_wheelSpringParams[0].m_strength = str;
    suspension.m_wheelSpringParams[1].m_strength = str;
    suspension.m_wheelSpringParams[2].m_strength = str;
    suspension.m_wheelSpringParams[3].m_strength = str;

    const float wd = 3.0f;
    // The damping force [N/(m/sec)] of the suspension at each wheel
    suspension.m_wheelSpringParams[0].m_dampingCompression = wd;
    suspension.m_wheelSpringParams[1].m_dampingCompression = wd;
    suspension.m_wheelSpringParams[2].m_dampingCompression = wd;
    suspension.m_wheelSpringParams[3].m_dampingCompression = wd;

    suspension.m_wheelSpringParams[0].m_dampingRelaxation = wd;
    suspension.m_wheelSpringParams[1].m_dampingRelaxation = wd;
    suspension.m_wheelSpringParams[2].m_dampingRelaxation = wd;
    suspension.m_wheelSpringParams[3].m_dampingRelaxation = wd;
```

```
    // The hardpoints MUST be positioned INSIDE the chassis
    {
        const hkReal hardPointFrontX = 1.2f;
        const hkReal hardPointBackX = -1.2f;
        const hkReal hardPointY = 0.1f;
        const hkReal hardPointZ = 0.85f;

        // A point INSIDE the chassis to which
        // the wheel suspension is attached.
        suspension.m_wheelParams[0].m_hardpointChassisSpace.set(
            hardPointFrontX, hardPointY, -hardPointZ);
        suspension.m_wheelParams[1].m_hardpointChassisSpace.set(
            hardPointFrontX, hardPointY,  hardPointZ);
        suspension.m_wheelParams[2].m_hardpointChassisSpace.set(
            hardPointBackX, hardPointY, -hardPointZ);
        suspension.m_wheelParams[3].m_hardpointChassisSpace.set(
            hardPointBackX, hardPointY,  hardPointZ);
    }

    const hkVector4 downDirection( 0.0f, -1.0f, 0.0f );
    // The suspension direction (in Chassis Space)
    suspension.m_wheelParams[0].m_directionChassisSpace =
        downDirection;
    suspension.m_wheelParams[1].m_directionChassisSpace =
        downDirection;
    suspension.m_wheelParams[2].m_directionChassisSpace =
        downDirection;
    suspension.m_wheelParams[3].m_directionChassisSpace =
        downDirection;
}

/**
    Sets up the vehicle aerodynamics configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
    hkpVehicleDefaultAerodynamics& aerodynamics)
{
    // The density of the air that surrounds
    // the vehicle, usually, 1.3 kg/m3.
    aerodynamics.m_airDensity = 1.3f;

    // The frontal area, in m2, of the car.
    aerodynamics.m_frontalArea = 1.0f;

    // The drag and lift coefficients of the car.
    aerodynamics.m_dragCoefficient = 0.7f;
    aerodynamics.m_liftCoefficient = -0.3f;

    // Extra gavity applies in world space
    // (independent of m_chassisCoordinateSystem).
    aerodynamics.m_extraGravityws.set(0.0f, -5.0f, 0.0f);
}

/**
    Sets up the vehicle velocity damper configuration.
*/
void UmrVehicle::setupComponent(
    const hkpVehicleData& data,
```

```
    hkpVehicleDefaultVelocityDamper& velocityDamper)
{
    // Caution: setting negative damping values
    // will add energy to system. Setting the value
    // to 0 will not affect the angular velocity.

    // Damping the change of the chassis' angular
    // velocity when below m_collisionThreshold.
    // This will affect turning radius and steering.
    velocityDamper.m_normalSpinDamping    = 0.0f;

    // Positive numbers dampen the rotation of the chassis and
    // reduce the reaction of the chassis in a collision.
    velocityDamper.m_collisionSpinDamping = 4.0f;

    // The threshold in m/s at which the algorithm switches from
    // using the normalSpinDamping to the collisionSpinDamping.
    velocityDamper.m_collisionThreshold   = 1.0f;
}

/**
    Sets up the vehicle wheel colliding configuration.
*/
void UmrVehicle::setupWheelCollide(
    hkpWorld* world,
    const hkpVehicleInstance& vehicle,
    hkpVehicleRaycastWheelCollide& wheelCollide)
{
    wheelCollide.m_wheelCollisionFilterInfo =
      vehicle.getChassis()->getCollidable()->getCollisionFilterInfo();
}
```

Listado 35. Configuración de los componentes del vehículo

6.6.5.4 Actualización de la física

La actualización de la física se realiza en una hebra diferente a la principal, de esta forma cuando ejecutemos el juego en una plataforma multiprocesador se observará una ganancia considerable de rendimiento.

Como introdujimos en un apartado anterior, la actualización de la física junto con la de la entrada del usuario se produce en la clase *UmrUpdateThread*:

```
/**
    Constructor.
    @param
        mutex Mutex for graphics and physics updates synchronization.
*/
UmrUpdateThread::UmrUpdateThread(UmrMutexPtr &mutex) : m_pMutex(mutex)
{
    m_pPhysics = UmrPhysics::getInstance();
}

/**
    Destructor.
```

```
*/
UmrUpdateThread::~UmrUpdateThread(void)
{
}

/**
    Sets the vehicles to simulate.
*/
void UmrUpdateThread::setVehicles(UmrVehiclePtr (&vehicles)[NCARS])
{
    for (int i = 0; i < NCARS; i++)    {
        m_vehicles[i] = vehicles[i];
    }
}

/**
    Executes the thread's code.
*/
unsigned UmrUpdateThread::run()
{
    TimerPtr timer = TimerPtr(new Ogre::Timer());
    ULONG elapsedTime, timeLastUpdate = 0;
    bool quit = 0;

    // Init physics thread before performing
    // any operation on the world.
    m_pPhysics->initThread();

    timer->reset();
    while(!m_isDying && !m_quit) {
        elapsedTime = timer->getMilliseconds() - timeLastUpdate;
        if (elapsedTime < TIME_STEP_MS) {
            Sleep(TIME_STEP_MS - elapsedTime);
        }
        // Only human player can quit the game
        quit = m_vehicles[0]->updateControl();
        if (quit) {
            _InterlockedIncrement(&m_quit);
            break;
        }
        for (int i = 1; i < NCARS; i++) {
            m_vehicles[i]->updateControl();
        }

        timeLastUpdate = timer->getMilliseconds();
        // Step physics when it's safe, after acquiring mutex
        m_pMutex->acquire();
            m_pPhysics->stepPhysics();
        m_pMutex->release();
    }

    m_pPhysics->quitThread();
    return 0;
}

/**
    Sets the quit flag for next update.
*/
const bool UmrUpdateThread::stopped() const
```

```
{
    return m_quit != 0;
}
```

Listado 36. Definición de UmrUpdateThread

6.6.6 Gestión de entrada

La gestión de la entrada es un importante aspecto del sistema, ya que es la única vía de comunicación del usuario con el juego y el vehículo. Utilizando un buen diseño podemos aprovechar la estructura de clases de la gestión de entrada para comunicar a un agente inteligente con el control del vehículo tal y como se haría con un jugador normal.

Lo primero es implementar la interfaz común para todos los controladores:

```
/**
    Controller class.
    Base class for any class able to control a vehicle.
    This class is an abstraction for the input of a driver
    to the controls of the vehicle.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrController : public UmrObject
{
public:
    UmrController (void) {}
    virtual ~UmrController (void) {}

    virtual bool updateState (float stepTime) = 0;
    const bool isActive () const { return m_active; }
    const float getXPosition () const { return m_inputXPosition; }
    const float getYPosition () const { return m_inputYPosition; }
    const bool getTrainNNetwork () const { return m_trainNNetwork; }
    virtual void setInputStatus (
        hkpVehicleDriverInputAnalogStatus* status) = 0;
protected:
    virtual void processInput(float stepTime) {}

    /// Container for the input values obtained from the contoller.
    hkpVehicleDriverInputAnalogStatus* m_pControllerStatus;

    /// Whether the controller is active or not.
    bool    m_active;

    /// True when the vehicle should record training data.
    bool    m_trainNNetwork;

    /// The current input for the steering wheel
    float    m_inputXPosition;

    /// The current input for the acceleration and brake pedal
    float    m_inputYPosition;
};
```

```
typedef boost::shared_ptr<UmrController> UmrControllerPtr;
```

Listado 37. Declaración de UmrController

Una vez tenemos la interfaz, debemos realizar su implementación para cada tipo de controlador. Hay que destacar que el vehículo de Havok que usamos utiliza dos parámetros para controlarlo:

- Giro del volante. Valor entre -1 y 1 mapeado sobre el eje de coordenadas X.
- Aceleración. Valor entre -1 y 1 mapeado sobre el eje de coordenadas Y.

En cada actualización, iniciada por el método `updateControl()` de *UmrVehicle*, tendremos que recoger la entrada del usuario y actualizar la entrada del vehículo.

6.6.6.1 Teclado

En la implementación del controlador de teclado utilizamos la API de DirectInput, lo que conlleva un proceso definido para utilizar el dispositivo en la aplicación.

Además implementamos un mecanismo de mapeado de acciones, de forma que podemos definir en fichero de configuración la tecla asociada a una acción determinada.

```
#define KEYDOWN(name, key) (name[key] & 0x80)

#define CLAMP(value, min, max) ((value <= min) ? min : ((value >= max) ? 
max : value))

#define MAX_ACTIONS    9
#define ACCELERATE    0
#define BRAKE        1
#define STEER_LEFT    2
#define STEER_RIGHT    3
#define SHIFT_UP    4
#define SHIFT_DOWN    5
#define HAND_BRAKE    6
#define QUIT        7
#define TRAIN        8

using namespace std;

/**
    Describes the state of certain input actions.
*/
__declspec(align(MEM_ALIGNMENT))
struct UmrKeyboardInputState : public UmrObject
{
    bool    accelerate;
    bool    brake;
```

```
    bool    steerLeft;
    bool    steerRight;
    bool    shiftUp;
    bool    shiftDown;
    bool    handBrake;
};

/**
    Keyboard input class.
    Provides the input to the controller through a keyboard device.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrKeyboardInput : public UmrController
{
public:
    UmrKeyboardInput(const Ogre::ConfigFile &cf);
    ~UmrKeyboardInput(void);

    virtual bool updateState(float timeStep);
    virtual void setInputStatus(
        hkpVehicleDriverInputAnalogStatus* status);

protected:
    virtual void processInput(float timeStep);
    void setupKeyMap();
    void setupActionMap(const Ogre::ConfigFile &cf);

    /// Pointer to the DirectInput subsystem
    LPDIRECTINPUT8            m_pDI;

    /// Pointer to the joystick input device
    LPDIRECTINPUTDEVICE8    m_pKeyboard;

    /// Last state of the device
    UmrKeyboardInputState    m_state;

    /// Holds the key binding for every input action
    byte                    m_actionMap[MAX_ACTIONS];

    /// Maps a string to a DirectInput key definition
    map<string, byte>        m_keyMap;

    /// True to quit the application
    bool                    m_quit;
};

typedef boost::shared_ptr<UmrKeyboardInput> UmrInputPtr;
```

Listado 38. Declaración de UmrKeyboardInput

```
// Two last keyboard device states
char g_keyboardState[256];
char g_lastKeyboardState[256];

/**
    Constructor.
*/
UmrKeyboardInput::UmrKeyboardInput(const Ogre::ConfigFile &cf) :
```

```
    m_pDI(NULL), m_pKeyboard(NULL), m_quit(false)
{
    HWND hWnd;
    DWORD _dwCoopFlags = DISCL_EXCLUSIVE | DISCL_FOREGROUND;
    HRESULT _hr;

    m_inputXPosition = 0;
    m_inputYPosition = 0;
    //m_trainNNetwork = true;

    // Get window handle
    hWnd = FindWindow("OgreD3D9Wnd", TITLE);
    if (!IsWindow(hWnd))
        hWnd = FindWindow("OgreGLWindow", TITLE);

    // Create a DInput object
    _hr = DirectInput8Create(
        GetModuleHandle(NULL), DIRECTINPUT_VERSION,
        IID_IDirectInput8, ( VOID** )&m_pDI, NULL);
    if (FAILED(_hr)) {
        m_quit = true;
        return;
    }

    // Obtain an interface to the system keyboard device.
    _hr = m_pDI->CreateDevice(GUID_SysKeyboard, &m_pKeyboard, NULL);
    if (FAILED(_hr)) {
        m_quit = true;
        return;
    }

    // Set the data format to "keyboard format" - a predefined
    // data format.
    //
    // A data format specifies which controls on a device we
    // are interested in, and how they should be reported.
    //
    // This tells DirectInput that we will be passing an array
    // of 256 bytes to IDirectInputDevice::GetDeviceState.
    _hr = m_pKeyboard->SetDataFormat(&c_dfDIKeyboard);
    if (FAILED(_hr)) {
        m_quit = true;
        return;
    }

    // Set the cooperativity level to let DirectInput know how
    // this device should interact with the system and with other
    // DirectInput applications.
    _hr = m_pKeyboard->SetCooperativeLevel(hWnd, _dwCoopFlags);
    if (FAILED(_hr)) {
        m_quit = true;
        return;
    }

    // Acquire the newly created device
    m_pKeyboard->Acquire();

    ZeroMemory(g_keyboardState, sizeof(g_keyboardState));
    ZeroMemory(&m_state, sizeof(m_state));
```

```
    setupKeyMap();
    setupActionMap(cf);
}

/**
    Destructor.
*/
UmrKeyboardInput::~UmrKeyboardInput(void)
{
    m_pControllerStatus->removeReference();
    if (m_pDI) {
        if (m_pKeyboard) {
            // Always unacquire device before calling Release()
            m_pKeyboard->Unacquire();
            m_pKeyboard->Release();
            m_pKeyboard = NULL;
        }
        m_pDI->Release();
        m_pDI = NULL;
    }
}

/**
    Sets the physics input controller.
*/
void UmrKeyboardInput::setInputStatus(
    hkpVehicleDriverInputAnalogStatus* status)
{
    m_pControllerStatus = status;
    m_pControllerStatus->addReference();
}

/**
    Updates the input state of the keyboard.
*/
void UmrKeyboardInput::processInput(float timeStep)
{
    HRESULT hr;

    memcpy(g_lastKeyboardState, g_keyboardState, sizeof(char)*256);
    hr = m_pKeyboard->GetDeviceState(
            sizeof(g_keyboardState),(LPVOID)&g_keyboardState);
    if (FAILED(hr)) {
        hr = m_pKeyboard->Acquire();
        while(hr == DIERR_INPUTLOST)
            hr = m_pKeyboard->Acquire();

        // hr may be DIERR_OTHERAPPHASPRIO or other errors.  This
        // may occur when the app is minimized or in the process of
        // switching, so just try again later
        return;
    }

    m_quit = KEYDOWN(g_keyboardState, m_actionMap[QUIT]) != 0;
    if (m_quit) return;

    m_state.accelerate =
        KEYDOWN(g_keyboardState, m_actionMap[ACCELERATE]) != 0;
```

```
    m_state.brake =
        KEYDOWN(g_keyboardState, m_actionMap[BRAKE]) != 0;

    m_state.steerLeft =
        KEYDOWN(g_keyboardState, m_actionMap[STEER_LEFT]) != 0;

    m_state.steerRight =
        KEYDOWN(g_keyboardState, m_actionMap[STEER_RIGHT]) != 0;

    m_state.handBrake =
        KEYDOWN(g_keyboardState, m_actionMap[HAND_BRAKE]) != 0;

    if (KEYDOWN(g_keyboardState, m_actionMap[TRAIN]) == 0 &&
        KEYDOWN(g_lastKeyboardState, m_actionMap[TRAIN]) != 0)
        m_trainNNetwork = !m_trainNNetwork;
}

/**
    Updates the state of the controller.
*/
bool UmrKeyboardInput::updateState(float timeStep)
{
    const float steerSpeed    =  3.0f * timeStep;
    const float backSteerSpeed = 10.0f * timeStep;
    hkReal deltaY = -m_inputYPosition  * 0.2f;
    hkReal deltaX = -m_inputXPosition  * backSteerSpeed;

    if (m_pControllerStatus == NULL) return false;

    processInput(timeStep);
    if (m_quit) return true;

    m_active = m_state.accelerate || m_state.brake ||
               m_state.handBrake || m_state.steerRight ||
               m_state.steerLeft;

    if (m_state.accelerate) deltaY = -0.1f;
    else if (m_state.brake) deltaY = 0.1f;

    if (m_state.steerRight) {
        if (m_inputXPosition >= 0.0f) {    deltaX = 0.0f;    }
        deltaX += steerSpeed;
    } else if (m_state.steerLeft) {
        if ( m_inputXPosition <= 0.0f){    deltaX = 0.0f;    }
        deltaX -= steerSpeed;
    }

    m_inputXPosition =
        hkMath::clamp( m_inputXPosition+deltaX, -1.0f, 1.0f);
    m_inputYPosition =
        hkMath::clamp( m_inputYPosition+deltaY, -1.0f, 1.0f);

    // Now  -1 <= m_inputXPosition <= 1 and
    //        -1 <= m_inputYPosition <= 1
    m_pControllerStatus->m_positionX = m_inputXPosition;
    m_pControllerStatus->m_positionY = m_inputYPosition;

    m_pControllerStatus->m_handbrakeButtonPressed = m_state.handBrake;
```

```
    return false;
}

/**
    Sets up the keys mapping.
    @remarks
        Each key string is mapped to its DirectInput definition.
*/
void UmrKeyboardInput::setupKeyMap()
{
    m_keyMap["ESCAPE"] = DIK_ESCAPE;
    m_keyMap["A"] = DIK_A;
    m_keyMap["B"] = DIK_B;
    m_keyMap["C"] = DIK_C;
    m_keyMap["D"] = DIK_D;
    m_keyMap["E"] = DIK_E;
    m_keyMap["F"] = DIK_F;
    m_keyMap["G"] = DIK_G;
    m_keyMap["H"] = DIK_H;
    m_keyMap["I"] = DIK_I;
    m_keyMap["J"] = DIK_J;
    m_keyMap["K"] = DIK_K;
    m_keyMap["L"] = DIK_L;
    m_keyMap["M"] = DIK_M;
    m_keyMap["N"] = DIK_N;
    m_keyMap["O"] = DIK_O;
    m_keyMap["P"] = DIK_P;
    m_keyMap["Q"] = DIK_Q;
    m_keyMap["R"] = DIK_R;
    m_keyMap["S"] = DIK_S;
    m_keyMap["T"] = DIK_T;
    m_keyMap["U"] = DIK_U;
    m_keyMap["V"] = DIK_V;
    m_keyMap["W"] = DIK_W;
    m_keyMap["X"] = DIK_X;
    m_keyMap["Y"] = DIK_Y;
    m_keyMap["Z"] = DIK_Z;
    m_keyMap["1"] = DIK_1;
    m_keyMap["2"] = DIK_2;
    m_keyMap["3"] = DIK_3;
    m_keyMap["4"] = DIK_4;
    m_keyMap["5"] = DIK_5;
    m_keyMap["6"] = DIK_6;
    m_keyMap["7"] = DIK_7;
    m_keyMap["8"] = DIK_8;
    m_keyMap["9"] = DIK_9;
    m_keyMap["0"] = DIK_0;
    m_keyMap["LCONTROL"] = DIK_LCONTROL;
    m_keyMap["LSHIFT"] = DIK_LSHIFT;
    m_keyMap["TAB"] = DIK_TAB;
    m_keyMap["LALT"] = DIK_LALT;
    m_keyMap["SPACE"] = DIK_SPACE;
    m_keyMap["RALT"] = DIK_RALT;
    m_keyMap["RCONTROL"] = DIK_RCONTROL;
    m_keyMap["COMMA"] = DIK_COMMA;
    m_keyMap["PERIOD"] = DIK_PERIOD;
    m_keyMap["RSHIFT"] = DIK_RSHIFT;
    m_keyMap["ENTER"] = DIK_RETURN;
```

```
    m_keyMap["BACKSPACE"] = DIK_BACKSPACE;
    m_keyMap["HOME"] = DIK_HOME;
    m_keyMap["END"] = DIK_END;
    m_keyMap["DELETE"] = DIK_DELETE;
    m_keyMap["PRIOR"] = DIK_PRIOR;
    m_keyMap["NEXT"] = DIK_NEXT;
    m_keyMap["UP"] = DIK_UP;
    m_keyMap["DOWN"] = DIK_DOWN;
    m_keyMap["LEFT"] = DIK_LEFT;
    m_keyMap["RIGHT"] = DIK_RIGHT;
    m_keyMap["F1"] = DIK_F1;
    m_keyMap["F2"] = DIK_F2;
    m_keyMap["F3"] = DIK_F3;
    m_keyMap["F4"] = DIK_F4;
    m_keyMap["F5"] = DIK_F5;
    m_keyMap["F6"] = DIK_F6;
    m_keyMap["F7"] = DIK_F7;
    m_keyMap["F8"] = DIK_F8;
    m_keyMap["F9"] = DIK_F9;
    m_keyMap["F10"] = DIK_F10;
    m_keyMap["F11"] = DIK_F11;
    m_keyMap["F12"] = DIK_F12;
}

/**
    Reads the key binding configuration from a file.
*/
void UmrKeyboardInput::setupActionMap(const Ogre::ConfigFile &cf)
{
    string key;

    key = cf.getSetting("Accelerate", "Keyboard", "UP");
    m_actionMap[ACCELERATE] = m_keyMap[key];

    key = cf.getSetting("Brake", "Keyboard", "DOWN");
    m_actionMap[BRAKE] = m_keyMap[key];

    key = cf.getSetting("SteerLeft", "Keyboard", "LEFT");
    m_actionMap[STEER_LEFT] = m_keyMap[key];

    key = cf.getSetting("SteerRight", "Keyboard", "RIGHT");
    m_actionMap[STEER_RIGHT] = m_keyMap[key];

    key = cf.getSetting("HandBrake", "Keyboard", "SPACE");
    m_actionMap[HAND_BRAKE] = m_keyMap[key];

    key = cf.getSetting("Quit", "Keyboard", "ESCAPE");
    m_actionMap[QUIT] = m_keyMap[key];

    key = cf.getSetting("Train", "Keyboard", "T");
    m_actionMap[TRAIN] = m_keyMap[key];
}
```

Listado 39. Definición de UmrKeyboardInput

DirecInput define un valor para identificar cada tecla, por lo que para implementar el mecanismo de mapeado de acciones asociamos a la definición de la tecla a una cadena de

texto que la representa. Con esta asociación podemos relacionar acciones determinadas a la cadena de texto de cada tecla.

Una vez configurado el dispositivo de DirectInput para el teclado podemos acceder al fichero `input.cfg` donde tenemos almacenadas las asociaciones de teclas.

En el método `updateState()` realizamos la actualización del estado de las acciones y utilizando parámetros tales como la velocidad de giro del volante, la velocidad de retroceso del volante y un incremento fijo de giro y aceleración actualizamos el estado del controlador del vehículo.

6.6.6.2 Joystick

Al igual que con el teclado, para utilizar un joystick hay que realizar una serie de pasos para configurar y tener acceso al dispositivo de DirectInput.

```
/**
    Joystick keys binding definition.
*/
__declspec(align(MEM_ALIGNMENT))
struct UmrJoystickKeys : public UmrObject
{
    unsigned handbrake;
    unsigned train;
    unsigned quit;
};

/**
    Joystick input class.
    Provides the input to the controller through a joystick device.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrJoystickInput : public UmrController
{
public:
    UmrJoystickInput (const Ogre::ConfigFile &cf);
    ~UmrJoystickInput (void);

    virtual bool updateState(float timeStep);
    virtual void setInputStatus(
        hkpVehicleDriverInputAnalogStatus* status);
protected:
    static int CALLBACK enumJoystickCallback(
        const DIDEVICEINSTANCE* pdidInstance,
        VOID* pContext);
    static int CALLBACK enumObjectsCallback(
        const DIDEVICEOBJECTINSTANCE* pdidoi,
        VOID* pContext);
    virtual void processInput(float timeStep);
    void setupKeys (const Ogre::ConfigFile &cf);
```

```
    /// Pointer to the DirectInput subsystem
    LPDIRECTINPUT8 m_pDI;

    /// Pointer to the joystick input device
    LPDIRECTINPUTDEVICE8 m_pJoystick;

    /// Structures holding the two last joystick device states
    DIJOYSTATE2    m_state;
    DIJOYSTATE2    m_lastState;

    /// True to quit the application
    bool m_quit;

    UmrJoystickKeys m_keys;
};

typedef boost::shared_ptr<UmrJoystickInput> UmrJoystickInputPtr;
```

Listado 40. Declaración de UmrJoystickInput

```
/**
    Constructor.
*/
UmrJoystickInput::UmrJoystickInput(const Ogre::ConfigFile &cf) :
    m_pDI(NULL), m_pJoystick(NULL), m_quit(false)
{
    HWND hWnd;
    HRESULT hr;

    m_pControllerStatus = HK_NULL;
    m_inputXPosition = 0;
    m_inputYPosition = 0;
    m_active = true;
    m_trainNNetwork = false;

    // Get window handle
    hWnd = FindWindow("OgreD3D9Wnd", TITLE);
    if (!IsWindow(hWnd))
        hWnd = FindWindow("OgreGLWindow", TITLE);

    // Register with the DirectInput subsystem and get a pointer
    // to a IDirectInput interface we can use.
    // Create a DInput object
    hr = DirectInput8Create(GetModuleHandle(NULL),DIRECTINPUT_VERSION,
        IID_IDirectInput8, (VOID**)&m_pDI, NULL);
    if (FAILED(hr)) {
        m_quit = true;
        return;
    }

    // Look for a simple joystick we can use for this sample program.
    hr = m_pDI->EnumDevices(DI8DEVCLASS_GAMECTRL,
        &UmrJoystickInput::enumJoystickCallback,
        this, DIEDFL_ATTACHEDONLY);
    if (FAILED(hr) || m_pJoystick == NULL) {
        m_quit = true;
        return;
```

```
    }

    // Set the data format to "simple joystick" - a predefined
    // data format.
    //
    // A data format specifies which controls on a device we
    // are interested in, and how they should be reported. This
    // tells DInput that we will be passing a DIJOYSTATE2 structure
    // to IDirectInputDevice::GetDeviceState().
    hr = m_pJoystick->SetDataFormat(&c_dfDIJoystick2);
    if (FAILED(hr)) {
        m_quit = true;
        return;
    }

    // Set the cooperative level to let DInput
    // know how this device should interact with
    // the system and with other DInput applications.
    hr = m_pJoystick->SetCooperativeLevel(
            hWnd, DISCL_EXCLUSIVE | DISCL_FOREGROUND);
    if (FAILED(hr)) {
        m_quit = true;
        return;
    }

    // Enumerate the joystick objects. The callback
    // function enabled user interface elements for
    // objects that are found, and sets the min/max
    // values property for discovered axes.
    hr = m_pJoystick->EnumObjects(
        &UmrJoystickInput::enumObjectsCallback,
        this, DIDFT_ALL);
    if (FAILED(hr)) {
        m_quit = true;
        return;
    }

    m_pJoystick->Acquire();

    setupKeys(cf);
}

/**
    Destructor.
*/
UmrJoystickInput::~UmrJoystickInput(void)
{
    if (m_pControllerStatus != HK_NULL)
        m_pControllerStatus->removeReference();
    if (m_pDI) {
        if (m_pJoystick) {
            // Always unacquire device before calling Release()
            m_pJoystick->Unacquire();
            m_pJoystick->Release();
            m_pJoystick = NULL;
        }
        m_pDI->Release();
        m_pDI = NULL;
    }
```

```
}

/**
    Sets the physics input controller.
*/
void UmrJoystickInput::setInputStatus(
    hkpVehicleDriverInputAnalogStatus* status)
{
    m_pControllerStatus = status;
    m_pControllerStatus->addReference();
//    m_pControllerStatus->m_handbrakeButtonPressed = false;
//    m_pControllerStatus->m_reverseButtonPressed = false;
}

/**
    Callback function called once for each enumerated joystick. If we
    find one, create a device interface on it so we can play with it.
    @param
        pdidInstance Pointer to an instance of a DirectInput device.
    @param
        pContext Pointer to this class.
*/
int CALLBACK UmrJoystickInput::enumJoystickCallback(
    const DIDEVICEINSTANCE *pdidInstance, void *pContext)
{
    UmrJoystickInput* pThis = (UmrJoystickInput*)pContext;
    HRESULT hr;

    // Obtain an interface to the enumerated joystick.
    hr = pThis->m_pDI->CreateDevice(
            pdidInstance->guidInstance, &(pThis->m_pJoystick), NULL);

    // If it failed, then we can't use this joystick.
    // (Maybe the user unplugged it while we were in
    // the middle of enumerating it.)
    if( FAILED( hr ) )
        return DIENUM_CONTINUE;

    // Stop enumeration. Note: we're just taking the
    // first joystick we get. You could store all the
    // enumerated joysticks and let the user pick.
    return DIENUM_STOP;
}

/**
    Callback function for enumerating objects (axes, buttons, POVs)
    on a joystick. This function enables user interface elements for
    objects    that are found to exist, and scales axes min/max values.
    @param
        pdidoi Pointer to a device object instance.
    @param
        pContext Pointer to this class.
*/
int CALLBACK UmrJoystickInput::enumObjectsCallback(
    const DIDEVICEOBJECTINSTANCE* pdidoi, VOID* pContext)
{
    UmrJoystickInput* pThis = (UmrJoystickInput*)pContext;

    // Number of returned slider controls
```

```
    static int nSliderCount = 0;
    // Number of returned POV controls
    static int nPOVCount = 0;

    // For axes that are returned, set the DIPROP_RANGE property
    // for the enumerated axis in order to scale min/max values.
    if( pdidoi->dwType & DIDFT_AXIS )
    {
        DIPROPRANGE diprg;
        diprg.diph.dwSize = sizeof( DIPROPRANGE );
        diprg.diph.dwHeaderSize = sizeof( DIPROPHEADER );
        diprg.diph.dwHow = DIPH_BYID;
        // Specify the enumerated axis
        diprg.diph.dwObj = pdidoi->dwType;
        diprg.lMin = -1000;
        diprg.lMax = +1000;

        // Set the range for the axis
        if(FAILED(pThis->m_pJoystick->SetProperty(
                    DIPROP_RANGE, &diprg.diph)))
            return DIENUM_STOP;
    }
    return DIENUM_CONTINUE;
}

void UmrJoystickInput::setupKeys(const Ogre::ConfigFile &cf)
{
    m_keys.handbrake = Ogre::StringConverter::parseUnsignedInt(
                        cf.getSetting("HandBrake", "Joystick", "7"));
    m_keys.train = Ogre::StringConverter::parseUnsignedInt(
                        cf.getSetting("Train", "Joystick", "0"));
    m_keys.quit = Ogre::StringConverter::parseUnsignedInt(
                        cf.getSetting("Quit", "Joystick", "3"));
}

/**
    Updates the input state of the joystick.
*/
void UmrJoystickInput::processInput(float timeStep)
{
    HRESULT hr;

    if (m_pJoystick == NULL) return;

    // Poll the device to read the current state
    hr = m_pJoystick->Poll();
    if(FAILED(hr))
    {
        // DInput is telling us that the input stream has been
        // interrupted. We aren't tracking any state between polls,
        // so we don't have any special reset that needs to be done.
        // We just re-acquire and try again.
        hr = m_pJoystick->Acquire();
        while(hr == DIERR_INPUTLOST)
            hr = m_pJoystick->Acquire();

        // hr may be DIERR_OTHERAPPHASPRIO or other errors.  This
        // may occur when the app is minimized or in the process of
        // switching, so just try again later
```

```
        return;
    }

    memcpy(&m_lastState, &m_state, sizeof(DIJOYSTATE2));
    // Get the input's device state
    if(FAILED(hr = m_pJoystick->GetDeviceState(
                        sizeof(DIJOYSTATE2), &m_state)))
        // The device should have been acquired during the Poll()
        return;
    m_quit = m_state.rgbButtons[m_keys.quit] != 0;
}

/**
    Updates the state of the controller.
*/
bool UmrJoystickInput::updateState(float timeStep)
{
    if (m_pControllerStatus == NULL) return false;

    processInput(timeStep);

    if (m_quit) return true;

    float inputX = float(m_state.lX) * 0.001f;
    // Map acceleration to positive values and braking to negative
    // values since they come from different joystick axis
    float accel = float(m_state.lY-1000) * 0.0005f;
    float brake = float(m_state.lRz-1000) * -0.0005f;
    float inputY = accel + brake;

    m_inputXPosition = inputX;
    m_inputYPosition = inputY;
    m_pControllerStatus->m_positionX = m_inputXPosition;
    m_pControllerStatus->m_positionY = m_inputYPosition;
    m_pControllerStatus->m_handbrakeButtonPressed =
        m_state.rgbButtons[m_keys.handbrake] != 0;
    // Detect single keystroke
    if (m_lastState.rgbButtons[m_keys.train] != 0 &&
        m_state.rgbButtons[m_keys.train] == 0)
        m_trainNNetwork = !m_trainNNetwork;

    return false;
}
```

Listado 41. Definición de UmrJoystickInput

DirectInput utiliza valores enteros para almacenar el valor de giro de los ejes del dispositivo, por lo que cuando realizamos la actualización de nuestro controlador tenemos que transformar este valor para que quede dentro del rango [-1, 1].

Para poder configurar la acción asociada a cada tecla del dispositivo implementamos un mecanismo parecido al del teclado. En este caso los botones del joystick están almacenados en un array, por lo que a cada acción tenemos que asociarle el número de botón que la realiza.

6.6.7 Inteligencia artificial

La implementación del sistema de inteligencia artificial se encuentra repartida entre varias clases: *UmrAIControl*, *UmrAIControlSlow*, *UmrAIControlFast* y *UmrVehicle*. Las tres primeras, utilizando una red neuronal, se encargan de trasladar las acciones del agente inteligente al vehículo y la última contiene los sensores y la capacidad para registrar sus valores en un fichero de texto que posteriormente se utilizará en el entrenamiento de las redes neuronales.

6.6.7.1 Control del agente

La interfaz común que tienen que implentar los controladores de un agente inteligente la proporciona *UmrAIControl*:

```
using namespace Flood;

typedef boost::shared_ptr<MultilayerPerceptron>
MultilayerPerceptronPtr;

/**
    AI Control class.
    Uses two previously trained neural networks to provide
    input values in order to control the vehicle.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrAIControl : public UmrController
{
public:
    UmrAIControl(UmrVehicle* vehicle);
    virtual ~UmrAIControl(void);

    virtual bool updateState(float timeStep) = 0;
    virtual void setInputStatus(
        hkpVehicleDriverInputAnalogStatus* status);
protected:
    /// Neural network for vehicle turning
    MultilayerPerceptronPtr m_pNNetwork1;

    /// Neural network for vehicle acceleration
    MultilayerPerceptronPtr m_pNNetwork2;

    /// Controlled vehicle
    UmrVehicle* m_pVehicle;
};

typedef boost::shared_ptr<UmrAIControl> UmrAIControlPtr;
```

Listado 42. Declaración de UmrAIControl

Como se introdujo en apartados anteriores, el sistema de inteligencia artitifial utiliza dos

versiones de controladores. Uno que realiza una conducción más prudente (*UmrAIControlSlow*) y otro que conduce de manera arriesgada (*UmrAIControlFast*). Ambas implementaciones utilizan distintos sensores de entrada para la red neuronal por lo que requieren de un método de actualización diferente.

```
/**
    Slow AI Control class.
    AI Control version that uses a neural
    network trained with low speed driving data.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrAIControlSlow : public UmrAIControl
{
public:
    UmrAIControlSlow(UmrVehicle* vehicle);
    ~UmrAIControlSlow(void) {}

    virtual bool updateState(float timeStep);
};
```

Listado 43. Declaración de UmrAIControlSlow

```
/**
    Constructor.
*/
UmrAIControlSlow::UmrAIControlSlow(UmrVehicle* vehicle)
    : UmrAIControl(vehicle)
{
    m_pNNetwork1->load(
        "./gamedata/vehicles/Razor/MultilayerPerceptron11.ini");
    m_pNNetwork2->load(
        "./gamedata/vehicles/Razor/MultilayerPerceptron12.ini");
}

/**
    Updates the state of the controller.
*/
bool UmrAIControlSlow::updateState(float timeStep)
{
    Vector<double> input1(4);
    Vector<double> input2(5);
    Vector<double> output;

    input1[0] = m_pVehicle->m_rays[0];
    input1[1] = m_pVehicle->m_rays[1];
    input1[2] = m_pVehicle->m_rays[3];
    input1[3] = m_pVehicle->m_rays[4];

    input2[0] = m_inputYPosition;
    input2[1] = m_pVehicle->m_speed * 0.01f;
    input2[2] = m_pVehicle->m_rays[2];
    input2[3] = m_pVehicle->m_lastRays[2];
    input2[4] = m_pVehicle->m_tilt;

    output = m_pNNetwork1->calculateOutput(input1);
    m_inputXPosition = output[0];
```

```
    output = m_pNNetwork2->calculateOutput(input2);
    m_inputYPosition = output[0];

    m_pControllerStatus->m_positionX =
        hkMath::clamp(m_inputXPosition, -1.0f, 1.0f);
    m_pControllerStatus->m_positionY =
        hkMath::clamp(m_inputYPosition, -1.0f, 1.0f);

    return false;
}
```

Listado 44. Definición de UmrAIControlSlow

```
/**
    Fast AI Control class.
    AI Control version that uses a neural
    network trained with high speed driving data.
*/
__declspec(align(MEM_ALIGNMENT))
class UmrAIControlFast : public UmrAIControl
{
public:
    UmrAIControlFast(UmrVehicle* vehicle);
    ~UmrAIControlFast(void) {}

    virtual bool updateState(float timeStep);
};
```

Listado 45. Declaración de UmrAIControlFast

```
/**
    Constructor.
*/
UmrAIControlFast::UmrAIControlFast(UmrVehicle* vehicle)
    : UmrAIControl(vehicle)
{
    m_pNNetwork1->load(
        "./gamedata/vehicles/Razor/MultilayerPerceptron21.ini");
    m_pNNetwork2->load(
        "./gamedata/vehicles/Razor/MultilayerPerceptron22.ini");
}

/**
    Updates the state of the controller.
*/
bool UmrAIControlFast::updateState(float timeStep)
{
    Vector<double> input1(17);
    Vector<double> input2(17);
    Vector<double> output;
    int i, j;
    float speed = m_pVehicle->m_speed * 0.01f;

    // Set inputs for the first net
    input1[0] = m_inputYPosition;
    input1[1] = speed;
    for (i = 2, j = 0; j < NRAYS; i++, j++) {
        input1[i] = m_pVehicle->m_rays[j];
    }
```

```
    for (j = 0; j < NRAYS; i++, j++) {
        input1[i] = m_pVehicle->m_lastRays[j];
    }
    input1[16] = m_pVehicle->m_dirDiff;

    // Set inputs for the second net
    input2[0] = m_inputXPosition;
    input2[1] = m_inputYPosition;
    input2[2] = speed;
    for (i = 3, j = 0; j < NRAYS; i++, j++) {
        input2[i] = m_pVehicle->m_rays[j];
    }
    for (j = 0; j < NRAYS; i++, j++) {
        input2[i] = m_pVehicle->m_lastRays[j];
    }

    output = m_pNNetwork1->calculateOutput(input1);
    m_inputXPosition = output[0];
    output = m_pNNetwork2->calculateOutput(input2);
    m_inputYPosition = output[0];

    m_pControllerStatus->m_positionX =
        hkMath::clamp(m_inputXPosition, -1.0f, 1.0f);
    m_pControllerStatus->m_positionY =
        hkMath::clamp(m_inputYPosition, -1.0f, 1.0f);

    return false;
}
```

Listado 46. Definición de UmrAIControlFast

La actualización de los controladores únicamente consiste en, dada la entrada recogida de los sensores del vehículo y mediante cada red neuronal, calcular la salida de la red neuronal y modificar el valor de giro y aceleración del controlador del vehículo.

6.6.7.2 Sensores y su registro

En la implementación final del vehículo se encuentra la infraestructura de sensores de las dos últimas iteraciones del desarrollo de la red neuronal, ya que son las que funcionaban de manera satisfactoria. En la declaración de *UmrVehicle* podemos encontrar las variables de las que hacemos uso para mantener los sensores:

```
    /// Calculated current speed of the vehicle.
    float m_speed;

    /// Vehicle coordinate system.
    hkVector4 m_forward;
    hkVector4 m_right;
    hkVector4 m_up;

    /// Rays used to detect distance to the road's borders.
    /// Rays are casted in the angles: 45, 22.5, 0, -22.5, -45
    float m_rays[NRAYS];
```

```
    float m_lastRays[NRAYS];

    /// Current road tilt.
    float m_tilt;

    /// Difference between vehicle direction and actual direction.
    float m_dirDiff;

    /// Training sample to be recorded on the training file.
    float m_NNInput[NNINPUTS];
    float m_NNOutput[NNOUTPUTS];

    /// File which contains training data
    /// from the last training session.
    std::ofstream m_trainingData;
```

Listado 47. Variables para los sensores del vehículo

Unas de las variables más importantes son los tres vectores (`m_forward`, `m_right`, y `m_up`) que definen el eje de coordenadas del vehículo. A partir de ellos vamos a poder calcular el valor de los sensores que están almacenados en las variables:

- Velocidad: `m_speed`.
- Rayos: m_rays.
- Rayos previos: m_lastRays.
- Inclinación del terreno: m_tilt.
- Diferencial de dirección: m_dirDiff.

Cada vez que se actualiza el controlador del vehículo debemos actualizar el estado de los sensores, empezando por el eje de coordenadas:

```
/**
    Updates the axis vectors of the vehicle.
*/
void UmrVehicle::updateVehicleAxis()
{
    hkQuaternion rotation(m_pChassis->getRotation());
    hkVector4 forward(1, 0, 0);
    hkVector4 right(0, 0, 1);
    hkVector4 up(0, 1, 0);
    m_forward.setRotatedDir(rotation, forward);
    m_forward.normalize3();
    m_right.setRotatedDir(rotation, right);
    m_right.normalize3();
    m_up.setCross(m_right, m_forward);
    m_up.normalize3();
    m_tilt = m_forward.dot3(up);
}
```

Listado 48. Actualización del eje de coordenadas del vehículo

A partir del eje de coordenadas estándar y la rotación actual del vehículo podemos calcular el nuevo eje de coordenadas, así como el valor para el sensor de inclinación del terreno.

```
/**
    Updates speed of the vehicle.
*/
void UmrVehicle::updateSpeed()
{
    hkVector4 velocity = m_pChassis->getLinearVelocity();
    float speed = velocity.length3(); //m_pVehicle->calcKMPH();
    m_speed = speed * 2.0f;
    hkVector4 vSpeed(velocity);
    vSpeed.normalize3IfNotZero();
    m_dirDiff = m_right.dot3(vSpeed);
    m_dirDiff *= -1.0f;
    vSpeed.add3clobberW(m_forward);
    if (vSpeed.equals3(hkVector4::getZero(), 0.5f)) m_speed *= -1.0f;
}
```

Listado 49. Actualización de la velocidad del vehículo

Basándonos en la velocidad lineal del chasis calculamos la nueva velocidad del vehículo y el diferencial de dirección actual.

Para actualizar los sensores de distancia utilizamos el mecanismo de trazado de rayos de Havok. Con este mecanismo podemos lanzar un rayo desde un punto dado hasta otro y saber si el rayo colisiona con algún objeto y a la distancia relativa a la que choca. La comprobación de estas colisiones la implementamos en el método `checkHit()` de *UmrVehicle*:

```
/**
    Checks if a ray hits against an obstacle.
    @param
        from The starting point of the ray in world space.
    @param
        to The end point of the ray in world space.
*/
float UmrVehicle::checkHit(const hkVector4 &from, const hkVector4 &to)
{
    hkpWorldRayCastInput input;
    hkpClosestRayHitCollector output;
    hkpWorld* world = m_pChassis->getWorld();

    input.m_filterInfo = hkpGroupFilter::calcFilterInfo(RAY_FILTER);
    input.m_from = from;
    input.m_to = to;

    world->castRay(input, output);
    if (output.hasHit()) {
        const hkpWorldRayCastOutput hit = output.getHit();
        return hit.m_hitFraction;
    } else {
        return 1.0f;
```

```
    }
}
```

Listado 50. Comprobación de la colisión de un rayo

Con el método anterior estamos en disposición de actualizar los sensores de distancia con tan sólo proporcionar el origen y el final del rayo asociado al sensor.

```
/**
    Update rays to measure the distance to the border of the track.
*/
void UmrVehicle::updateRays()
{
    hkVector4 pos(m_pChassis->getPosition());
    float degToRad = HK_REAL_PI / 180.0f;
    // 75 m ahead of the vehicle's position
    hkVector4 to(pos(0) + 75.0f * m_forward(0),
                 pos(1) + 75.0f * m_forward(1),
                 pos(2) + 75.0f * m_forward(2));

    for (int i = 0; i < NRAYS; i++) {
        m_lastRays[i] = m_rays[i];
    }

    // front ray
    m_rays[2] = checkHit(pos, to);

    // back ray
    to = hkVector4 (pos(0) + 75.0f * -m_forward(0),
                    pos(1) + 75.0f * -m_forward(1),
                    pos(2) + 75.0f * -m_forward(2));
    m_backRay = checkHit(pos, to);

    float front;
    to = hkVector4 (pos(0) + 25.0f * m_forward(0),
                    pos(1) + 25.0f * m_forward(1),
                    pos(2) + 25.0f * m_forward(2));
    front = checkHit(pos, to);

    hkVector4 from(pos(0) + 24.5f * front * m_forward(0),
                   pos(1) + 24.5f * front * m_forward(1),
                   pos(2) + 24.5f * front * m_forward(2));

    // left turn detection ray
    to = hkVector4(from(0) + 20.0f * -m_right(0),
                   from(1) + 20.0f * -m_right(1),
                   from(2) + 20.0f * -m_right(2));
    m_rays[5] = checkHit(from, to);

    // right turn detection ray
    to = hkVector4(from(0) + 20.0f * m_right(0),
                   from(1) + 20.0f * m_right(1),
                   from(2) + 20.0f * m_right(2));
    m_rays[6] = checkHit(from, to);

    // left side ray
    {
```

```
        float angle = 45.0f * degToRad;
        hkQuaternion quat(m_up, angle);
        hkVector4 forward;
        forward.setRotatedDir(quat, m_forward);
        forward.normalize3();
        to = hkVector4(pos(0) + 75.0f * forward(0),
                       pos(1) + 75.0f * forward(1),
                       pos(2) + 75.0f * forward(2));
        m_rays[0] = checkHit(pos, to);
    }

    // front left side ray
    {
        float angle = 22.5f * degToRad;
        hkQuaternion quat(m_up, angle);
        hkVector4 forward;
        forward.setRotatedDir(quat, m_forward);
        forward.normalize3();
        to = hkVector4(pos(0) + 75.0f * forward(0),
                       pos(1) + 75.0f * forward(1),
                       pos(2) + 75.0f * forward(2));
        m_rays[1] = checkHit(pos, to);
    }

    // front right side ray
    {
        float angle = -22.5f * degToRad;
        hkQuaternion quat(m_up, angle);
        hkVector4 forward;
        forward.setRotatedDir(quat, m_forward);
        forward.normalize3();
        to = hkVector4(pos(0) + 75.0f * forward(0),
                       pos(1) + 75.0f * forward(1),
                       pos(2) + 75.0f * forward(2));
        m_rays[3] = checkHit(pos, to);
    }

    // right side ray
    {
        float angle = -45.0f * degToRad;
        hkQuaternion quat(m_up, angle);
        hkVector4 forward;
        forward.setRotatedDir(quat, m_forward);
        forward.normalize3();
        to = hkVector4(pos(0) + 75.0f * forward(0),
                       pos(1) + 75.0f * forward(1),
                       pos(2) + 75.0f * forward(2));
        m_rays[4] = checkHit(pos, to);
    }
}
```

Listado 51. Actualización de los sensores de distancia

Lo primero que hacemos es guardar el valor de las distancias de la etapa anterior en las variables dedicadas a almacenar las distancias previas. Despues calculamos el origen y el destino de cada rayo, según vimos en el diseño de dichos sensores, y comprobamos si

colisionan con algún obstáculo actualizándolo con la distancia a la que ocurre la colisión.

Tras actualizar todos los sensores, cuando el usuario activa el modo de entrenamiento, escribimos en un fichero los valores de los sensores junto con los valores del controlador para el giro del volante y la aceleración.

```
/**
    Stores the inputs for the neural network training.
*/
void UmrVehicle::prepareNNInputs()
{
    int i = 3;
    m_NNInput[0] = m_pController->getXPosition();
    m_NNInput[1] = m_pController->getYPosition();
    // Normalize speed into the range [-1, 1]
    m_NNInput[2] = m_speed * 0.01f;
    for (int j = 0; j < NRAYS; i++, j++)
        m_NNInput[i] = m_rays[j];
    for (int j = 0; j < NRAYS; i++, j++)
        m_NNInput[i] = m_lastRays[j];
    m_NNInput[17] = m_tilt;
    m_NNInput[18] = m_dirDiff;
}

/**
    Stores the outputs for the neural network training.
*/
void UmrVehicle::prepareNNOutputs()
{
    m_NNOutput[0] = m_pController->getXPosition();
    m_NNOutput[1] = m_pController->getYPosition();
}

/**
    Writes the last stored training data to a file.
*/
void UmrVehicle::writeTrainingData()
{
    for (int i = 0; i < NNINPUTS; i++)
        m_trainingData << m_NNInput[i] << " ";

    for (int i = 0; i < NNOUTPUTS; i++)
        m_trainingData << m_NNOutput[i] << " ";
    m_trainingData << endl;
}
```

Listado 52. Registro de los sensores

Una vez tenemos la infraestructura para calcular el valor de los sensores y registrarlos, podemos incorporarla al método que se encarga de actualizar el control del vehículo.

```
/**
    Updates the input to the vehicle from the player's controller.
*/
```

```
bool UmrVehicle::updateControl()
{
    bool trainNN = m_pController->getTrainNNetwork();
    bool wrongDir = false;

    // Save transform only when it's safe
    if (m_speed > 40.0f && m_rays[2] > 0.2f) {
        m_lastValidTrans = m_pChassis->getTransform();
        m_lastValidDir = m_forward;
    }

    wrongDir = m_forward.dot3(m_lastValidDir) < -0.85f;

    m_elapsedTime = m_pTimer->getMilliseconds() - m_lastTime;
    // Check every second if the car is stuck
    if (m_elapsedTime > 1000) {
        m_lastTime = m_elapsedTime;
        if (m_speed >= -0.5f && m_speed <= 0.5f &&
            (m_rays[2] < 0.05f || m_rays[0] < 0.05f ||
            m_rays[4] < 0.05f || m_backRay < 0.05f))
            m_pChassis->setTransform(m_lastValidTrans);
        else if (wrongDir)
            m_pChassis->setTransform(m_lastValidTrans);
        else if (m_up(1) < 0)
            m_pChassis->setTransform(m_lastValidTrans);
    }

    updateVehicleAxis();
    updateSpeed();
    updateRays();

    if (trainNN) {
        prepareNNInputs();
    }
    // Get input from the controller
    bool quit = m_pController->updateState(TIME_STEP_S);
    if (trainNN) {
        prepareNNOutputs();
        // write training set
        writeTrainingData();
    }

    if (m_pController->isActive()) m_pChassis->activate();
    updateStats();

    return quit;
}
```

Listado 53. Actualización del control del vehículo

6.7 Entrenamiento de la red neuronal

El entrenamiento de la red neuronal se implementa en una aplicación diferente al juego ya que no necesita ninguna información adicional aparte del fichero que contiene las entradas a la red neuronal junto con las salidas deseadas. A partir de la tercera iteración se utiliza la biblioteca

Flood para redes neuronales, momento en que el uso de la redes neuronales resultó fructuoso. Por lo tanto, en esta memoria sólo se mostrará la implementación del método de entrenamiento que funcionó. Las implementaciones de las dos versiones anteriores se pueden consultar en el CD adjunto.

```
using namespace Flood;
using namespace std;

int main (int argc, char *argv[])
{
    ofstream evaluationHistory, evaluation, gradientNormHistory;
    cout << "Neural network training for vehicle control." << endl;

    // Initialize random number generator
    srand((unsigned)time(NULL));

    InputTargetDataSet trainingDataSet;
    InputTargetDataSet validationDataSet;

    // Load training and validation data
    trainingDataSet.load("TrainingDataSet.ini");
    validationDataSet.load("ValidationDataSet.ini");

    Vector<Vector<std::string>> information =
        trainingDataSet.getAllInformation();
    Vector<Vector<double>> statistics =
        trainingDataSet.calculateAllStatistics();

    // Setup net arquitecture
    int numberOfInputs = trainingDataSet.getNumberOfInputVariables();
    int numberOfHiddenNeurons = 12;
    int numberOfOutputs =trainingDataSet.getNumberOfTargetVariables();

    MultilayerPerceptron multilayerPerceptron(
        numberOfInputs, numberOfHiddenNeurons, numberOfOutputs);
    multilayerPerceptron.setOutputLayerActivationFunction(
        Perceptron::HyperbolicTangent);

    NormalizedSquaredError trainingError(
        &multilayerPerceptron, &trainingDataSet);

    QuasiNewtonMethod quasiNewtonMethod(&trainingError);
    // Setup training parameters
    quasiNewtonMethod.setMaximumNumberOfEpochs(1000);
    quasiNewtonMethod.setDisplayPeriod(10);
    quasiNewtonMethod.setMinimumImprovement(1.0e-9);
    quasiNewtonMethod.setGradientNormGoal(0.0);
    quasiNewtonMethod.setReserveEvaluationHistory(true);
    quasiNewtonMethod.setReserveGradientNormHistory(true);

    quasiNewtonMethod.train();

    // Log training results
    evaluationHistory.open("EvaluationHistory.ini",
                           ios::out | ios::trunc);
```

```
    gradientNormHistory.open("GradientNormHistory.ini",
                             ios::out | ios::trunc);
    Vector<double> evals = quasiNewtonMethod.getEvaluationHistory();
    Vector<double> norms = quasiNewtonMethod.getGradientNormHistory();
    for (int i = 0; i < evals.getSize(); i++) {
        evaluationHistory << i << " " << evals[i] << endl;
        gradientNormHistory << i << " " << norms[i] << endl;
    }
    evaluationHistory.close();
    gradientNormHistory.close();

    // Save neural network
    multilayerPerceptron.setAllInformation(information);
    multilayerPerceptron.setAllStatistics(statistics);
    multilayerPerceptron.save("MultilayerPerceptron.ini");

    NormalizedSquaredError validationError(
        &multilayerPerceptron, &validationDataSet);

    cout << "Validation error:" << endl
        << validationError.calculateEvaluation() << endl;
    // Log training evaluation
    evaluation.open("Evaluation.ini", ios::out | ios::trunc);
    evaluation << "Training error: "
               << evals[evals.getSize()-1] << endl;
    evaluation << "Validation error: "
               << validationError.calculateEvaluation() << endl;
    evaluation.close();

    system("pause");

    return 0;
}
```

Listado 54. Entrenamiento de la red neuronal

Los pasos que seguimos para entrenar la red neuronal son los siguientes:

- Leemos dos conjuntos de datos, un conjunto de datos de entrenamiento y otro de validación.
- Configuramos la arquitectura de la red neuronal a entrenar (número de capas ocultas y función de activación de la capa de salida).
- Configuramos la instancia de la clase que realiza el entrenamiento sobre la red neuronal (épocas de entrenamiento, periodo para mostrar salida por pantalla, error objetivo, etc.).
- Tras realizar el entrenamiento guardamos en ficheros el historial con las evaluaciones y la norma del gradiente en cada época de entrenamiento.
- Para finalizar guardamos la red neuronal en un fichero y la evaluación final en otro fichero.

Capítulo 7

Entrenamiento y pruebas

En este capítulo se documenta el proceso de entrenamiento de las redes neuronales así como las pruebas realizadas posteriormente.

Para recoger los datos de entrenamiento simplemente utilizamos un vehículo manejado por nosotros mismos con el que se recogen los datos necesarios mientras se condujo alrededor del circuito (ver Figura 7.1) durante varias vueltas. Es preciso utilizar este método de alto nivel porque calcular manualmente la entrada y su salida deseada para cubrir un número de situaciones adecudo resultaría una tarea inabordable. Siguiendo este método nos aseguramos de que tenemos muestras de sobra que cubrirán la mayoría de las situaciones posibles.

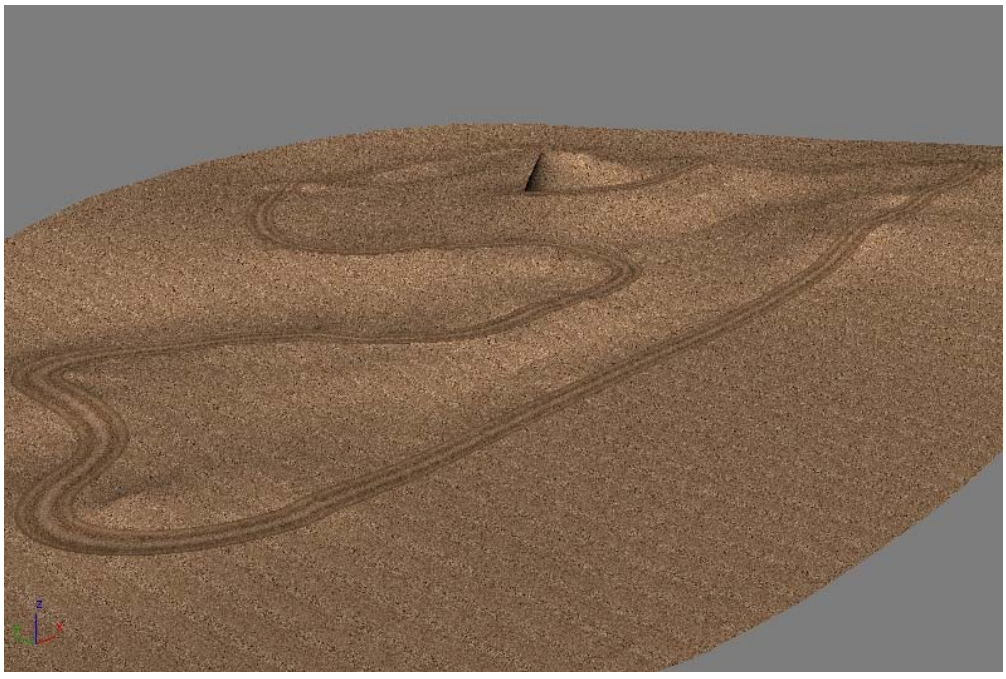

Figura 7.1. Circuito usado para el entrenamiento

A la hora de realizar la implementación y el entrenamiento hay que diferenciar claramente dos versiones debido a la gran diferencia en los resultados obtenidos.

7.1 Primera versión

La implementación de la primera versión utilizó una adaptación de la red neuronal presentada en el capítulo 21 del libro *AI Game Engine Programming* [10].

7.1.1 Primera iteración

Como vimos en el capítulo 3 la primera red neuronal quedó determinada por las siguientes variables:

- Giro previo.
- Aceleración previa.
- Velocidad.
- Deslizamiento del vehículo.
- Rayo 1 (-45º).
- Rayo 2 (-22.5º).
- Rayo 3 (0º).
- Rayo 4 (22.5º).
- Rayo 5 (45º).
- Distancia a pendiente.
- Dirección equivocada.

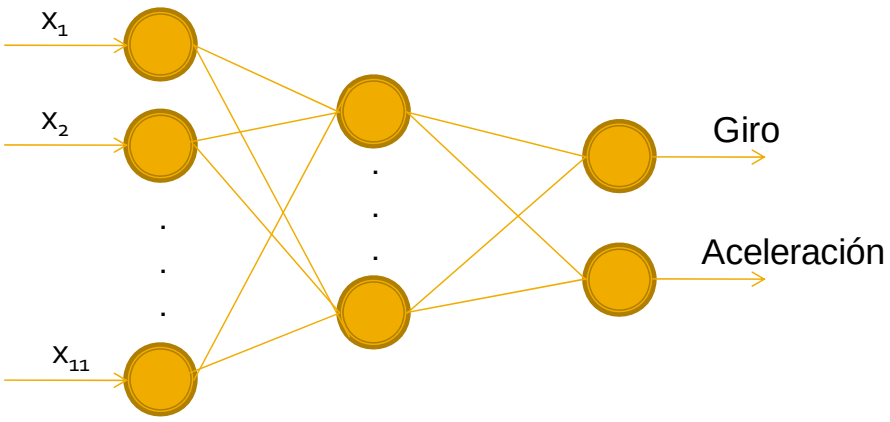

Figura 7.2. Primera red neuronal

Para el entrenamiento de la red neuronal se generó un conjunto de datos de entrenamiento compuesto por 5329 muestras formadas por los sensores de entrada junto con la salida

deseada. El entrenamiento se detuvo al cabo de 2000 épocas o cuando se alcanzó un error menor a 0.005. Se alcanzó el error máximo en la primera época de entrenamiento.

La primera implementación resultó en un vehículo incapaz de avanzar más alla de la decena de metros, ya que se dirigía constantemente hacia los bordes de la carretera quedándose bloqueado en todo momento.

7.1.2 Segunda iteración

En la segunda iteración, se utilizaron redes neuronales diferentes para el giro y la aceleración, ambas con las mismas entradas usadas en la iteración anterior, quedando la siguiente arquitectura:

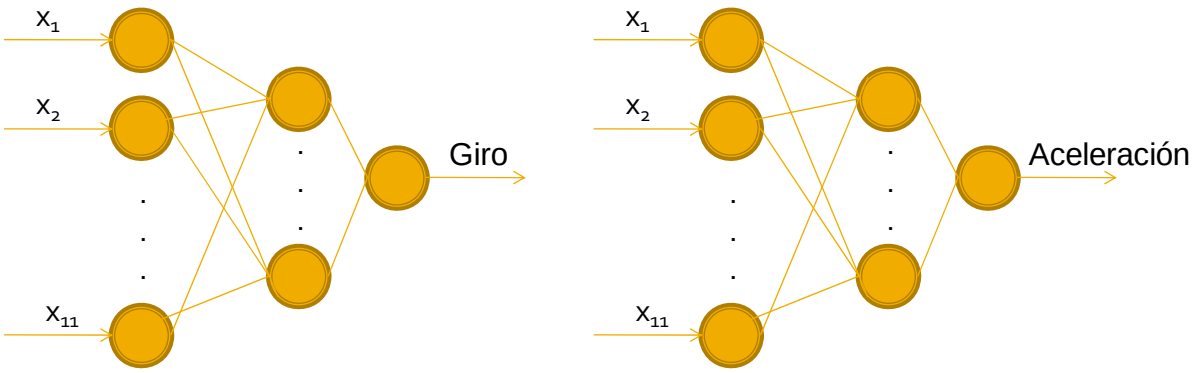

Figura 7.3. Segunda red neuronal

Para el entrenamiento de la red neuronal se generó un conjunto de datos de entrenamiento compuesto por 5329 muestras formadas por los sensores de entrada junto con la salida deseada. El entrenamiento se detuvo al cabo de 2000 épocas o cuando se alcanzó un error menor a 0.005. Se alcanzó el error máximo en la primera época de entrenamiento.

Con la segunda implementación el resultado del coche mejoró en línea recta, no así en curvas, en las cuales la acción del vehículo era seguir recto con la consiguiente colisión contra el borde del circuito. Aun así no se consigue completar una vuelta al circuito.

7.2 Segunda versión

A partir de la tercera implementación y con la ayuda de las utilidades proporcionadas por la

biblioteca para redes neuronales Flood [9], el entrenamiento de la red neuronal fue satisfactorio y proporcionaba un buen comportamiento al utilizarla con el vehículo desde el primer momento.

En el entrenamiento de la red neuronal se utilizaron dos conjuntos de datos diferentes; un conjunto de entrenamiento para determinar los parámetros de la red neuronal y un conjunto de validación, distinto del anterior, para estimar el error de generalización.

Para establecer la arquitectura óptima a utilizar por la red neuronal se realizaron varios entrenamientos con distinto número de neuronas en la capa oculta y se recogieron los resultados para medir el rendimiento de cada red neuronal.

Así mismo se realizaron varios entrenamientos para la misma arquitectura para asegurarnos de que el entrenamiento no se quedaba estancado en mínimos locales. Normalmente se llegaron a los mismos resultados, pero de esta forma evitamos posibles errores.

7.2.1 Tercera iteración

Para la tercera iteración se utilizaron las siguientes redes neuronales:

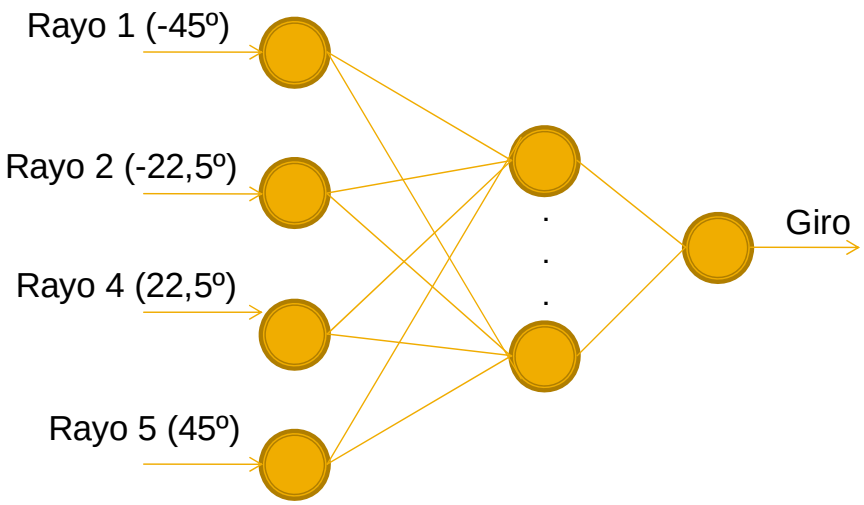

Figura 7.4. Tercera red neuronal – Giro

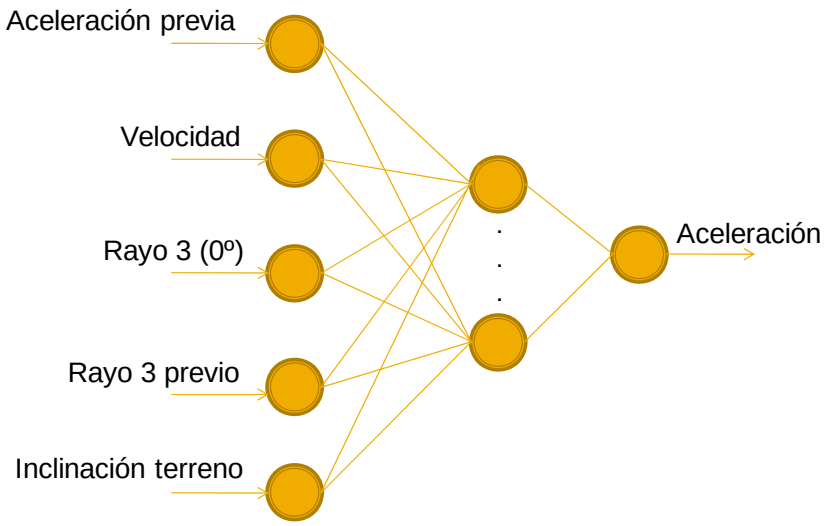

Figura 7.5. Tercera red neuronal – Aceleración

Los datos del conjunto de entrenamiento se generaron conduciendo el coche de manera prudente, manteniéndose en todo momento en el centro de la carretera, reduciendo a baja velocidad en el trazado de la curva y evitando derrapar.

Para el entrenamiento de la primera red neuronal se generó un conjunto de datos de entrenamiento compuesto por 1660 muestras formadas por los sensores de entrada junto con la salida deseada. El conjunto de datos de validación está compuesto por 836 muestras con la misma estructura y entrenamiento se lleva a cabo durante 1000 épocas con los siguientes resultados finales:

Giro de volante					
Nº neuronas	***3***	***4***	***5***	***6***	***7***
Error entrenamiento	0.0135891	0.0133099	0.018027	0.0120551	0.0072015
Error validación	0.00525743	0.00660529	0.00804562	0.00365268	0.00150064

Tabla 1. Tercera iteración - Entrenamiento red neuronal para el giro

De los resultados anteriores se puede observar que a priori una red neuronal con 7 neuronas ocultas proporcionó mejores resultados ya que fue la arquitectura con menor error de validación.

El entrenamiento de la segunda red neuronal se realizó de igual manera que la anterior, cambiando sólo el contenido de los conjuntos de datos de acuerdo al diseño de la red.

Aceleración					
Nº neuronas	***3***	***4***	***5***	***6***	***7***
Error entrenamiento	0.0558365	0.0538607	0.0713993	0.0560998	0.0579774
Error validación	0.0290183	0.027948	0.0391798	0.0311026	0.0326203

Tabla 2. Tercera iteración - Entrenamiento red neuronal para la aceleración

En el caso de la segunda red neuronal 4 neuronas ocultas nos da mejor resultado.

Con los datos obtenidos ya estamos en disposición de utilizar la red neuronal en el controlador del agente inteligente. Al utilizar esta versión de red neuronal se pudo comprobar como el coche se comportaba correctamente en el circuito. No sólo fue capaz de completar vueltas al circuito, también fue capaz de evitar obstáculos y adelantar a otros vehículos. Un punto en contra es que cuando el vehículo se encontraba dentro de una curva la velocidad resultaba un tanto baja.

7.2.2 Cuarta iteración

Por último, en la cuarta iteración se hizo uso de las siguientes redes neuronales:

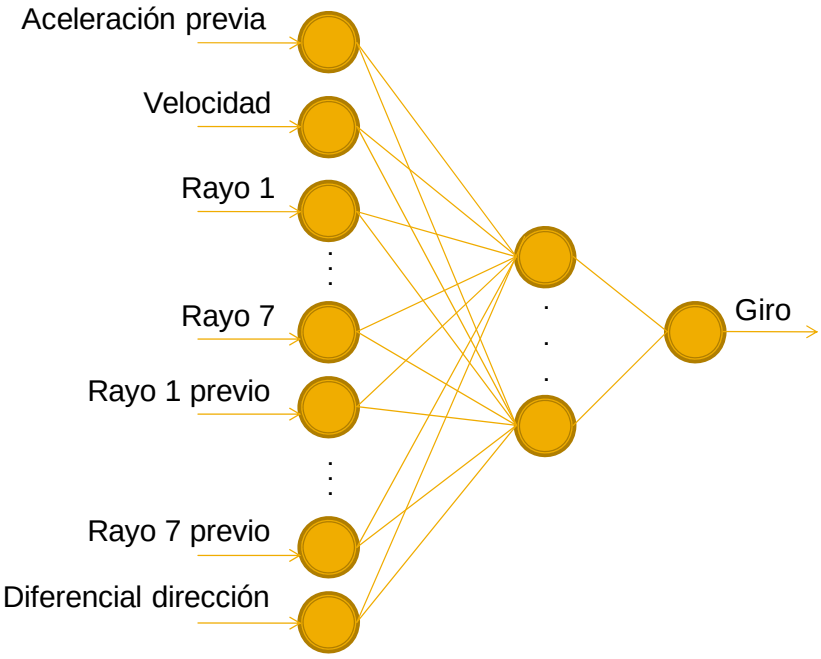

Figura 7.6. Cuarta red neuronal – Giro

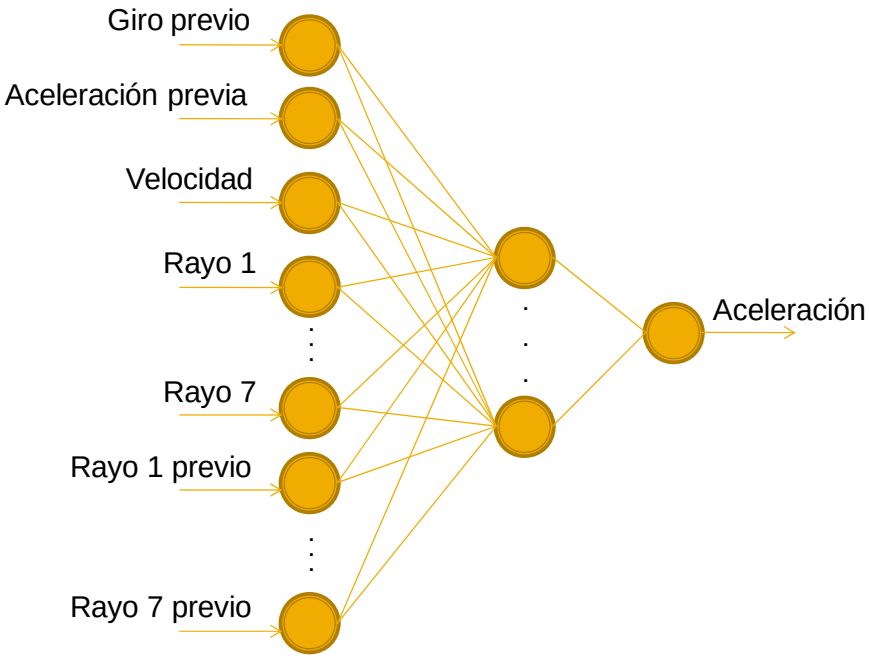

Figura 7.7. Cuarta red neuronal - Aceleración

El entrenamiento de la última implementación siguió las mismas directrices establecidas en el anterior. Lo único que cambió fue el conjunto de datos de entrenamiento a utilizar. Conjunto que se generó conduciendo el vehículo lo más rápido posible a través del circuito, derrapando

y arriesgando en curvas.

Puesto que el número de sensores que utilizamos es mayor, el número de neuronas en la capa oculta debió ser mayor para obtener buenos resultados.

Para el entrenamiento de la primera red neuronal se generó un conjunto de datos de entrenamiento compuesto por 8868 muestras formadas por los sensores de entrada junto con la salida deseada. El conjunto de datos de validación estuvo compuesto por 4293 muestras con la misma estructura y entrenamiento se llevó a cabo durante 1000 épocas con los siguientes resultados finales:

Giro de volante					
Nº neuronas	***6***	***8***	***10***	***12***	***14***
Error entrenamiento	0.078435	0.0916848	0.0819638	0.0946181	0.0786959
Error validación	0.168689	0.14796	0.196588	0.22912	0.290933

Tabla 3. Cuarta iteración - Entrenamiento red neuronal para el giro

La red neuronal con 8 neuronas en la capa oculta fue la que arrojó mejores resultados, ya que a mayor cantidad de neuronas ocultas podemos ver que comienza a ocurrir un superajuste.

El entrenamiento de la segunda red neuronal se realizó bajos las mismas condiciones que la anterior, cambiando sólo el contenido de los conjuntos de datos de acuerdo al diseño de la red.

Aceleración					
Nº neuronas	***8***	***10***	***12***	***14***	***16***
Error entrenamiento	0.0098665	0.0130266	0.0150245	0.0083187	0.00945627
Error validación	0.0098868	0.0122609	0.0123908	0.00866064	0.00918383

Tabla 4. Cuarta iteración - Entrenamiento red neuronal para la aceleración

Según los resultados de la tabla anterior, una red neuronal con 14 neuronas en la capa oculta fue la mejor elección.

Al llevar los datos de la teoría a la práctica observamos que el comportamiento del coche fue también correcto. El vehículo pudo completar el recorrido del circuito sin problemas, pero en el cómputo global resultó más lento que la versión anterior. Aunque las curvas rápidas las puede trazar de manera casi óptima, las curvas lentas, en las que hay que frenar antes o derrapar para tomarla a mayor velocidad, no es capaz de realizarlas correctamente llegándose a chocar con los bordes de la carretera. Así mismo, al igual que en la implementación anterior, el vehículo fue capaz de evitar obstáculos en la mayoría de las ocasiones y realizar adelantamientos.

Un importante aspecto a destacar es que, debido a la utilización únicamente de información de sensores para representar el entorno, los coches que se obtuvieron en las dos últimas iteraciones fueron capaces de conducir alrededor de nuevos circuitos sin la necesidad de realizar un nuevo entrenamiento de sus redes neuronales.

Capítulo 8

Conclusiones

En este último capítulo se anotan las conclusiones extraídas a lo largo del desarrollo del proyecto. Haremos un análisis del cumplimiento de los objetivos originales que nos propusimos al iniciar el proyecto y de las posibles mejoras de la aplicación donde pueden ir dirigidos futuros desarrollos.

8.1 Objetivos originales

Tras la evaluación de los objetivos originales del proyecto se puede concluir que estos han sido cumplidos satisfactoriamente. A continuación analizamos cada apartado individualmente.

8.1.1 Ingeniería del software

A lo largo del proyecto se ha utilizado una metodología de desarrollo iterativa e incremental que nos ha permitido en cada iteración construir un sistema mayor sobre una base sólida. El desarrollo del sistema completo se ha acometido por áreas de funcionalidad, de manera que se ha intentado desacoplar en la medida de lo posible los distintos componentes.

El lenguaje de modelado UML nos ha proporcionado una inestimable ayuda a la hora de obtener un conocimiento más amplio del sistema y sus necesidades, lo que ha facilitado enormemente el hecho de mantener en mente el objetivo que se perseguía.

Otra de los aspectos más importantes para desarrollar el sistema exitosamente es la utilización de bibliotecas ya existentes, las cuales han hecho posible acortar el tiempo de desarrollo del proyecto.

8.1.2 Gráficos

El objetivo de lograr un aspecto visual atractivo se ha conseguido gracias al uso del motor gráfico Ogre3D. Se cumple el cometido básico de trasladar al usuario todo lo que ocurre en la

simulación del juego de una manera eficaz y sin demasiada carga visual.

8.1.3 Física

Lograr un comportamiento del coche físicamente real era uno de los principales objetivos, el cual hemos podido cumplir al usar el motor físico Havok y todo el conjunto de herramientas que proporciona. Se ha conseguido un buen compromiso entre el realismo de la simulación física y la facilidad de manejo del vehículo, un aspecto importante a la hora de lograr la aceptación por parte del usuario.

8.1.4 Inteligencia artificial

El objetivo al desarrollar la inteligencia artificial era conseguir un agente inteligente capaz de aprender del comportamiento de un jugador humano y que en última instancia pudiera llegar a ser competitivo. Utilizando un mecanismo de aprendizaje automático como las redes neuronales pudimos obtener un vehículo que aprende de un jugador humano y no necesita información previa del circuito. No llega al nivel de competitividad del usuario medio, pero el vehículo que utiliza el agente inteligente es capaz de conseguir tiempos aceptables y no cometer muchos errores de conducción.

8.2 Futuros desarrollos

8.2.1 Inteligencia artificial

Incluyendo información previa del circuito, tal como línea óptima a seguir por la carretera o información sobre la velocidad a seguir en cada tramo se podría mejorar notablemente la competitividad del vehículo. En este caso el objetivo sería aprender las variaciones en la trazada que realiza un jugador.

Un futuro desarrollo de la inteligencia artificial del juego podría incluir métodos de aprendizaje evolutivo, de manera que el comportamiento del vehículo mejore tras el entrenamiento inicial.

8.2.2 Sonido

La inclusión de un sistema de sonido mejoraría el aspecto final de la aplicación y ayudaría a

conseguir una mayor inmersión en el juego. Se pueden incorporar, además de música de fondos, efectos básicos de sonido producidos al acelerar el coche, al derrapar las ruedas sobre el asfalto y al deslizarse sobre un terreno.

8.2.3 Efectos gráficos

Otro aspecto que influiría en el atractivo del juego es la inclusión de efectos gráficos de última generación. La incorporación de efectos especiales usando shaders mediante el motor gráfico Ogre3D es un futuro desarrollo factible y mejoraría el producto final de cara al usuario.

Apéndice A.

Contenido del CD-ROM

En este apéndice se explica la distribución de carpetas y ficheros que se sigue en el CD-ROM adjunto a esta memoria.

- Aplicación. Almacena los recursos relacionados con el desarrollo del proyecto.
  - Análisis-Diseño. Contiene los diagramas realizados durante la fase de análisis y diseño de la aplicación.
  - Implementación. En esta carpeta se encuentra la implementación de las distintas implementaciones realizadas.
  - Ejecutable. Carpeta con el ejecutable de la última versión del juego.
- Contenido. Almacena los recursos gráficos asociados al proyecto.
  - 3ds. Modelos 3D en formato de Autodesk 3ds Max.
  - GMT. Modelos 3D en formato de rFactor.
- Memoria. Contiene esta memoria en formato Word y PDF.
- RedesNeuronales. Contiene los resultados obtenidos en el entrenamiento de las redes neuronales.
  - Iteración 1. Contiene los datos de entrenamiento y la red neuronal utilizada en la primera iteración almacenada en un fichero.
  - Iteración 2. Contiene los datos de entrenamiento y la red neuronal utilizada en la segunda iteración almacenada en un fichero.
  - Iteración 3.
    - Red aceleración. Contiene los datos de entrenamiento, además de los resultados y las redes neuronales organizados en carpetas por número de neuronas de la capa oculta.
    - Red giro. Contiene los datos de entrenamiento, además de los resultados y las redes neuronales organizados en carpetas por número de neuronas de la capa oculta.

- Iteración 4.
  - Red aceleración. Contiene los datos de entrenamiento, además de los resultados y las redes neuronales organizados en carpetas por número de neuronas de la capa oculta.
  - Red giro. Contiene los datos de entrenamiento, además de los resultados y las redes neuronales organizados en carpetas por número de neuronas de la capa oculta.

Bibliografía

[1] (2009) rFactor. [Online]. http://www.rfactor.net

[2] (2009) Autodesk 3ds Max. [Online]. http://www.autodesk.es/adsk/servlet/index?siteID=455755&id=12341473

[3] (2009) rFactor: Developer's Corner. [Online]. http://www.rfactor.net/index.php?page=devcorner

[4] (2009) OgreMax. [Online]. http://www.ogremax.com/

[5] (2009) Boost. [Online]. http://www.boost.org/

[6] (2009) Boost Smart Pointers. [Online]. http://www.boost.org/doc/libs/1_42_0/libs/smart_ptr/smart_ptr.htm

[7] (2009) NedMalloc Allocator. [Online]. http://www.nedprod.com/programs/portable/nedmalloc/

[8] (2009) DirectInput. [Online]. http://msdn.microsoft.com/en-us/library/ee416842(VS.85).aspx

[9] (2008) Flood. [Online]. http://www.cimne.com/flood/

[10] B. Schwab, *AI Game Engine Programming*., 2004.

[11] J.P. Flynt and O. Salem, *Software Engineering for Games Developers*., 2005.

[12] G. Junker, *Pro OGRE 3D Programming*., 2006.

[13] M. Nuñez and F. Villatoro, *Simulación de procesos físicos usando Java3D*.: E.T.S.I. Informática, Universidad de Málaga, 2008.

[14] S. Rabin, *AI Game Programming Wisdom*., 2002.

[15] M. Buckland, *AI Techniques for Game Programming*., 2002.

[16] G. Dreyfus, *Neural Networks: Methodology and Applications*., 2004.

[17] J. E. Laird and M. van Lent, *Machine Learning for Computer Games*., 2005.

[18] J. Muñoz Perez, *Modelos computacionales*.: E.T.S.I. Informática, Universidad de Málaga.

[19] Telekinesys Research Ltd., *Havok Physics and Animation User Guide*., 2008.

[20] AI-Junkie: Entrevista a Jeff Hannan. [Online]. http://www.ai-junkie.com/misc/hannan/hannan.html

[21] (2007) Open Dynamics Engine. [Online]. http://www.ode.org/

[22] (2009) NVIDIA PhysX. [Online]. http://developer.nvidia.com/object/physx.html

[23] (2009) Bullet Physics Library. [Online]. http://bulletphysics.org/wordpress/

[24] (2009) Havok. [Online]. http://www.havok.com/

[25] (2009) Ogre3D. [Online]. http://www.ogre3d.org/