Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Proyecto Fin de Carrera
Ingeniería en Informática

Videojuego de coches en red para la evaluación de un sistema P2P de compartición de información

Víctor J. Rújula Nasarre de Letosa

Directores:
Sergio Ilarri Artigas
Eduardo Mena Nieto

Área de Lenguajes y Sistemas Informáticos
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Agosto 2013

Videojuego de coches en red para la evaluación de un sistema P2P de compartición de información

RESUMEN

El objetivo de este proyecto es desarrollar un videojuego cuya utilización permita evaluar estrategias de gestión de información en redes vehiculares, una tarea actualmente realizada mediante simuladores, pero que resulta muy costosa debido a la dificultad de ajustar correctamente los parámetros usados por las diferentes estrategias.

Con este fin, se ha desarrollado un videojuego de coches para múltiples jugadores en red, en el que los jugadores tienen que completar diferentes objetivos en misiones de carácter competitivo mientras circulan por escenarios creados con datos reales obtenidos mediante el sistema de mapas de carretera OpenStreetMap.

Para facilitar la explotación del juego como método de evaluación, se han añadido diversos elementos como vehículos no humanos del tráfico, vehículos de servicios de emergencia y plazas de aparcamiento que son ocupadas dinámicamente por el tráfico y los jugadores.

Ha sido necesario el estudio del sistema de gestión de información VESPA para su posterior implementación en el juego, así como también el estudio y la aplicación de diferentes arquitecturas y técnicas de optimización de red y de técnicas de control de los vehículos no humanos.

El juego ha sido implementado siguiendo una arquitectura de red de tipo cliente-servidor con predicción en el cliente, usando técnicas como la interpolación, la extrapolación y la compresión delta, haciendo uso de los protocolos TCP y UDP. También se han definido los interfaces necesarios para poder integrar en el juego cualquier estrategia de gestión de información, a partir de las cuales se ha desarrollado una implementación del sistema VESPA.

Para facilitar la recogida de datos también se ha desarrollado un servidor dedicado, para tener un lugar centralizado desde el cual recopilar dichos datos, y otro proceso con la función de servidor de recogida de estadísticas, el cual recibe la información recopilada por los diferentes servidores durante las partidas.

Los resultados obtenidos han demostrado que a pesar de que el videojuego puede suponer una buena herramienta para recopilar mucha información para una gran variedad de escenarios, los resultados obtenidos deben ser tomados con precaución, ya que la pericia del jugador o los elementos introducidos para aumentar la diversión del juego pueden alterar los resultados obtenidos.

Por este motivo el videojuego no debe verse como un sustitutivo de los métodos tradicionales de evaluación, como los simuladores, sino como un complemento, ya que puede ayudar a recopilar con menos esfuerzo los datos que serán utilizados para afinar el protocolo y también puede servir para obtener conclusiones iniciales previas a la evaluación en el simulador.

Agradecimientos

Me gustaría agradecer este Proyecto Fin de Carrera a todas las personas que lo han hecho posible con su apoyo y dedicación.

En primer lugar a mis directores de proyecto Sergio Ilarri y Eduardo Mena, por su paciencia y su inestimable ayuda, sin la cual este proyecto no hubiera sido posible. A mis compañeros y amigos de clase, con los que he compartido estos años de carrera, por hacer que esos momentos de estudio y de prácticas fuesen agradables y amenos. A mi familia y amigos más cercanos, por su paciencia y por motivarme para seguir adelante en los momentos más complicados. A mis amigos Dani y Jorge por su inestimable colaboración cuando fue necesario probar el funcionamiento del juego con varios jugadores. A mi ex-compañera de piso Megan, por ayudarme en todas las dudas que me surgieron al traducir los textos y el manual de uso al inglés, así como al resto de compañeros de piso, por haber sido como una segunda familia para mí.

Y por supuesto, a la Universidad de Zaragoza y a todos aquellos profesores de los que he aprendido tanto a lo largo de estos años.

También debo agradecer el uso que realizo en este proyecto de las librerías *JLayer*, *Xerces*, *Guava* y *OpenSteer*, y los algoritmos obtenidos del libro *Developing Games In Java*, así como también a Josh Woodward y Howarang Van K. por el uso de su música.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

En este capítulo se mostrará la motivación existente para la realización de este Proyecto Fin de Carrera, los objetivos que han sido marcados para el proyecto, las librerías y herramientas utilizadas para su elaboración y también se analizará el trabajo relacionado. Finalmente se mostrará la estructura seguida en este documento.

1.1. Motivación del proyecto

Han sido varias las razones que me llevaron a elegir desarrollar este Proyecto Fin de Carrera. La primera y principal ha sido el interés personal en el ámbito del desarrollo de videojuegos, que siempre me ha apasionado. Por otro lado, realizar un proyecto complejo como es un videojuego, partiendo desde cero y sin tener ningún conocimiento particular de este ámbito, suponía un gran reto que deseaba afrontar porque me permitiría ampliar mis conocimientos en campos diversos como inteligencia artificial, arquitecturas y optimizaciones de red, etc., de las que poseía unos conocimientos limitados. Además, consideré que la experiencia que me podía aportar este proyecto aumentaría mis posibilidades de desarrollar mi carrera profesional en este ámbito.

1.2. Objetivos

El Proyecto Fin de Carrera que se describe en este documento tiene los siguientes objetivos:

- Desarrollar un videojuego de coches, que cuente con coches controlados por el ordenador y otros controlados por jugadores humanos conectados a través de la red.

- Desarrollar lo necesario para que se compita en escenarios creados a partir de datos reales obtenidos de algún sistema que proporcione mapas de carreteras.
- Desarrollar una funcionalidad de descarga de mapas, de forma que introduciendo la localización en la que deseas jugar se descargue una porción de mapa alrededor del punto elegido.
- Integración de los mecanismos de funcionamiento básicos de VESPA, de forma que el videojuego desarrollado permita la evaluación de las ventajas de contar con este sistema frente a un competidor que no lo tenga. La integración se realizará de forma que resulte sencillo modificar el funcionamiento de VESPA para evaluar el impacto de los cambios.

Además de estos objetivos marcados por la propuesta del Proyecto Fin de Carrera, también se ha tenido como objetivo lograr que el resultado sea un juego divertido, con un nivel de dificultad moderado, de forma que no suponga un problema para los jugadores más inexpertos pero que a la vez pueda llegar a suponer un reto para los jugadores experimentados, y que pueda lograr despertar el interés por seguir jugando de los usuarios que lo prueben.

1.3. Trabajo previo y herramientas

En esta sección se listan las librerías externas y herramientas utilizadas para el desarrollo del proyecto, acompañadas de una breve descripción del porqué de su uso.

Código externo

Para la implementación de la inteligencia artificial del manejo de vehículos se han adaptado varios algoritmos contenidos en la librería *OpenSteer (Steering Behaviors for Autonomous Characters)*[1], en la cual se hayan implementaciones de varios de los métodos sugeridos en [17].
Concretamente los métodos de esta librería adaptados son los siguientes: *Obstacle avoidance*, *Path following*, *Unaligned collision avoidance* y *Wander*.

Librerías usadas

Para el desarrollo del proyecto se ha hecho uso de diversas librerías externas que han permitido la implementación en un tiempo razonable de ciertas funciones necesarias que no formaban parte de los objetivos del proyecto:

[1] `http://opensteer.sourceforge.net/`

- **Apache Xerces2 java**[2], para el análisis y extracción de datos de los documentos XML obtenidos del servicio OpenStreetMap.
- **JLayer**[3], para poder decodificar y reproducir sonido en MP3. Necesario para que la música de fondo pueda tener un tamaño reducido.
- **Guava-12.0**[4], conjunto de librerías de *Google* de la que hago uso de su clase *ImmutableMap*.

Herramientas de desarrollo

Durante el desarrollo de este Proyecto Fin de Carrera se han utilizado las siguientes herramientas:

- **NetBeans IDE 6.8**, como editor de código y ha sido especialmente beneficiosa la ayuda que proporciona para elaborar interfaces gráficas.
- **Subversion**, como herramienta de control de versiones.
- **ClockingIT**, como herramienta de gestión de tareas del proyecto.
- **GIMP 2**, editor gráfico usado para la elaboración de elementos gráficos de los menús y para la creación y/o edición de los sprites de los elementos del juego.
- **Audacity**, editor de audio usado para la edición de los efectos de sonido del juego.
- **Oracle VM VirtualBox**, herramienta de virtualización usada para virtualizar el sistema operativo *Mac OS X* y poder realizar pruebas para asegurar la compatibilidad del juego.
- **CamStudio 2.7 y Adobe Premiere Pro CS4**, herramientas de captura y edición de video usadas para la creación del video de la página web realizada para el proyecto.
- **Java VisualVM**, herramienta en la que se muestra información detallada de las aplicaciones java en funcionamiento, usada para analizar las mejoras de rendimiento que se introducían y analizar cuáles eran las partes del código que más afectaban al rendimiento.

[2] `http://xerces.apache.org/#xerces2-j`
[3] `http://www.javazoom.net/javalayer/javalayer.html`
[4] `https://code.google.com/p/guava-libraries/`

- **Netem**, herramienta que proporciona funcionalidades de emulación de redes para probar protocolos y emular las propiedades de WANs[5]. Usada para simular un entorno de internet o WAN para realizar pruebas mientras se implementaban las técnicas de optimización de red.

Herramientas de documentación

Además de las anteriormente citadas, también se han usado las siguientes herramientas para la elaboración de la documentación del proyecto:

- **LaTeX**, lenguaje usado para la elaboración de este documento.
- **Visual Paradigm for UML 6.4, Microsoft Visio 2010 y Dia**, editores de diagramas.

1.4. Trabajo relacionado

La idea de beneficiarse de acciones humanas para mejorar o evaluar sistemas se ha aplicado previamente en otros videojuegos, como por ejemplo:

- *ESP game* [23], en el cual los jugadores ayudan implícitamente a etiquetar imágenes mientras juegan.
- *CodeSpells* [8], un videojuego de fantasía en el que los jugadores deben escribir hechizos sirviéndose del lenguaje Java
- *Planet PI4* [14], un juego multijugador online que pretende servir para probar arquitecturas P2P para juegos.

Sin embargo aparentemente este es el primer juego que será diseñado para ayudar a evaluar estrategias de gestión de información para redes vehiculares.

Un juego similar en concepción en cuanto que también permite circular por escenarios reales es *Mini Maps*[6], pero en dicho juego se pinta la vista de satélite de *google maps* y se superpone el vehículo, en lugar de crear un escenario a partir de la información del mapa.
Otro juego, *Push-Cars 2: On Europe Streets*[7] también usa escenarios reales, pero la diferencia es que no se puede jugar en cualquier localización deseada ya que los escenarios están prefijados ya que, aunque a partir de datos reales, están prediseñados de antemano.

[5] `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`
[6] `https://apps.facebook.com/minimaps/`
[7] `http://www.push-cars.com`

1.5. Estructura de la memoria

El contenido de la memoria está distribuido de la siguiente forma:

- En el capítulo 2 se expone el trabajo desarrollado para la elaboración del juego, excluyendo lo realizado para el sitio web o la explotación.
- En el capítulo 3 se analiza la posible explotación del juego como método para probar diferentes técnicas de *data sharing* y se expone el trabajo realizado para permitirlo. También se muestra el rendimiento obtenido del videojuego.
- En el capítulo 4 se muestran las conclusiones del proyecto, el posible trabajo futuro de cara a mejorar el juego o la explotación y una breve valoración personal.

Respecto al contenido de los anexos:

- En el anexo A se muestra todo lo referente al análisis realizado.
- En el anexo B se muestra todo lo relacionado con la fase de diseño.
- El anexo C contiene aspectos sobre el desarrollo del videojuego que no han podido ser tratados en el capítulo 2 o han sido tratados de forma resumida.
- En el anexo D se detalla el funcionamiento y la implementación del sistema VESPA y todos los aspectos desarrollados para el uso del videojuego como método de evaluación. También se amplía la información sobre el rendimiento del juego.
- El anexo E contiene el artículo presentado para el *workshop IMMoA'13*, realizado conjuntamente con Sergio Ilarri y Eduardo Mena (en inglés).
- En el anexo F se proporciona el manual de usuario.

Capítulo 2

Videojuego desarrollado

En este capítulo se explican las funcionalidades básicas del sistema desarrollado. Debido a la limitada extensión de la memoria, este capítulo estará centrado únicamente en los aspectos más importantes. Para profundizar más sobre estos aspectos acudir a los anexos.

2.1. Resumen del juego

`Vanet-X` es un juego de coches que puede ser jugado por hasta 8 personas a través de Internet y que permite seleccionar escenarios de localizaciones reales obtenidos a través del servicio de mapas OpenStreetMap.

El juego está basado en el clásico videojuego Rally-X (Namco 1980) (ver figura 2.1), en el que el vehículo del jugador debía recoger banderas en un tiempo limitado mientras huía de los vehículos enemigos que le perseguían insaciablemente, contra los cuales podía usar nubes de humo de motor con el objetivo de desorientarlos.

`Vanet-X` cuenta con cuatro modos de juego diferenciados pero que comparten ciertas características en común.
En todos se sigue un sistema de rondas continuo, inspirado en el existente en los modos de juego *zombis* y *supervivencia* de los últimos videojuegos de la saga *Call of Duty*.
Este sistema consiste en que cuando se completa satisfactoriamente una ronda u oleada, se avanza a la siguiente sin salir del mundo de juego ni con la aparición de menús intermedios, simplemente informando al jugador que la ronda ha sido completada y que tiene unos segundos de descanso antes de que de comienzo la siguiente, pero durante este intervalo de tiempo se puede seguir moviendo libremente por el mundo de juego. De esta forma, se consigue una mayor inmersión evitando pausas innecesarias.

Figura 2.1: Juego Rally-X (Namco 1980) en el cual se ha basado este videojuego

Modos de juego

- **Capturar la bandera**: el más similar al clásico Rally-X. Debes capturar las banderas en un tiempo limitado mientras coches enemigos te persiguen.
- **Capturar los coches enemigos**: como el anterior no hay banderas y en lugar de huir de los enemigos debes capturarlos.
- **Tareas**: mientras los coches enemigos te persiguen, y en un tiempo limitado, debes lograr completar una o varias tareas, que consisten en llegar a pie, en coche o lograr un aparcamiento cercano a un punto de interés (por ejemplo una tienda o un museo) o una dirección.
- **Supervivencia**: como el anterior pero al completar los objetivos en lugar de puntos logras dinero, que gastarás en repostar combustible o en reparaciones del vehículo. A diferencia de los anteriores modos, al completar la ronda los enemigos no desaparecen, y además tienden a ser más numerosos contra más rondas se avance, y también a diferencia del resto de modos de juego, no completar satisfactoriamente una ronda no supone perder la partida sino que únicamente se deja de ganar el dinero que el objetivo proporcionaba.

2.2. Arquitectura del sistema

La aplicación `Vanet-X` integra de forma transparente para el usuario un módulo cliente y un módulo servidor. Los motivos de la elección de una arquitectura cliente-servidor se explican en el capítulo 2.8.

El diseño que se ha realizado no se ha basado en ningún modelo específico existente, sino que se ha realizado un diseño específico para el videojuego desarrollado (ver figuras 2.2 y 2.3). Para más detalles consultar anexo B.5.

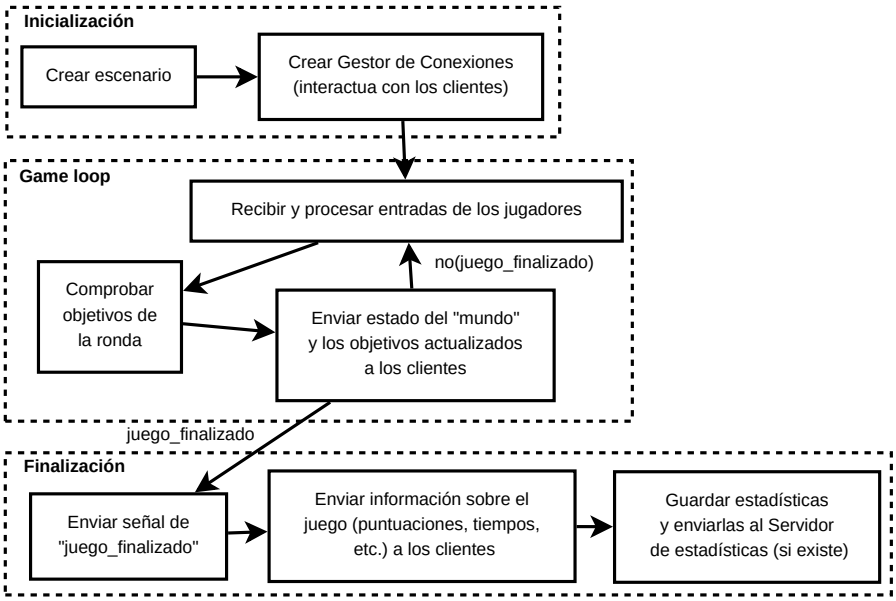


Figura 2.2: Funcionamiento del servidor

Haciéndose una aproximación muy simplificada se podría considerar que el cliente es el encargado de recoger los eventos de teclado del jugador, enviárselos al servidor para que los procese y, con los resultados que le devuelve, pintar la vista del cliente del mundo de juego.
Como se explica en el capítulo 2.8.2, esto no es del todo cierto, ya que al introducir la técnica de la predicción, el cliente también incluye parte de la lógica del juego, aunque los resultados de la lógica del servidor siguen teniendo prioridad sobre la del cliente.

Para la implementación del servidor se ha seguido un esquema multihilo (ver anexo B.6), en el que existe un hilo principal con gran parte de la lógica del juego, pero además cada actor[1] tiene su propio hilo de ejecución donde ejecuta su lógica asociada.
Además, también existe un hilo donde se ejecuta el gestor de mensajes TCP (ver capítulo 2.8), y la implementación de VESPA también crea sus propios hilos (cuatro en la implementación desarrollada).

Por su parte el cliente también es multihilo, a pesar de que en este caso los actores no tienen sus propios hilos ya que su lógica se ejecuta en el servidor (en

[1] un actor es una entidad con un comportamiento autónomo

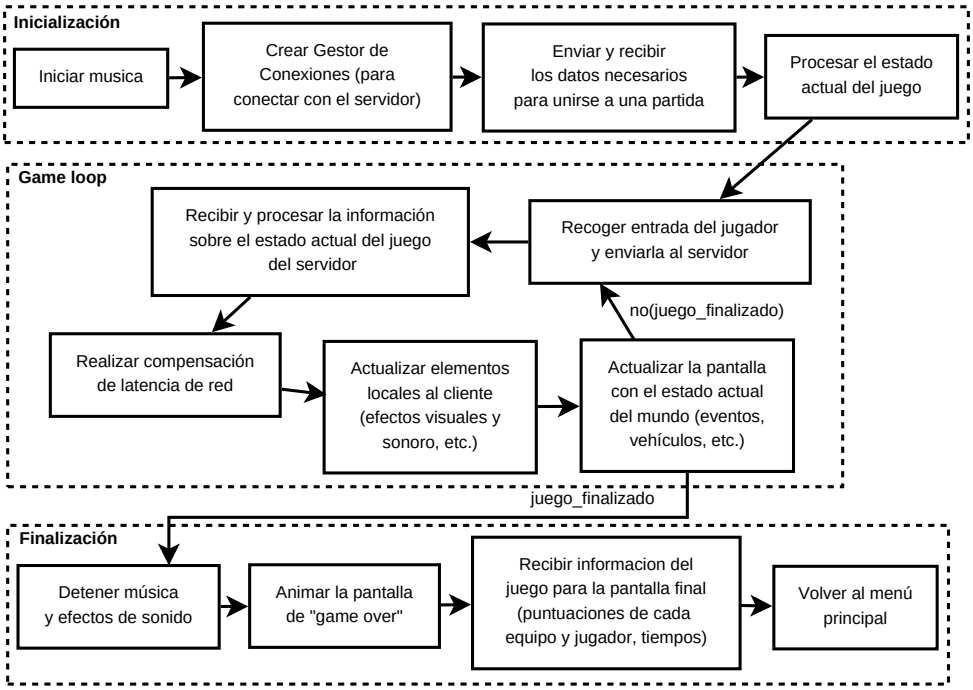


Figura 2.3: Funcionamiento del cliente

el cliente se ejecuta su predicción, pero ésta debe estar sincronizada y por ello se ejecuta en el hilo principal). El gestor de sonido tiene sus propios hilos (uno por cada sonido simultáneo permitido) y al igual que en el servidor, también existe un gestor de mensajes TCP en un hilo propio.

2.3. Menús del juego

Como en la práctica totalidad de los juegos, `Vanet-X` cuenta con un sistema de menús que permiten configurar múltiples aspectos del juego y la partida antes de comenzarla.

A diferencia de otros tipos de aplicaciones, en un videojuego es muy importante que los menús sean visualmente agradables e intuitivos, de forma que para configurar los aspectos básicos del juego no sea imprescindible leer el manual de instrucciones. Por ello en `Vanet-X` se han seguido las siguientes directrices a la hora de elaborar el menú:

- Usar diversas pantallas con botones en lugar de barras de menús
- Dotar de dinamismo al menú. Esto se ha conseguido mediante un fondo en movimiento en lugar de estático.

- Mantener similitudes en la estructura de las diferentes pantallas. En este caso en la mayoría de las pantallas existen los botones «Guardar cambios y volver», «Deshacer cambios y volver» y «Establecer valores por defecto» situadas en la misma posición, así como también los botones para avanzar o retroceder de pantalla en las que no tienen parámetros configurables.

El esquema de las pantallas se puede observar en la figura 2.4

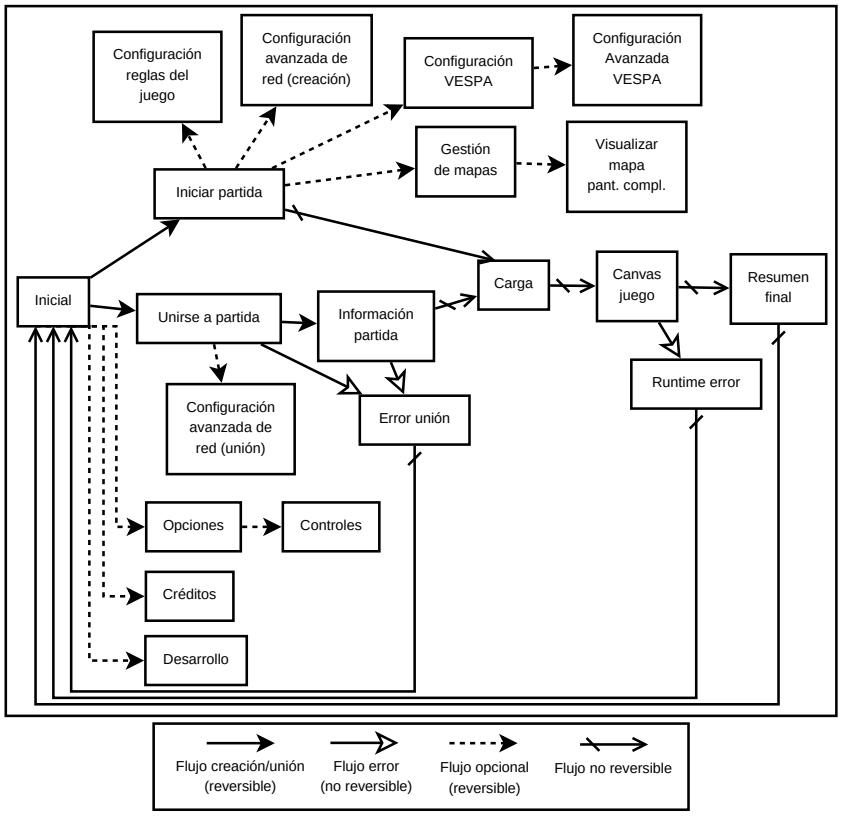


Figura 2.4: Diagrama de navegación

Existen tres «caminos» básicos en el menú:

1. Crear una partida nueva, donde se puede seleccionar modo de juego, escenario y otros parámetros y cuenta con cuatro subpantallas donde se puede configurar los puertos de la comunicación de red, los parámetros de VESPA, las reglas de la partida y gestionar los mapas de juego.

2. Unirse a una partida ya existente, donde debes introducir la dirección IP del anfitrión y te muestra datos sobre la partida en curso como por ejemplo qué jugadores están conectados.

3. Las opciones globales, donde se controla el volumen, se pueden modificar los controles y cambiar la carpeta en la que se guardan los archivos de configuración del juego.

También se han creado dos tipos de pantallas de error, que aparecen antes de la pantalla con el resumen de la partida. La primera se usa en errores durante la partida, y en ella se imprimen las diferentes trazas de error diferenciadas por pestañas según el elemento que las haya causado. La otra pantalla de error está dedicada para errores al intentar conectar a una partida, y su texto se modifica dinámicamente dependiendo del tipo de problema.

Las modificaciones de los parámetros realizadas en los menús y los nuevos escenarios descargados perduran entre diferentes ejecuciones de la aplicación gracias a que se almacenan en un directorio elegido previamente por el usuario y se cargan al inicio de la aplicación.

La pantalla de gestión de mapas permite previsualizar cualquier área, descargada o no, tarea que se realiza obteniendo de una *API* la imagen en la que se representa el área seleccionada. Para no tener que pedir una nueva imagen cada vez que se varíe el tamaño seleccionado, se descarga con el tamaño máximo permitido en el juego y, mediante una función de recorte, se muestra únicamente el área proporcional al tamaño elegido.

Para más detalles sobre estos aspectos consultar el anexo C.1.

2.4. Obtención de mapas

Uno de los objetivos principales del proyecto consiste en la posibilidad de competir en escenarios reales. Esto se ha conseguido mediante el uso de los mapas proporcionados por el servicio OpenStreetMap[2] y del servicio de búsqueda OpenStreetMap Nominatim Tool[3], que permite buscar coordenadas a partir de nombres o direcciones.

La obtención y gestión de estos mapas se realiza desde el menú de la aplicación, obteniendo los datos mediante una de las APIs de OpenStreetMap.
Para garantizar la obtención de los datos, en lugar de hacer uso de una única API, se define una lista de APIs, modificable por el usuario, que serán usadas en orden secuencial hasta que una de ellas esté operativa.

[2] `http://www.openstreetmap.org`
[3] `http://nominatim.openstreetmap.org`

El proceso de la obtención de un mapa se ha implementado de la siguiente forma:

1. Se hace una petición al servicio *OpenStreetMap Nominatim Tool* con las palabras clave de la dirección deseada, y éste devuelve un listado de los lugares coincidentes. Cada lugar incluye, entre otros datos, el nombre y las coordenadas.

2. Una vez seleccionado el lugar deseado, se hace una petición al API de OpenStreetMap, pidiendo el área creada mediante las coordenadas devueltas el listado y el valor del radio deseado. El API devolverá un documento con formato *OSM XML*[4] que contendrá los datos requeridos, y que será el que se almacene en el directorio habilitado a tal efecto.

Los mapas descargados se muestran en una tabla en la que además del alias asignado se incluye la dirección en torno a la cual esta creado, el número de nodos[5] que incluye y el tamaño del área en km^2.
Estos mapas y sus previsualizaciones son almacenados para que no sea necesario volverlos a descargar cada vez que se inicie el programa. De esta forma, es posible jugar a Vanet-X aun sin tener conexión a internet, solo siendo necesario tener los mapas descargados en la carpeta correspondiente.
Las previsualizaciones de los mapas que se han buscado pero no descargado también son almacenados pero éstos solo durante la ejecución del programa.

Para más detalles sobre estos aspectos consultar el anexo C.2.

2.5. Elementos del terreno

Como se ha explicado anteriormente (Capítulo 2.4), los datos necesarios para crear los escenarios son obtenidos del servicio de mapas OpenStreetMap.

Existen tres tipos de elementos: nodos, caminos y multipolígonos. Los caminos están formado por nodos y los multipolígonos están formados por caminos.
De estos elementos, los nodos son los únicos que no tienen representación visual, usándose solo para formar el resto de elementos. Los caminos y multipolígonos se organizan en diferentes capas de profundidad de pintado según su etiqueta (es decir, según sean calles peatonales, zonas residenciales, carriles bici, etc.).

Un *lugar de interés* es un nodo que tiene dos datos adicionales: una lista con los tres aparcamientos más cercanos y el valor de la distancia existente desde el nodo

[4] `http://wiki.openstreetmap.org/wiki/OSM_XML`
[5] un nodo es el menor de los datos primitivos que conforman un mapa de OpenStreetMap

hasta el camino más cercano transitable por los jugadores. Es un concepto introducido para permitir los objetivos de tipo *tarea* en los modos de juego «resolver tareas» y «supervivencia».

Todos los elementos constan de un identificador y las etiquetas obtenidas de OSM, y a excepción de los nodos, también la capa en la que se pintarán. Además, cada tipo de elemento está formado por más campos:

Nodo: tiene una posición expresada en pixeles que es el resultado de la conversión de las coordenadas *WGS84* a *UTM* y éstas a su vez a las del sistema del juego. También posee dos listados con los nodos con los que está directamente conectado, uno con los nodos que son accesibles con las reglas de tránsito de los vehículos enemigos y otro con las de los vehículos del tráfico, y un tercer listado que contiene la distancia a otros nodos no directamente conectados, que se va rellenando dinámicamente durante la ejecución y sirve para evitar la repetición de ciertos cálculos (ver Anexo G.4 y Figura G.1).
Estos cálculos de nodos interconectados se pueden realizar gracias a que también contiene un listado con los identificadores de los caminos en los que está incluido este nodo.

Camino: cuenta con un listado de los nodos que componen el camino y una lista con los segmentos rectos entre los nodos. Estos segmentos se utilizan no solo para el pintado sino también para detectar si los vehículos están sobre el camino. También se incluyen diversos parámetros con propiedades para la circulación y el pintado.
Al igual que los *lugares de interés*, también incluye una lista con los tres aparcamientos más cercanos.

Multipolígono: dependiendo de la implementación usada (ver Anexo C.3.3) contiene una estructura que permite que cada camino formante del multipolígono tenga un rol definido (anillo interior o exterior del área) o un polígono representando el área. Además, en ambas implementaciones también existen diversos parámetros con las propiedades del terreno.

En el anexo C.3 se explican con mayor detalle todos estos aspectos.

2.6. Física

Debido a la simpleza de las físicas necesarias para un juego de este tipo, que no requiere un gran realismo, no se consideró necesario utilizar un motor de física

existente sino que se tomó la decisión de implementar personalmente las funciones que se consideraron necesarias.

El sistema de físicas implementado puede dividirse en cuatro algoritmos: la detección de colisiones y la aplicación de fuerzas resultantes de la colisión, ambos de forma diferenciada para colisiones con el terreno y con los actores.

La detección de colisiones con el terreno consiste en recorrer todos los elementos que conforman el terreno[6] y para cada uno de los elementos se realiza el siguiente proceso:
Se comprueba si la menor circunferencia capaz de contener al vehículo colisiona con el menor rectángulo capaz de contener al elemento. Si esta comprobación devuelve como resultado que no hay colisión, se puede asegurar con total fiabilidad y el coste de procesamiento que ha supuesto es muy bajo. Sin embargo, si devuelve lo contrario, significaría que es posible que exista colisión y para averiguarlo se debe realizar una segunda comprobación, más costosa, que comprueba si alguno de los cuatro vértices del vehículo está en el interior del elemento del terreno.

La aplicación de dicha colisión consiste en reconocer las propiedades del terreno sobre el cual se circula y aplicar las restricciones correspondientes al vehículo: inmovilizarlo, ralentizarlo, dañarlo, etc. En los anexos C.4.1 y C.4.4 se encuentra una explicación más detallada al respecto.

La detección de colisiones con los demás actores del juego se realiza de una forma similar:
Se obtiene el menor rectángulo rotado capaz de contener al vehículo, y se comprueba si colisiona con el de algún otro actor. Si se produce esa colisión, significa que es posible que realmente colisionen, y se realiza una segunda comprobación más detallada aplicando el *Separating Axis theorem*[7]. En el anexo C.4.2 se encuentra una explicación más detallada al respecto.

Para calcular el resultado de una colisión entre actores primero se debe calcular la fuerza resultante de suma de las fuerzas de los dos vehículos implicados. Esto se calcula con el siguiente algoritmo:

1. Se obtienen los vectores velocidad de los dos implicados. Ver Figura 2.5(a)
2. Se realiza una rotación de forma que la línea imaginaria entre los dos implicados quede en el eje Y. Ver Figura 2.5(b)

[6] como se explica en el apartado 4.3, una mejora importante sería dividir el terreno en una cuadrícula y solo comprobar los elementos de su zona

[7] `http://en.wikipedia.org/wiki/Hyperplane_separation_theorem#Use_in_collision_detection`

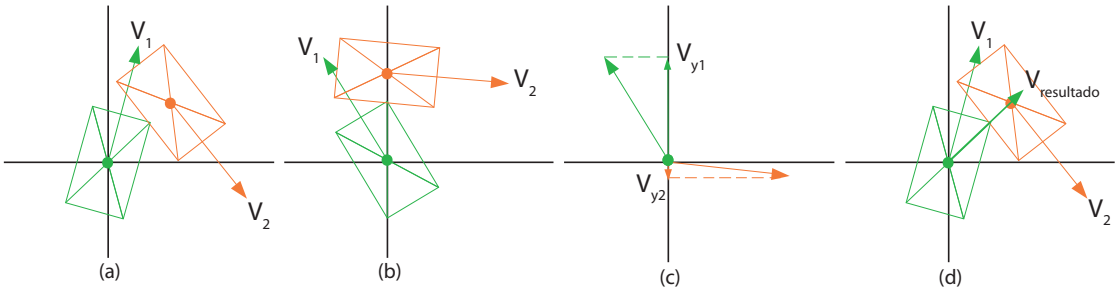


Figura 2.5: Etapas del cálculo del vector fuerza resultante de una colisión

3. El vector resultado (provisional) es la componente Y del vector velocidad del propio vehículo, salvo que se trate de un choque «por alcance» en cuyo caso se le debe restar la componente Y del vector del otro vehículo. Ver Figura 2.5(c)

4. Para asegurar que no se genere una fuerza de atracción cuando el vector está en sentido opuesto al otro implicado (sentido negativo de la Y), se debe asegurar que el valor resultado nunca será mayor a -1 (se ha elegido este valor en lugar de 0 para asegurar que siempre exista fuerza de repulsión entre los coches).

5. Se vuelve a rotar el vector resultado invirtiendo la rotación realizada en el paso 2. Ver Figura 2.5(d)

Esta fuerza resultante es sumada a la del otro vehículo, salvo que aquel se encontrase en estado «inmóvil», en cuyo caso en su lugar esa fuerza sería restada a la del vehículo propio. Posteriormente se establece la velocidad del propio vehículo a cero.

2.7. Inteligencia artificial

En `Vanet-X`, como cualquier otro juego en el que intervengan actores no controlados por jugadores, se necesita una inteligencia artificial que sea capaz de controlar estos actores. En el caso de este juego, los actores que requieren de una inteligencia artificial son los coches enemigos, las ambulancias y los coches neutrales que conforman el tráfico del escenario. Como todos estos actores son vehículos, todos ellos comparten gran parte de las funciones desarrolladas para la inteligencia, siendo sus diferencias solo pequeñas variaciones en los comportamientos.

La inteligencia desarrollada consta de tres partes:

- Unos patrones de comportamiento de alto nivel, en forma de máquina de estados, que indiquen las acciones a realizar en cada momento. (Por ejemplo: buscar aparcamiento, huir del jugador, alcanzar un punto determinado...).

- Unos patrones de manejo de la dirección del vehículo, de forma que indicando un objetivo y un modo de conducción (huir, perseguir, alcanzar...) el vehículo sea capaz de lograrlo sin salirse de las zonas permitidas para circular y sin colisionar contra otros vehículos o elementos fijos del terreno.

- Un algoritmo de búsqueda A* capaz de determinar el camino a tomar para llegar de un punto a otro, teniendo en cuenta las diferentes restricciones de paso que tienen los diferentes vehículos.

Cada una de las partes hace uso de la siguiente, es decir, los patrones de comportamiento de alto nivel usan patrones de manejo de la dirección, y éstos a su vez usan el algoritmo A* del *path-finding*.

Para profundizar en más detalle en cada uno de las tres partes anteriormente mencionadas, dirigirse al anexo C.5.

2.7.1. Comportamiento de los vehículos

Cada vehículo dotado de inteligencia propia posee unos patrones de comportamiento que indican que tipo de acciones realizarán. Estos patrones difieren según el tipo de vehículo y se componen de combinaciones de *steering behaviors*, comportamientos de bajo nivel explicados en la siguiente subsección.

En esta subsección se analizará únicamente el comportamiento de los vehículos del tráfico y un movimiento común a todos los vehículos: «desatascarse», debiéndose consultar el anexo C.5.2 para el resto de tipos de vehículos.

Desatascarse se trata de salir marcha atrás de una posición en la que se encuentra inmovilizado.
Este comportamiento es necesario ya que a pesar de los esfuerzos por mejorar la inteligencia de los vehículos, es común que por diversos motivos un vehículo pueda acabar estrellado contra los márgenes de la carretera, en cuyo caso no podrá seguir avanzando ya que no está permitido salirse de las vías habilitadas.
Este comportamiento establece como objetivo un punto situado a una distancia determinada justo detrás del vehículo y hace que avance hacia él en marcha atrás. Una vez se ha conseguido llegar al objetivo (y por lo tanto se ha desinmovilizado), el comportamiento pasa a estado normal.

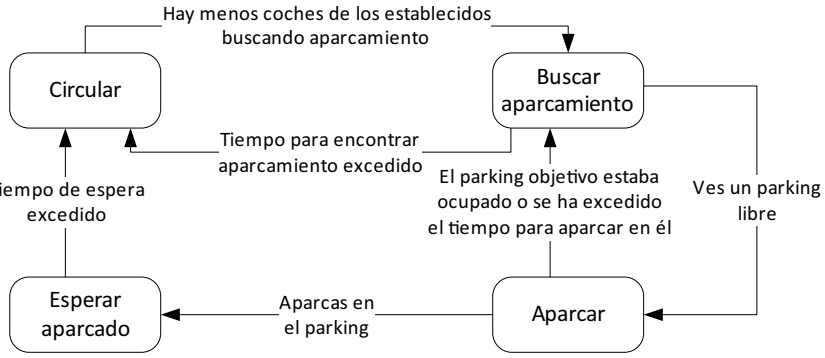


Figura 2.6: Detalle del estado *Normal* de la inteligencia de los vehículos del tráfico

Tráfico: Como se muestra en la Figura 2.6, tiene cuatro posibles subestados:

- *Circular*: consiste en elegir un nodo objetivo y alcanzarlo. Este estado se repite hasta que al vehículo le llegue una señal que indique que hay menos coches buscando parking de los establecidos. Ver Figura C.19 en el anexo C.5.2.
- *Buscar aparcamiento*: es el mismo comportamiento que el anterior pero ahora buscando aparcamientos vacíos. De esta forma si se visualiza un aparcamiento libre pasará al siguiente estado (*Aparcar*), mientras que si en un tiempo establecido no se ha logrado visualizar ninguno vuelve al estado anterior (*Circular*). Además, si se recibe un evento VESPA de parking libre se dirige a la posición del evento (aunque por el camino puede encontrar otro aparcamiento libre más cercano y aparcar en él).
- *Aparcar*: consiste en realizar la maniobra de aparcamiento sobre el aparcamiento libre objetivo. Se realiza mediante un comportamiento de llegada para que el vehículo frene al aparcar. Si se logra aparcar se avanza al siguiente estado (*Esperar aparcado*) mientras que si no se ha logrado se retrocede al estado anterior.
- *Esperar aparcado*: consiste en esperar quieto dentro de la plaza de aparcamiento durante un tiempo determinado. Una vez completado ese tiempo se vuelve al estado inicial (*Circular*).

2.7.2. Aplicando Steering behaviors

El manejo de la dirección del vehículo ha sido realizado mediante el uso de los *steering behaviors*: «comportamientos que permiten a los agentes autónomos navegar por su entorno de una manera improvisada a la vez que realista» [17].

Para este proyecto se han implementado los siguientes comportamientos sugeridos en [17]: *seek*, *flee*, *pursuit*, *evasion*, *arrival*, *obstacle avoidance*, *wander*, *path following* y *unaligned collision avoidance*. Estos comportamientos son explicados en detalle en el Anexo C.5.1.

Estos comportamientos se pueden combinar entre sí para crear comportamientos más complejos. Esta combinación de comportamientos se ha realizado según el siguiente método:
Se le asigna una prioridad a cada comportamiento y se evalúan uno a uno en orden de prioridad. Si un comportamiento concluye que debe realizarse una corrección de la dirección, se aplica esa corrección y se acaba el comportamiento. En caso de que no sea necesario un ajuste de la dirección, se evalúa el siguiente comportamiento, y así sucesivamente hasta encontrar un comportamiento que requiera un cambio de dirección o hasta haber evaluado todos.

Un ejemplo de esta combinación es el comportamiento que se le aplica a las ambulancias:
Primero, trata de no salirse de la carretera (*path following*). Si este comportamiento evalúa que no son necesarios cambios, trata de no colisionar contra obstáculos fijos (*obstacle avoidance*) primero, ni móviles (*unaligned collision avoidance*) después. Finalmente, si ninguna corrección ha sido necesaria, trata de avanzar hacia el objetivo (*seek*).

2.7.3. Búsqueda de caminos

La última de las tres partes en las que se divide la implementación realizada de la inteligencia artificial es el *path-finding* (búsqueda de caminos), que es el encargado de, a partir de un grafo de nodos que contiene la estructura de las calles, un nodo inicial y un nodo objetivo, devolver una lista ordenada de nodos que permitan alcanzar el nodo objetivo.

Como la obtención del resultado usualmente tarda más que el tiempo de ciclo del juego, la implementación se ha realizado de forma asíncrona, con un hilo de ejecución dedicado en exclusiva a recibir peticiones, ejecutar el algoritmo y devolver su resultado cuando esté listo (ver Figura 2.7).

De esta forma, los actores que hayan pedido un camino, en lugar de realizar la petición y esperar bloqueados a la respuesta, comprueban en cada ciclo si ya está lista, y en caso contrario, en lugar de no hacer nada, lo cual quedaría muy extraño, siguen avanzando hacia el objetivo mediante el *steering behavior* de llegada, de forma que para cuando obtengan la respuesta del path-finding, generalmente se habrán aproximado más al objetivo.

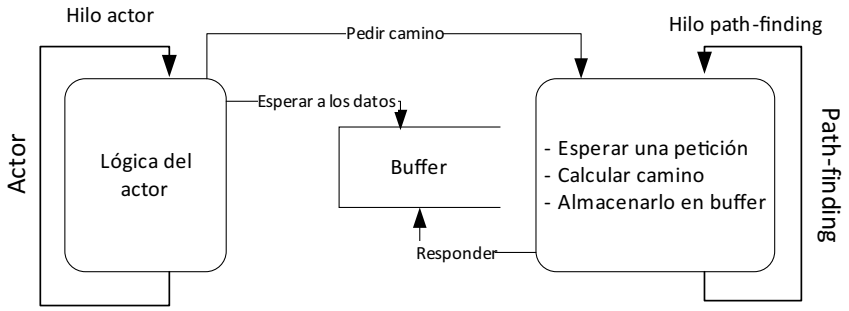


Figura 2.7: Path-finding asíncrono

2.8. Funcionamiento en red

Como se ha explicado anteriormente, la arquitectura de red de este proyecto está basada en el modelo de red del *Quake3* [13], que fue un gran éxito debido a su simplicidad y a la facilidad de entenderlo. Los modelos de red de muchos juegos posteriores son muy similares o están basados en este, como por ejemplo juegos superventas como *Half-life* (más conocido por su modificación *CounterStrike*), y su secuela, que usa el modelo de red del *Source Engine* [22], el cual también ha sido tomado como referencia para crear la arquitectura de red de este proyecto.

Las ideas obtenidas del estudio de estos dos modelos son las siguientes:

- seguir una estructura cliente-servidor con predicción en el lado del cliente [2, 12, 10, 18]
- usar compresión delta sobre el último estado conocido, para reducir el tamaño de los paquetes enviados [13]
- interpolación/extrapolación de las entidades, de forma que se prevengan los «saltos» cuando no se reciban paquetes de red [22]:

Una vez que estas ideas estaban claras, el siguiente paso fue elegir el protocolo de red que usaría la aplicación.

En un juego que necesita actualizaciones de estado en tiempo real es importante que los datos lleguen lo más rápido posible, y no necesitamos que se reenvíen los paquetes perdidos ya que al reenviarlos ya llegarían obsoletos y tendríamos que desecharlos. Es por este motivo que los modelos citados anteriormente usan el protocolo UDP, el cual tiene un mejor rendimiento a costa de no implementar características que si poseen TCP y RMI [21, 12].

En este proyecto, se ha considerado que además de usar UDP para las actualizaciones de estado, se podría combinar con el uso de TCP para ciertas tareas que no requieren velocidad de transmisión y sí confiabilidad. Por lo tanto, se han usado conjuntamente los protocolos UDP y TCP de la siguiente manera:

- UDP se ha usado para las actualizaciones de los «estados» que se envían en cada ciclo.

- TCP se ha usado para el envío del estado inicial al cliente, también para comunicaciones con clientes que todavía no se han unido al juego (cuando se quieren unir a la partida, reunir información o recibir las estadísticas finales) y para ciertos eventos poco comunes que se envían al cliente (como por ejemplo actualizaciones de la puntuación o mensajes que informan de logros nuestros o de otros jugadores[8]).

Para mejorar la experiencia de juego, se han implementado varias técnicas que aumentan la fluidez con las que el usuario percibe que funciona el juego. Se trata de la predicción en el lado cliente y de la interpolación de las entidades.

2.8.1. Modelo de red

Al igual que en los modelos estudiados, se ha seguido un esquema cliente-servidor, ya que en un juego en tiempo real donde deben enviarse múltiples mensajes de estado por segundo, un esquema alternativo *Peer-to-peer*, donde cada cliente envía información a todos los demás, es inviable salvo en conexiones extremadamente rápidas como redes locales [10].
Concretamente el modelo elegido es cliente-servidor con servidor autoritativo, lo que significa que el servidor es el único capaz de tomar decisiones que afecten al estado del juego, dejando al cliente básicamente como un «terminal tonto» cuyas únicas funciones son recoger los comandos del jugador, transmitirlos al servidor, y con los datos de su respuesta pintar el mundo de juego (aunque con la técnica de la interpolación gana más funciones). La razón de esto es que si fuera el cliente el encargado de realizar las simulaciones y enviarle al servidor su estado, como por ejemplo su posición actual, sería demasiado fácil que alguien hiciera trampas logrando que el cliente envíe al servidor datos falsos. [10]

La lógica básica del cliente y el servidor es la siguiente:
Al inicio de cada ciclo los clientes envían al servidor los comandos realizados por el jugador, el servidor procesa estos comandos, genera el estado del mundo de juego de esta secuencia o ciclo y lo reenvía al cliente, que con estos datos pinta su vista del mundo de juego.
Concretamente el intercambio de mensajes entre cliente y servidor es el siguiente (Ver Figura 2.8):

- El cliente envía *InputSnapshots* que contienen los eventos de teclado, el identificador de jugador al que corresponden, y los siguientes datos de control:

[8] Vease lo referido a *Mensajes GUI* en el Capítulo 2.13

el número de secuencia del cliente en el que estamos (*seq*), y el número de secuencia del servidor del último estado aplicado (*ack*).

- El servidor envía *Snapshots* que contienen los datos de los actores del juego y los siguientes datos de control: el número de secuencia del servidor en el que estamos (*seq*), el número de secuencia del servidor del estado sobre el cual se ha hecho la compresión delta (*ack*), y el número de secuencia del cliente al que corresponden los últimos eventos de teclado recibidos (*last*).

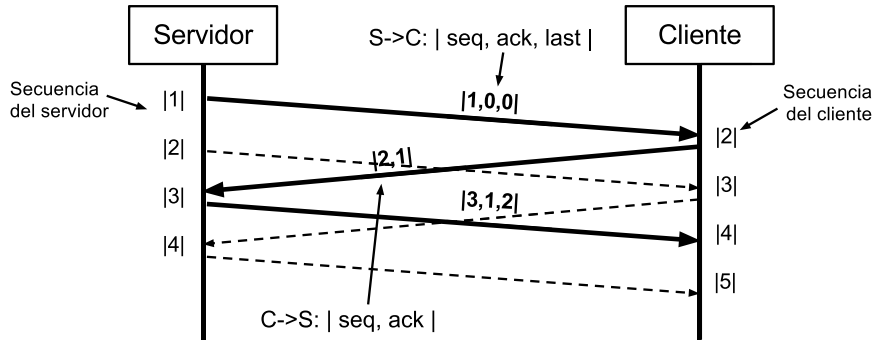


Figura 2.8: Flujo de mensajes UDP

Como este envío por parte del servidor del estado del mundo de juego se realiza 25 veces por segundo, y este estado contiene mucha información, realizar esta tarea supondría un ancho de banda inasumible para conexiones a través de internet. Por ello, como en los modelos anteriormente citados, se aplica la *delta-compression* (compresión delta). Ésta consiste en no enviar en cada secuencia el estado al completo, sino solamente los cambios respecto al último estado del que se ha recibido un *ack* por parte del cliente, lo cual supone que el servidor debe guardar cual es el ultimo *ack* recibido de cada cliente y también todos los estados enviados a dicho cliente con una secuencia superior a la de dicho *ack*. Para más detalles consultar el anexo C.6.4.

Además de esta técnica, para reducir más el tamaño de los paquetes con el estado del servidor se ha decidido que solo se envíe al cliente datos sobre los actores que se encuentran a menos de una determinada distancia de él[9]. Esto supuso numerosos problemas durante su implementación ya que no se había planificado correctamente y se hubo de rehacer múltiples funciones de la interpolación y predicción para soportarlo, como se explica en el Anexo C.6.5.

[9]modificable mediante fichero de configuración

En los anexos C.6.6 y C.6.7 se explican otras optimizaciones menores realizadas y el proceso de unión de un jugador a la partida, con los mensajes de red intercambiados por cada entidad que interviene.

2.8.2. Predicción del lado cliente

La predicción del lado del cliente es una técnica introducida por primera vez por *John Carmack* en el modelo *QuakeWorld*[10] y que tiene como objetivo reducir la latencia de los movimientos del jugador.

Sin el uso de esta técnica, cuando un jugador realiza un movimiento, al tratarse de un modelo cliente-servidor con el servidor autoritativo, debe esperar a que el comando enviado llegue al servidor, y éste le devuelva su nueva posición. Esto causa que el jugador note un retardo entre sus ordenes y el momento en que se ven cumplidas (ver figura 2.9).

Esta técnica consiste en lo siguiente: el cliente cuando reciba un comando del jugador, además de enviarlo al servidor, debe predecir lo mas fielmente posible el resultado que le va a ser devuelto, y debe aplicarlo en la vista del cliente. De esta forma, cuando llegue el resultado del servidor, si la predicción ha sido correcta, el nuevo estado del jugador coincidirá con el recibido y por lo tanto se habrá disimulado la latencia entre cliente y servidor.
Esta técnica no significa que el servidor deje de ser autoritativo, ya que siempre se aplicará el resultado del servidor cuando llegue, de forma que, si la predicción había sido incorrecta, ésta se verá corregida. Existen diversos métodos para que corregir esta desviación no suponga cambios demasiado bruscos en el cliente, pero se ha considerado que estaban fuera del alcance de este proyecto y que el tiempo de implementarlos sería más útil en otros aspectos del juego.

En el anexo C.6.3 se explica este método en más profundidad.

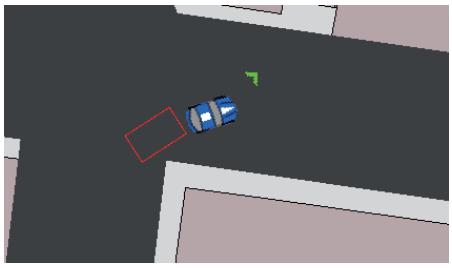


Figura 2.9: Captura de pantalla en la que se observa la diferencia entre la posición predicha y la recibida (rectángulo rojo)

[10] `http://es.wikipedia.org/wiki/QuakeWorld`

2.8.3. Interpolación de entidades

La interpolación de entidades es una técnica que se aplica en el cliente. Consiste en no procesar en cada ciclo el estado que recibes sino mantener un buffer de estados recibidos pendientes de ser procesados, de forma que, si el estado correspondiente a un ciclo se pierde, se pueda conseguir una aproximación interpolando entre los valores del último estado procesado y del siguiente estado de los recibidos. Con este método se consigue evitar los «saltos» que se producen en las posiciones de los actores cuando falla la recepción de varios estados consecutivos. Como contrapartida, esta técnica añade una latencia ya que no procesas directamente los estados. Por lo tanto teniendo un buffer de tres estados, y recibiendo un estado cada 40 ms, se crea artificialmente una latencia de 120 ms.

La interpolación se puede realizar siempre que se tengan estados pendientes de procesar en el buffer, pero cuando estos se acaban, todavía se puede aplicar otra técnica complementaria: la extrapolación. Así se hace en el caso del modelo de red del *Source Engine* [22].
En `Vanet-X` se ha tomado una solución diferente: en lugar de extrapolar los datos, se predice el comportamiento de los actores, con lo cual se obtienen unos resultados más precisos.
Aún así, contra más paquetes se pierdan más grandes serán los errores de la predicción. Por ello, se ha tomado la decisión de sólo calcular las cinco primeras predicciones ya que después no solo el error será muy grande sino que también se estará cargando de demasiado trabajo a la CPU.

Los valores a partir de los cuales la interpolación y extrapolación dejan de aplicarse son ajustables mediante el fichero *ParamConfig.txt* situado en el directorio base del juego.

En la Figura 2.10 se puede observar el comportamiento de la interpolación aplicada junto a la predicción. Para más detalles consultar el anexo C.6.2.

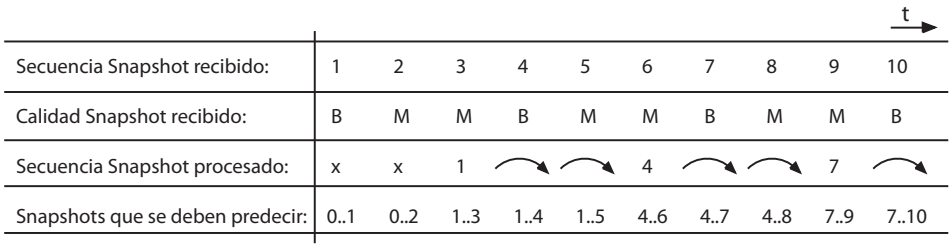


Figura 2.10: Predicción e interpolación (buffer de 2 estados) aplicadas conjuntamente. Leyenda: `B`: buena, `M`: mala, `X`: sin procesar, `flecha`: interpolación

2.9. Sistema VESPA

Esta es una de las partes más importantes del juego por ser uno de los objetivos del proyecto y también ser, junto al uso de escenarios reales, uno de los elementos diferenciadores del juego.

VESPA[11] es una *VANET* (*Vehicular Ad Hoc Network*), un sistema diseñado para que los vehículos puedan compartir entre sí información sobre eventos relevantes, como por ejemplo accidentes, plazas de aparcamiento libres, etc.

Este sistema proporciona gran parte de la información que se muestra en el radar, como la posición de otros jugadores, de coches enemigos, de vehículos en servicio de emergencias, de obstáculos en la calzada...

La implementación de VESPA ha sido realizada de forma modular para que con poco esfuerzo puedan ser modificadas partes del sistema o incluso sustituir la implementación entera por otro sistema de intercambio de información. Además, la implementación está realizada de forma que desde el menú de configuración se pueda cambiar todos los parámetros que intervienen en el funcionamiento de VESPA.

En el Capítulo 3 se profundiza sobre las características desarrolladas para la evaluación de las técnicas de gestión de datos propuestas en VESPA y en el anexo D.1 se muestran más detalles sobre el propio sistema y cómo se ha incorporado al juego.

2.10. Menú de pausa

Se ha desarrollado un típico «menú de pausa» (con la salvedad de que no se pausa el juego) que puede ser desplegado en cualquier momento de la partida.
Este menú se maneja mediante eventos de ratón, y es pintado sobre una capa semitransparente que cubre la pantalla de juego. Todo el menú es también semitransparente y se integra visualmente con el resto de los elementos del juego (ver figura 2.11).

Desde él se puede modificar (por separado) el volumen de la música y de los efectos de sonido, mostrar la configuración actual de los controles de juego y también desde él se puede abandonar la partida para volver al menú principal.

2.11. Sonido

Para el sonido se han implementado dos gestores diferenciados: uno para los efectos, que trabaja con archivos *wav*, y otro para la música, capaz de usar archivos

[11] `http://www.univ-valenciennes.fr/ROI/SID/tdelot/vespa/index.html`

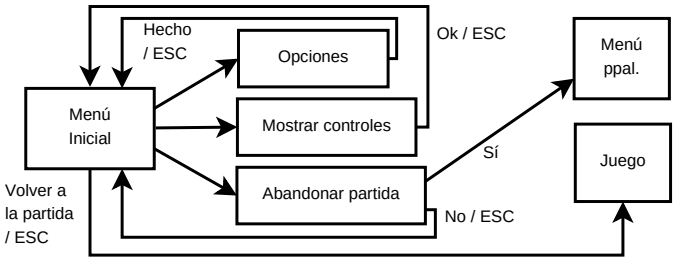


Figura 2.11: Diagrama navegación menú de pausa

mp3.
El uso de este segundo gestor de sonido vino motivado por la necesidad de usar archivos *mp3* para la música, por necesidad de espacio, mientras que seguía siendo interesante también contar con el primer gestor ya que los efectos de sonido en formato *wav* tienen una mayor calidad y al ser sonidos tan breves no suponen un problema de espacio.

El gestor de efectos está basado en el código fuente adjunto de [4][12], modificado para añadirle un control de volumen. Otras características de este gestor son que también se ha implementado un filtro 3D (para que el volumen de los sonidos se vea afectado por la distancia del emisor al oyente) y que tiene tantos hilos de ejecución como canales de audio son admitidos (por defecto, usa 32).

Por otro lado, el gestor de música se trata de la librería *JLayer*[13] modificada para admitir el control del volumen y para lograr una reproducción en bucle de la lista de canciones a reproducir.

2.12. Modos de juego y gestión de rondas y objetivos

`Vanet-X` tiene varios modos de juego, que si bien tienen ciertos aspectos en común (sistema basado en rondas, obtener puntos al lograr los objetivos...) tienen mecánicas diferentes. Para dar soporte a estas diferencias a la vez que se mantiene la independencia del resto de la implementación sobre el modo de juego en uso, se ha realizado la siguiente implementación: (Ver Figura 2.12)

Una clase «Ronda» que es el único punto de conexión con el resto del juego, y que utiliza las funciones de la interfaz «IObjetivos» para implementar las funcionalidades básicas necesarias (decidir si una ronda ha sido superada, establecer cuando comenzará la siguiente ronda, crear los objetivos...). El control de las rondas se realiza de la forma expresada en la Figura 2.13.

[12] `http://www.brackeen.com/javagamebook/#download`
[13] `http://www.javazoom.net/javalayer/javalayer.html`

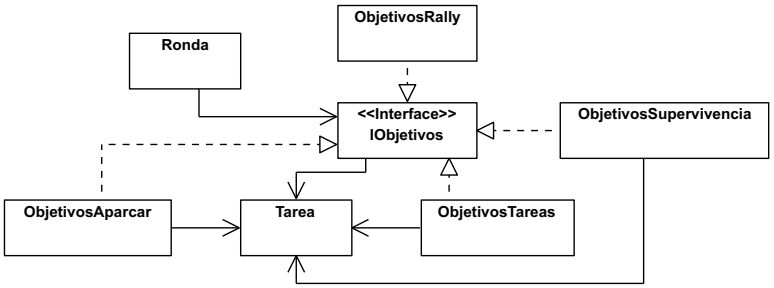


Figura 2.12: Diagrama de las clases de la gestión de rondas y objetivos

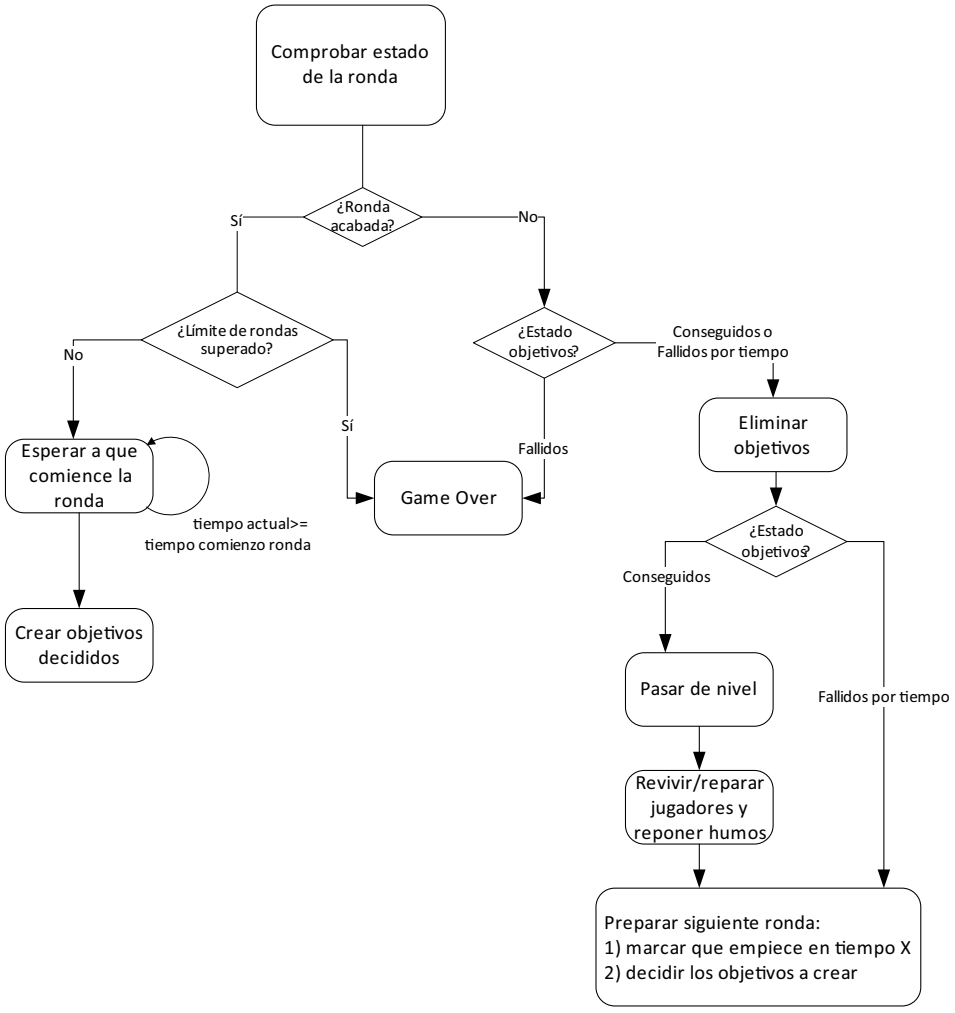


Figura 2.13: Diagrama de la gestión de rondas

Existen cuatro clases que implementan la interfaz «IObjetivos», una por cada modo de juego, adaptando las funciones definidas para crear la mecánica de juego deseada en cada modo.

Para los modos de juegos que no consisten en capturar banderas o vehículos sino en completar objetivos, se ha creado una nueva estructura llamada «Tarea» (ver anexo C.7.1), que incluye todos los datos requeridos de un objetivo: tipo de lugar, tipo de llegada y distancia al primer y tercer aparcamiento más cercano. También debido a las necesidades de estos modos se han incluido las plazas de aparcamiento (ver anexo C.7.2) y la posibilidad de abandonar el vehículo (ver anexo C.7.3).

Por último existe una clase destinada a contener todas las características de los modos de juego, como por ejemplo: el tipo de objetivo de una ronda (banderas, enemigos, tareas), el comportamiento de la inteligencia de los enemigos (perseguir, huir), si los jugadores sufren daños, etc.

2.13. Mensajes durante el juego

Durante la partida, existen varios medios con los que el jugador recibe información. Estos son los *mensajes GUI* y la *explicación de la ronda*.

Los *mensajes GUI* son un sistema mediante el cual se le suministra al jugador información sobre ciertos eventos relevantes como por ejemplo: la unión de un nuevo jugador a la partida, los puntos obtenidos por lograr un objetivo, el dinero que ha costado la reparación del vehículo, un logro de otro jugador, y muchas otras más.
Estos mensajes están visualmente muy integrados de forma que no molestan en la conducción pero a la vez son fáciles de percibir. Además, con el sistema desarrollado, cada evento puede definir su duración en pantalla y esta asegurado que en caso de acumularse varios mensajes no se superpondrán sino que mantendrán su orden y se irán mostrando uno a uno dejando un breve lapso de tiempo entre cada uno para que el cambio de mensaje sea perceptible.

La *explicación de la ronda* es un sistema que muestra información sobre la ronda actual. Esto es totalmente necesario ya que, como se ha mencionado anteriormente, el sistema de juego funciona mediante un sistema de rondas continuas sin abandonar la partida para volver a los menús. Por lo tanto resulta necesario un sistema que muestre durante la partida la información que necesita conocer el jugador. Esta información depende del modo de juego elegido pero tiene varios elementos comunes: el número de ronda en la que nos encontramos y el número total de rondas en caso de que exista un límite, el número de perseguidores enemigos, el tiempo disponible para completar la ronda, los objetivos de la ronda.

Esta explicación se muestra durante el llamado *tiempo entre rondas*[14] incluyendo una cuenta atrás del tiempo restante antes de comenzar la siguiente ronda. También se muestra parecida información (más resumida) pulsando en cualquier momento de la partida la tecla *Mostrar información* (por defecto tecla «Q»).
Todo esto se muestra en una zona rectangular semitransparente ubicada en la zona central superior de la zona de juego. Esta zona adapta dinámicamente su tamaño según los datos que deba mostrar.

En la figura 2.14 se muestra una captura del juego en la que se aprecia la explicación de la ronda (zona superior, en texto amarillo) y un *mensaje GUI* (en texto blanco en la zona inferior).

Figura 2.14: Captura de pantalla en la que se muestra la *explicación de la ronda* y un *mensaje GUI*

[14]configurable desde la pantalla *Configuración de las reglas de juego*

Capítulo 3

Explotación

En esta sección se analiza la explotación del juego como método para probar diferentes técnicas de *data sharing* y se expone el trabajo realizado para permitirlo, así como las ventajas y limitaciones existentes.

3.1. Motivación

Los análisis de diferentes técnicas de *data sharing* suelen realizarse mediante el uso de simuladores (p.ej. TraNS [16], SUMO [1], Veins [20], GrooveNet [15], o VanetMobiSim [11]) ya que realizar pruebas en un escenario real con un número significativo de vehículos sería un método caro y poco práctico. Aún con el uso de simuladores, el análisis de estas técnicas puede seguir siendo una tarea ardua que lleve mucho tiempo, ya que los resultados de muchas técnicas de *data sharing* dependen de la elección de ciertos parámetros, y puede no ser fácil determinar cuál sería una buena elección de estos parámetros.
Por este motivo se ha considerado que, de forma complementaria al uso de simuladores, se podría aplicar una estrategia de *crowdsourcing* mediante el uso de este juego de forma que se extraigan ciertas estadísticas de las partidas y los jugadores, mientras se divierten con un juego de coches, estén en realidad ayudando a calibrar estrategias de *data sharing*.

La idea de beneficiarse de acciones humanas para mejorar o evaluar sistemas no es nueva, ya que se lleva tiempo usando para tareas demasiado costosas usando los métodos tradicionales. Ejemplos de esto serían *mCrowd* [25], que usa los sensores de los smartphones para participar en tareas colaborativas como la monitorización del tráfico de carreteras, o *reCAPTCHA* [24], que se utiliza para evitar el uso de ciertos servicios por usuarios no humanos a la vez que sirve para mejorar la digitalización de textos.

3.2. Aplicación

Para llevar a cabo la recolección de estadísticas durante la partida se han habilitado varias opciones (en el fichero de texto de configuración *ParamConfig.txt*) que habilitan y deshabilitan de forma individual la recolección de distintos tipos de datos.
Estos tipos de datos que se recopilan son los siguientes (ver Anexo D.2.3 para los detalles las estadísticas recogidas):

- Estadísticas del juego: son las estadísticas que se muestran a los jugadores al finalizar la partida y comprende la puntuación de cada jugador, equipo al que pertenece y estado (habilitado o deshabilitado) de su protocolo *DMS* (*Data Management Strategy*).

- Estadísticas de los aparcamientos: se muestra cada aparcamiento realizado, tanto por jugadores como por vehículos del tráfico, incluyendo el tiempo requerido, el identificador del vehículo que lo realiza, si dicho vehículo tenía el protocolo *DMS* activado, con que tipo de protocolo contaba y si dicho aparcamiento fue facilitado por el uso del *DMS*.

- Estadísticas del protocolo VESPA: muestra de forma individualizada por vehículo, y también de manera conjunta, diversas estadísticas relacionadas con el funcionamiento y la eficiencia del protocolo (p.ej. número de eventos creados, porcentaje de eventos considerados relevantes, etc.).

Cuando la partida finaliza, estos datos recogidos se almacenan en ficheros en el computador donde reside el servidor junto con un fichero que contiene la información sobre todos los aspectos de la configuración de la partida, de forma que a partir del contenido de este fichero sea posible configurar una nueva partida con las mismas condiciones. Este almacenamiento se realiza únicamente en la máquina servidor y no en los clientes ya que los jugadores no tienen porqué tener interés en estos datos.

Al realizarse esta recopilación de datos únicamente en el servidor, en el caso de crearse múltiples partidas en diferentes computadores sería necesario recopilar de forma manual las estadísticas producidas. Para evitar esto, se han ideado dos sistemas que permiten un mayor control sobre las estadísticas: la creación de un servidor de recogida de estadísticas y la creación de un servidor dedicado.

El servidor de recogida de estadísticas es un proceso que está en permanente ejecución en un computador externo, y al que los servidores se conectarán al finalizar las partidas con la finalidad de transmitirle las estadísticas recopiladas durante la partida. (Ver Anexo D.2.2)

El otro sistema ideado consiste en el uso del servidor dedicado (ver Anexo D.2.1), que consiste en un proceso en permanente ejecución en un computador y que acepta las peticiones de conexión de los clientes (realizadas de la misma forma que para unirse a una partida normal) y, en el caso de que no haya una instancia del servidor en funcionamiento, la crea y les redirige para que se unan a dicha partida. De esta forma, un desarrollador/probador de un protocolo *DMS* puede habilitar un lugar en el que los jugadores se puedan unir a una partida de forma que no necesiten crear partidas propias y centralizando así las estadísticas en dicho lugar.

Estos dos sistemas son complementarios, de forma que el servidor dedicado al finalizar la partida tratará de conectarse con el servidor de recogida de estadísticas, si es que está activo, para comunicarle las estadísticas obtenidas.

En la Figura 3.1 se muestra la forma en que los diferentes componentes del juego están distribuidos en la red.

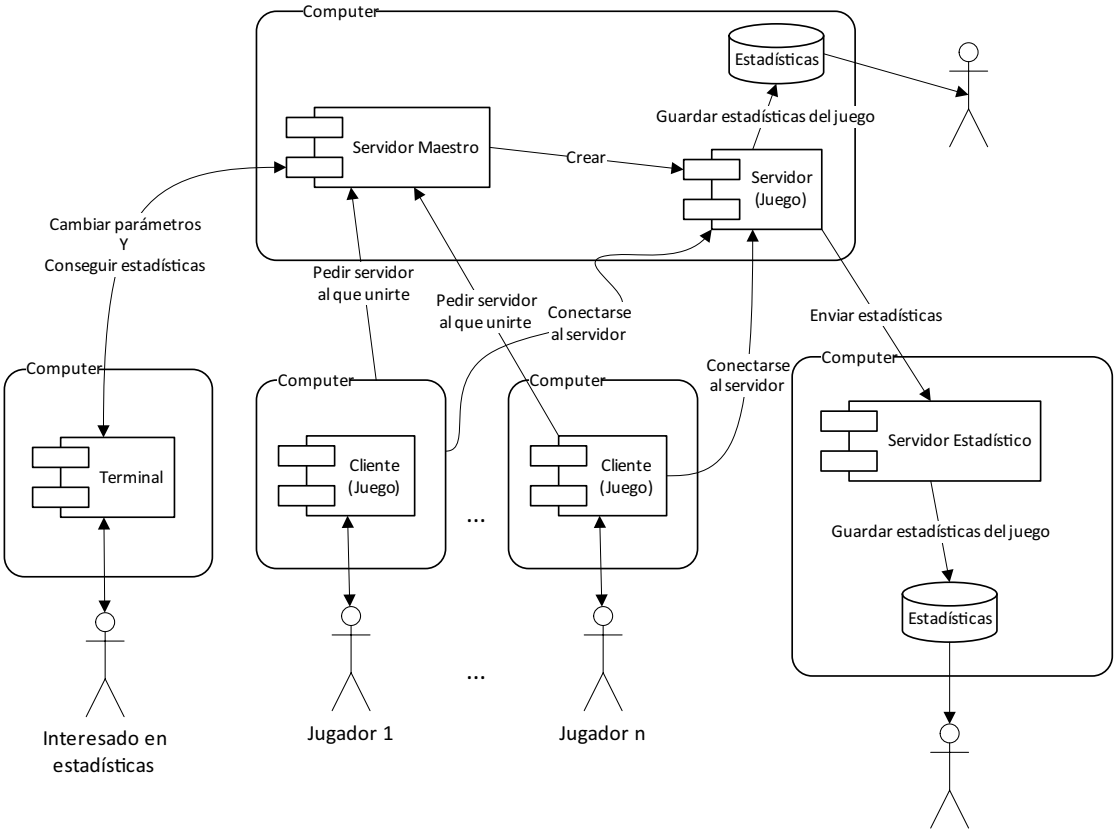


Figura 3.1: Despliegue de los componentes en una red

Además de estas características previamente mencionadas, se ha realizado la implementación de VESPA de forma que sea sencillo adaptar el juego para diversos protocolos o estrategias de gestión de datos. Esto se ha realizado definiendo las interfaces y clases abstractas básicas que se necesitan para poder interaccionar desde el juego con la *DMS*. Posteriormente se ha realizado una implementación del sistema VESPA como instanciación de dichos interfaces, haciendo uso del patrón de diseño *Factory method* (ver Anexo D.1 para más detalles).

Las más importantes de las interfaces definidas son las siguientes:

- ***IDataManagementStrategy***: declara los métodos que deben ser implementados por una *DMS* para permitir su integración con el videojuego.
- ***IVisible***: debe ser implementada por todas aquellas entidades que puedan ser «observadas» por el *DMS*.
- ***IVehicle***: debe ser implementada por todos los vehículos observables por el *DMS*.
- ***IPlayerVehicle***: debe ser implementada por todos los vehículos humanos observables por el *DMS*.
- ***ITrafficVehicle***: debe ser implementada por todos los vehículos del tráfico observables por el *DMS*.

Como se puede observar la primera interfaz es usada para que el juego se comunique con el *DMS* mientras que las cuatro restantes se usan para que el *DMS* pueda acceder a los datos de las entidades del juego. La figura 3.2 proporciona una visión de conjunto sobre la conexión entre el juego, las interfaces definidas y la implementación desarrollada.

3.3. Limitaciones

El uso de un juego como método de análisis tiene diversas limitaciones, por lo que no se plantea su uso como reemplazo del simulador sino como complemento.

Una es la dificultad de conseguir una simulación precisa y al mismo tiempo un juego divertido y jugable, ya que la inclusión de regulación semafórica, normas de circulación o tráfico más realista supondría una mayor fidelidad en la obtención de resultados pero sin embargo podría alterar la dinámica del juego disminuyendo su capacidad de atraer jugadores y por lo tanto disminuyendo la cantidad de estadísticas que se podrían obtener.

Así mismo tratar de aumentar la precisión de la simulación, ejecutar protocolos de gestión de datos complejos o tratar de simular una cantidad de tráfico elevada está

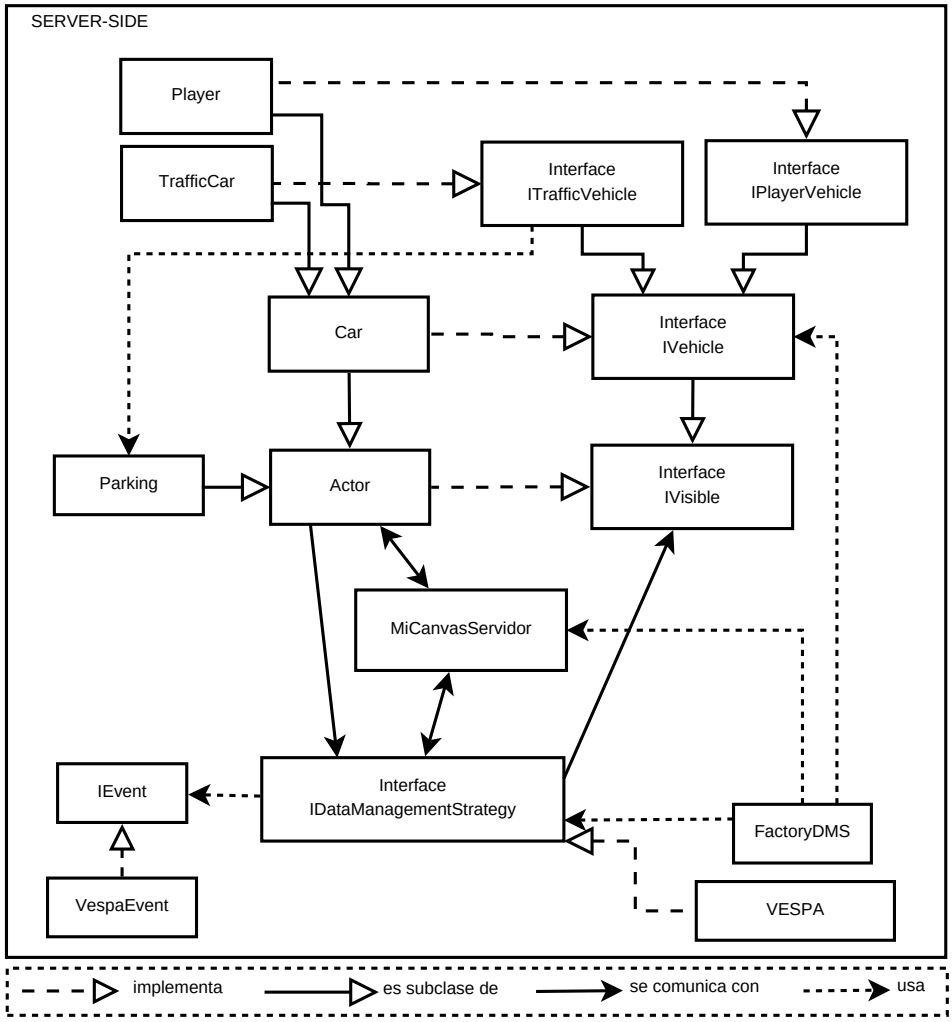


Figura 3.2: Arquitectura de la la conexión entre el juego y el *DMS*

enfrentado con la obtención de un rendimiento aceptable en el juego (en número de fotogramas por segundo).

Además otra dificultad radica en que los resultados obtenidos pueden depender de la pericia de los jugadores, por lo que comparar resultados sin tener este factor en cuenta puede conducir a conclusiones erróneas. Por este motivo es preciso comparar siempre resultados obtenidos con jugadores de similar pericia.
Para solventar esta dificultad, se ha implementado un sistema de cálculo de habilidad, que otorga a cada jugador un determinado nivel de pericia basado en su rapidez completando misiones durante las partidas y se mide en tareas completadas por unidad de tiempo. Este nivel de pericia se va modificando mediante la participación en nuevas partidas y queda reflejado en las estadísticas obtenidas, de forma que puede optarse por comparar solo resultados provenientes de jugadores de similar nivel de pericia.
Adicionalmente, existe una opción (configurable en *ParamConfig.txt*) para no permitir unirse a tu partida a jugadores con menos de un determinado nivel de habilidad.

3.4. Ventajas

Las ventajas que aporta este método de análisis son la introducción del componente humano, que es algo que no está presente en un simulador, y el hecho de que la utilización masiva del juego podría ayudar a identificar parámetros o alternativas de gestión de datos prometedoras que luego podrían verificarse por simulación.

Además, en vista de los resultados obtenidos, a pesar de que los valores absolutos difieren de los obtenidos mediante el simulador, los valores relativos observados sí que son coherentes.

3.5. Elementos añadidos al juego

Diversos elementos han sido añadidos al juego con el objetivo de facilitar la explotación del mismo como método de análisis de técnicas de *data sharing*: las plazas de aparcamiento, el modo de juego consistente en pruebas de aparcamiento y el cambio automático de estrategia de gestión de información.

- Se añadieron las plazas de aparcamiento, con el fin de contar con un elemento más en el que medir la eficiencia de las técnicas de *data sharing* usadas.
  Por motivos de rendimiento se crearon dos tipos de plazas de aparcamiento: las «reales» y las «falsas». Las reales son plazas inicialmente vacías que los vehículos pueden ocupar y desocupar según sus necesidades, mientras que

las falsas son plazas que siempre están ocupadas y su inclusión viene determinada para aparentar visualmente la existencia de un gran número de aparcamientos, reduciendo la monotonía del escenario, sin necesitar aumentar el número de vehículos del tráfico para que las ocupen y desocupen, con el coste computacional que ello supondría.

Para evitar que el jugador sea capaz de distinguir entre los tipos de plazas, y por lo tanto sepa que plazas ocupadas se terminarán por desocupar y cuáles no, se han utilizado diversas técnicas. Como los vehículos del tráfico no realizan aparcamientos perfectos, se tomó una doble estrategia: por un lado se evitó que las plazas falsas tuvieran una apariencia de aparcamiento perfecto, modificando el *sprite* que la representa de forma que el vehículo que se muestra aparcado esté ligeramente rotado y trasladado respecto a su posición ideal, y por otro lado, en lo referente a las plazas reales, se creó un método mediante el cual los vehículos mal aparcados en ellas son colocados de la forma correcta cuando no exista ningún jugador a menos de una determinada distancia que pueda observar la traslación cometida.

Para poder obtener resultados comparables a los resultados del simulador de VESPA, se estableció un ratio fijo entre el número de vehículos que buscan aparcamiento y el número de plazas existentes. Para conseguirlo, se programó la inteligencia de los vehículos del tráfico de forma que haya siempre un número permanente de vehículos buscando aparcamiento y cuando uno de ellos logre aparcar, otro vehículo que estuviera simplemente circulando se ponga a buscar aparcamiento inmediatamente.

El tiempo que un vehículo permanece aparcado no es constante y varía entre 20 y 40 segundos, los cuales son valores poco realistas pero que suponen un tiempo suficiente como para que el jugador no se quede a la espera de que se libere una plaza ocupada y a su vez no son lo suficientemente grandes como para que en una partida corta apenas se produzcan liberaciones de aparcamientos.

- Se añadió un nuevo modo de juego ideado expresamente para conseguir medir los tiempos de aparcamiento, de forma que mediante objetivos se facilitase la toma de datos. Este modo de juego consiste en un sistema de objetivos en el que cada ronda tiene un doble objetivo que se debe cumplir de forma secuencial: primero ir a un punto establecido (dado mediante una dirección o mediante un sitio de interés) y una vez logrado este objetivo, encontrar aparcamiento cercano (a menos de una distancia establecida por defecto en 500m pero modificable en *ParamConfig.txt*). Cuando se logra el objetivo de aparcar, se avanza de ronda y se repite este esquema hasta lograr el número de rondas seleccionadas en la configuración de la partida.

Este modo de juego se realiza con vida infinita y sin vehículos enemigos, y puede ser jugado únicamente en modo competitivo.

- Se desarrolló una opción que permite el cambio automático del protocolo cada cierto número de rondas en una misma partida, ya que se deseaba poder realizar pruebas con VESPA y el protocolo de reserva activados, con VESPA y sin protocolo de reserva y, por último, sin VESPA, lo cual suponía tener que iniciar una nueva partida con cada cambio de protocolo. De esta forma se permite recolectar datos de diversas configuraciones en exactamente el mismo escenario (ya que las situaciones de los aparcamientos son aleatorias y cambian entre partidas).

3.6. Posibles mejoras de VESPA y problemas encontrados

Durante la implementación del sistema VESPA en el juego se han encontrado los siguientes problemas y posibles mejoras:

- Los eventos móviles observados desde un agente externo, como se explica en el Anexo D.1.5, hacen necesario un método para calcular los vectores de movilidad y dirección.

- Cuando se envía un mensaje y no se obtiene respuesta (bien sea porque no hay ningún vehículo cercano o no lo consideran relevante) hay que reenviarlo periodicamente hasta que se obtenga respuesta. Este reenvio puede prolongarse durante grandes periodos de tiempo (p.ej. si el vehículo circula por un camino con una densidad de vehículos muy baja), por lo que podría ser de utilidad calcular de nuevo la EP[1] antes de cada envío ya que puede llegar un punto en el que ya no sea necesario seguir intentando difundir el evento porque haya quedado ya obsoleto.

3.7. Resultados experimentales

Con el propósito de evaluar el interés de `Vanet-X` como método de evaluación de estrategias de gestión de datos, se han desarrollado varios experimentos. En estos experimentos se ha evaluado el tiempo que le cuesta a un vehículo encontrar un aparcamiento libre, ya que los aparcamientos representan un tipo recurso escaso, perfecto para probar las bondades de las estrategias de gestión de información.

[1] probabilidad de encuentro, ver Anexo D.1.3

La configuración de los diversos parámetros de VESPA utilizada para la realización de los experimentos se muestra en la figura 3.1.

Rango de comunicación	$200m.$
Vehículos equipados con VESPA	50 %
α (coef. penalización sobre la distancia)	$1/1500$ ($\Delta d \leq 500m.$)
β (coef. penalización sobre el tiempo hasta la posición más cercana)	$1/180$ ($\Delta t \leq 60s.$)
γ (coef. penalización sobre la edad del evento)	$1/360$ ($\Delta g \leq 120s.$)
ζ (coef. penalización sobre el ángulo)	$1/270$ ($c \leq 90^{o}$)
Umbral de relevancia	75 %
Umbral de difusión	75 %
Umbral de almacenamiento	60 %
Refresco del *Query processor*	cada $2s.$
D (tiempo máximo de espera antes de la redifusión de un mensaje entrante)	$1s.$
D' (tiempo máximo de espera antes de la redifusión de un mensaje saliente)	$2s.$

Tabla 3.1: Configuración de VESPA

Se han considerado tres estrategias diferentes de compartición de información:

- VESPA sin protocolo de reserva (*VESPA-P*): se ha adaptado la propuesta de [5], desarrollada en el contexto del sistema VESPA [6].
- VESPA con protocolo de reserva (*VESPA+P*): los recursos escasos, como son los aparcamientos, pueden causar problemas de competición por un único recurso. Por ello se ha adaptado el trabajo presentado en [7], que añade un protocolo que coordina el procedimiento de la reserva del recurso, de forma que la información sobre dicho aparcamiento sea transmitida a un único interesado.
- Sin compartición de información: los vehículos no reciben ninguna información. Solo conocen los aparcamientos que están en su rango de visión.

El escenario utilizado para la realización de los experimentos es un área de $1km^2$ en torno a la calle «Sophie Oury» en la ciudad de Valenciennes (Francia). En este escenario se han simulado un número variable de vehículos circulando, y se ha medido el tiempo que necesitan para encontrar un aparcamiento libre cercano a

sus destinos programados. Se ha recogido información durante aproximadamente 14 horas de juego, que han correspondido a 400 aparcamientos realizados por el jugador.

Los resultados experimentales muestran los beneficios de contar con el sistema VESPA, especialmente usando un protocolo de reserva (ver figura 3.3). También se puede observar (figura 3.5) que los vehículos controlados por humanos obtienen un mayor beneficio que los que son controlados por el computador. Esto puede deberse a la implementación de la inteligencia artificial realizada, concretamente en lo que respecta al algoritmo de búsqueda de recursos, que propone destinos aleatorios dentro de un rango fijo, en lugar de ir incrementando este rango progresivamente.

Vehículos buscando	10	15	20	25
Aparcamientos	10			
Ratio vehículos buscando / aparcamientos libres	1	1.5	2	2.5
No humano con VESPA+P	20 %	9 %	10 %	11 %
No humano con VESPA-P	23 %	19 %	14 %	18 %
Humano con VESPA+P	26 %	28 %	32 %	28 %
Humano con VESPA-P	13 %	22 %	19 %	26 %

Tabla 3.2: Porcentaje de mejora del tiempo de aparcamiento

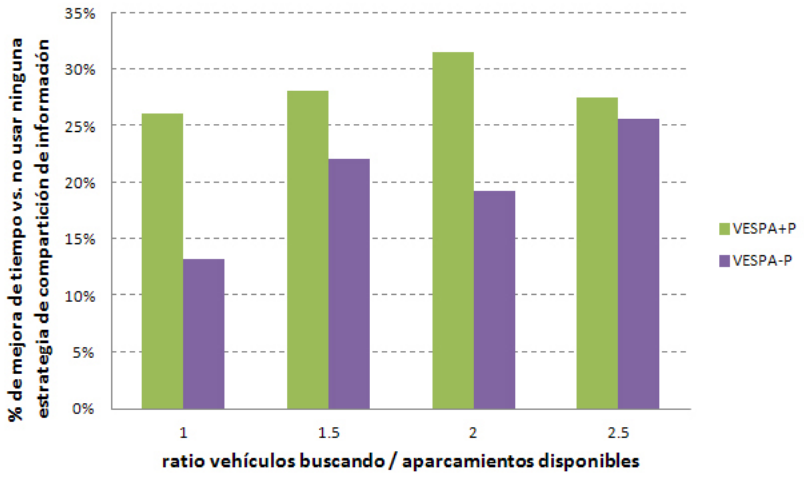


Figura 3.3: Mejoría en el tiempo para aparcar por un humano

A pesar de que se considera necesaria la realización de más test para considerar los resultados como válidos, los resultados obtenidos son consistentes con otros resultados experimentales obtenidos previamente mediante el uso del simulador.

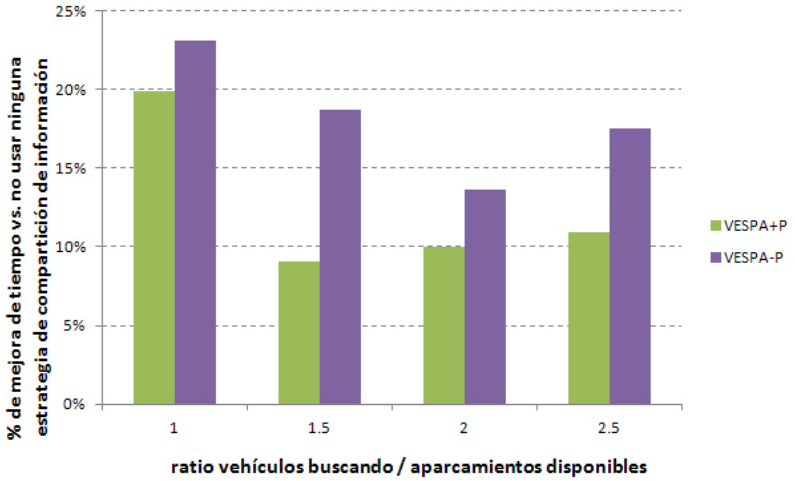


Figura 3.4: Mejoría en tiempo para aparcar por el computador

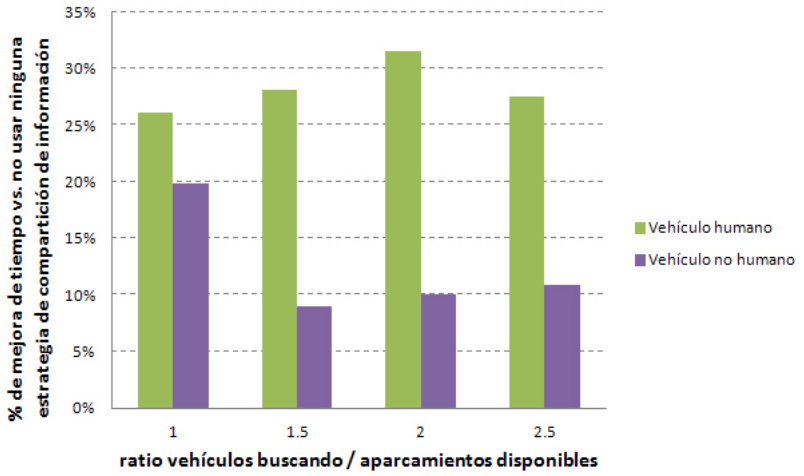


Figura 3.5: Comparativa mejoría de tiempo entre humanos y no humanos

3.8. Rendimiento del juego

El juego ha sido diseñado para funcionar con un *framerate* constante de 25 FPS (fotogramas por segundo), que coincide con la tasa de ciclos de juego por segundo. Se ha comprobado que usando la configuración de la partida más exigente de todas las posibles, este *framerate* se cumple en el ordenador utilizado para la realización de las pruebas.

Al tratarse de un juego en red es importante conseguir un tamaño reducido de los paquetes de red enviados. En la tabla 3.3 se observa el rendimiento medido para diferentes configuraciones probadas. Los datos enviados de red se refieren a los enviados por el servidor a cada cliente. Los enviados por los clientes al servidor tienen una tasa fija (si se cumple el *framerate* establecido) de 1,45 KB/s.

Núm. jugadores	Veh. tráfico	Veh. enemigos	Media datos enviados	Máximo memoria usada (servidor)	Máximo memoria usada (cliente)	Media uso CPU (servidor)	Media uso CPU (cliente)
1	10	4	20 KB/s	22 MB	37 MB	3 %	10 %
1	25	4	35 KB/s	33 MB	15 MB	4 %	11 %
2	25	4	57 KB/s	20 MB	35 MB	4 %	14 %
4	25	4	36 KB/s	25 MB	38 MB	5 %	16 %
1	50	8	57 KB/s	22 MB	35 MB	3 %	17 %

Tabla 3.3: Rendimiento obtenido con varias configuraciones

En el anexo D.3 se muestran diversas gráficas recogidas y se detallan aspectos como la cantidad de datos enviados por la red por cada tipo de entidad del juego.

Capítulo 4

Conclusiones

En este capítulo se recapitulan las conclusiones que se han obtenido de la realización de este Proyecto Fin de Carrera, repasando las diferentes iteraciones de su realización y analizando si se han cumplido los objetivos del mismo. También se presentan diversas líneas de trabajo futuro y una valoración personal del trabajo realizado.

4.1. Conclusiones

A lo largo de este proyecto se ha desarrollado un videojuego de coches, que puede ser jugado de forma cooperativa o competitiva por varias personas a través de la red, con escenarios basados en datos reales obtenidos a través del sistema OpenStreetMap, y que tiene como objetivo final permitir integrar un sistema de gestión de información en redes vehiculares, siendo elegido el sistema VESPA para su implementación.

El proyecto ha sido desarrollado siguiendo una metodología de desarrollo basada en diferentes iteraciones, de forma que se comenzase desarrollando un juego básico y en cada nueva iteración se le fuesen añadiendo funcionalidades.

Como base para el videojuego, se eligió adaptar el juego Rally-X, un *arcade* clásico de la compañía Namco del año 1980 en el que el jugador controla un vehículo a través de un laberinto de calles y debe tratar de conseguir las banderas repartidas por el escenario mientras huye de los vehículos enemigos que le persiguen, haciendo uso del lanzador de humo que permite despistarlos por unos instantes.
Este juego tenía muchas similitudes con el juego *Pac-Man* (compartían la misma placa) por lo que en realidad se puede considerar un arcade de laberintos, ya que el control del coche se limitaba a movimientos en giros de 90º.

Se pensó que sería una buena idea adaptar este juego ya que las banderas podrían representar eventos fijos del sistema VESPA y los vehículos enemigos representarían eventos móviles. Sin embargo, se decidió modificar otros aspectos para adecuarlo al tiempo actual.
De esta manera se cambió el control del vehículo de forma que ahora podría realizar giros de cualquier ángulo, pudiendo tomar una trayectoria más cerrada cuanto menos fuera la velocidad a la que circulase.
Otra modificación fue sustituir el laberinto sobre el que se circulaba por escenarios reales obtenidos a través del sistema de mapas OpenStreetMap.
Éstos fueron los objetivos de la primera iteración del juego.

A continuación se muestra el trabajo realizado en las diferentes iteraciones:

- **1ª iteración**: se tomó como base un tutorial de realización del clásico *Space Invaders* en Java[1] y la lectura del libro *Developing Games in Java* [4], lo cual aportó los conocimientos básicos sobre la estructura de un juego, el pintado o la inclusión de sonidos. La inteligencia artificial usada para los vehículos enemigos era muy básica ya que se cometió el error de tratar de realizarla partiendo desde cero.

- **2ª iteración**: consistió en añadir la capacidad de que participen varios jugadores a través de la red. Para ello se buscó información sobre las arquitecturas de red usadas habitualmente en videojuegos y se llegó a la conclusión de que una arquitectura del tipo cliente-servidor con predicción en cliente era la idónea, contando con la suerte de que este tipo de arquitectura era la más habitualmente utilizada y se disponía de suficiente documentación al respecto proveniente de los juegos *Quake3* y *Half-Life*, los cuales fueron los pioneros en usarla.
  Esta iteración fue una de las más costosas ya que a pesar de los artículos publicados al respecto, había aspectos insuficientemente documentados.

- **3ª iteración**: el objetivo fue convertir el juego, realizado hasta ahora con un único hilo de ejecución (exceptuando el sonido), en multi-hilo, de forma que cada actor tuviera su propio hilo de ejecución. Esto aparentemente entraba en conflicto con la técnica de predicción usada en la arquitectura de red, pero sin embargo no fue así, ya que esa técnica se desarrollaba en el cliente, y en éste no tenía sentido una implementación multi-hilo de los actores ya que éstos no realizaban tarea alguna más allá del pintado. De esta forma se realizó una implementación multi-hilo en el servidor y de un único hilo (más

[1] `http://balusoft.wordpress.com/2010/09/26/creando-un-space-invaders-con-java/`

los del sonido) en el cliente, ya que este último no dejaba de ser un mero terminal de *E/S* con algunas capacidades extra.
También se mejoró la representación visual, añadiendo nuevos tipos de terreno como diferentes tipos de caminos (utilizando los diferentes tipos existentes en OpenStreetMap) o los edificios, y se modificó el sistema de detección de colisiones con el terreno.

- **4ª iteración**: como se ha mencionado anteriormente, la inteligencia desarrollada para el manejo de los vehículos no humanos era muy básica y nada extensible, por lo que el objetivo de esta iteración consistió en rehacer dicha inteligencia. Vistos los resultados del intento de realizar la inteligencia desde cero, se decidió usar los diferentes comportamientos desarrollados en el artículo *Steering Behaviors For Autonomous Characters* [17], que permitían construir comportamientos complejos a partir de ellos y consistían en una aproximación basada en la diferencia entre el vector de trayectoria del vehículo y el vector hasta el punto objetivo deseado.
  Fue en este momento cuando se vio la necesidad de añadir más vehículos controlados por el ordenador: los vehículos del tráfico, que servirían para retransmitir los eventos VESPA además de para aportar más «vida» al escenario, y las ambulancias, como generador de eventos VESPA del tipo «servicio de emergencia».

- **5ª iteración**: hasta este momento el juego no contaba con un sistema de menús y al ejecutarlo ya comenzaba la partida en un mapa prefijado que se había obtenido manualmente desde el sitio web de OpenStreetMap. Por ese motivo esta iteración consistió en dotar al juego de un sistema de pantallas de menú que permitiese configurar diversos aspectos de la partida y permitir un método de añadir nuevos escenarios desde dentro del propio juego. Para esto último, se desarrolló la estructura necesaria para el almacenamiento de los escenarios descargados, ya que se consideró que los escenarios descargados debían permanecer disponibles para futuras ejecuciones de la aplicación.

- **6ª iteración**: consistió en implementar diferentes modos de juego que se acababan de definir en la reunión, para dotar mayor variedad al juego y lograr modos de juego con mayor diversión. Fue entonces cuando se implementaron los «lugares de interés», direcciones y negocios reales que serían utilizados como objetivo a alcanzar en la partida, en lugar de las banderas, y también se implementó la funcionalidad de poder salir del vehículo y avanzar a pie.

- **7ª iteración**: consistió en la implementación de una versión simplificada del sistema VESPA, añadiendo además todo lo necesario al juego para permitir su uso, como el radar en el que se mostrarían los eventos. Al final de esta

iteración se mostró el trabajo realizado hasta el momento al profesor Thierry Delot (del proyecto VESPA), realizando una presentación en inglés en la que se anotaron diversas posibles mejoras, que serían implementadas en la siguiente iteración.

- **8ª iteración**: se inició tras una reunión en la que se había realizado una prueba con los tutores del juego y se definieron multitud de ajustes y cambios de mayor calado que debían realizarse, además de nuevas características consideradas de interés. Se dedicó toda la iteración a realizar estos cambios. Fue aquí cuando se añadieron características como la cámara de muerte (permite a los jugadores muertos ver la visión del resto de jugadores para que la espera no sea aburrida) o el menú de pausa.

- **9ª iteración**: se desarrolló la implementación completa del sistema VESPA, sustituyendo a la versión simplificada que se había estado usando previamente. Esta iteración fue muy costosa, ya que fue necesaria la lectura de diversos artículos de VESPA para la comprensión del sistema, y en la implementación se contaba con el código fuente de una versión no final del simulador desarrollado para VESPA, el cuál contaba con varios errores por lo que fue necesario revisar detalladamente todo el código antes de poder usar las funciones que contenía.

- **10ª iteración**: En este momento, con el sistema VESPA ya implementado, se decidió avanzar en la explotación del juego como método de evaluación de dicho sistema, comenzando la realización del artículo finalmente presentado en *IMMoA'13*[2]. Para ello, se vio necesaria una última iteración en la que se incluyera un nuevo modo de juego consistente en realizar aparcamientos, para facilitar la toma de muestras de tiempos de aparcamiento, análisis en el cuál se iba a centrar dicho artículo. También fue en este momento cuando se implementó el sistema de atascos, aunque no se llegó a usar para el artículo.

Como se puede comprobar a continuación, se han cumplido todos los objetivos marcados inicialmente en la propuesta del Proyecto Fin de Carrera.

- Se ha desarrollado un videojuego de coches, que cuenta con vehículos controlados por el ordenador mediante la inteligencia artificial elaborada, y con otros vehículos controlados por jugadores humanos, los cuales se unen a la partida a través de la red.

- Los escenarios utilizados para las partidas están creados con datos reales obtenidos del sistema cartográfico OpenStreetMap.

[2] `http://www.dbis.rwth-aachen.de/IMMoA2013/`

- Estos escenarios pueden ser añadidos al juego de forma sencilla desde el sistema de menús, indicando una localización a través de unas palabras clave (la dirección) y seleccionando el tamaño del área a descargar.

- Se ha definido una interfaz que permite utilizar en el juego sistemas de gestión de información, siendo implementado el sistema VESPA.

Además, conjuntamente a lo realizado en este proyecto, se ha presentado un artículo (ver anexo E) al *workshop IMMoA'13*, el cual ha sido realizado en conjunción con los directores del proyecto Sergio Ilarri y Eduardo Mena.

4.2. Línea temporal de la realización del proyecto

Como ya se ha comentado anteriormente, el desarrollo del proyecto se ha realizado siguiendo una metodología de desarrollo basado en iteraciones. En esta sección se analizará el tiempo dedicado a cada iteración (tabla 4.2 y figura 4.2) así como la visión global dividiendo el tiempo en reuniones, investigación/análisis/diseño, implementación/pruebas y memoria (tabla 4.1 y figura 4.1).

Hay que tener en cuenta que cada iteración consiste no solo de implementación sino también de la investigación, el análisis, el diseño y las pruebas realizadas. Por ese motivo la duración mostrada en la tabla 4.2 se incluye en los apartados «investigación/análisis/diseño» e «implementación/pruebas» de la tabla 4.1.

También se muestra en la figura 4.3 el cronograma del desarrollo de las diferentes iteraciones. Como se puede observar, la extensión temporal de las diferentes iteraciones difieren con el valor mostrado en la columna «Duración». Esto es así debido a la superposición de la realización del proyecto con diversas actividades laborales y también con la realización del artículo presentado en el *workshop IMMoA'13*.

Tarea	**Horas**
Reuniones	27
Investigación/Análisis/Diseño	136
Implementación/Pruebas	497
Medidas de tiempos (Explotación)	51
Memoria	172
Total	**883**

Tabla 4.1: Separación de horas por tipo de trabajo

Iteración	Descripción	Horas
1	Rally-X, OSM, inteligencia basica	59
2	Red	105
3	Multi-hilo, terrenos, mejora visual	72
4	Inteligencia avanzada	73
5	Menús, gestión escenarios	24
6	Modos de juego	73
7	VESPA simple	21
8	Menú de pausa, cámara de muerte y múltiples correcciones	79
9	VESPA completo	51
10	Mejoras para explotación	76
Total		**633**

Tabla 4.2: Separación de horas por iteración

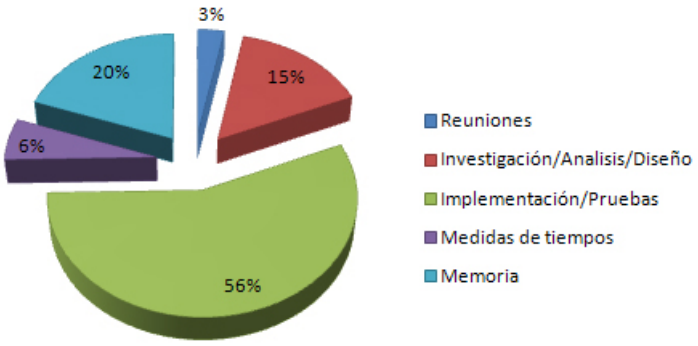


Figura 4.1: Porcentaje de horas de cada tipo de tarea

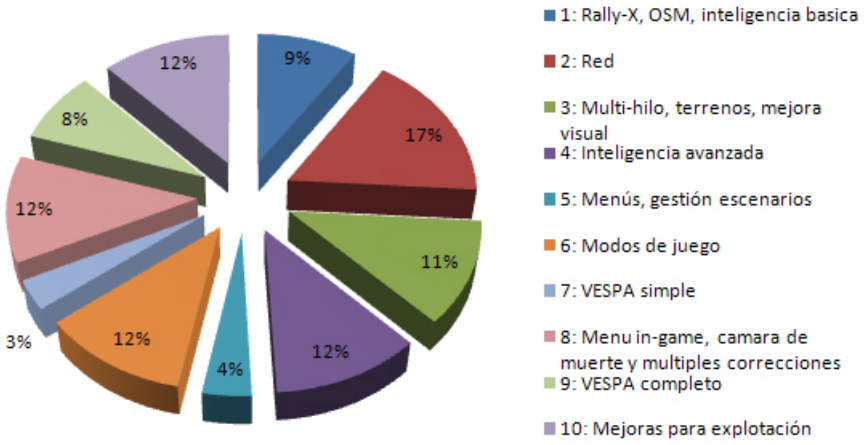


Figura 4.2: Porcentaje de tareas de cada iteración

Iteración	Comienzo	Final	Duración
1	01/08/2011	21/09/2011	59h
2	22/09/2011	13/12/2011	105h
3	14/12/2011	19/01/2012	72h
4	20/01/2012	06/03/2012	73h
5	07/03/2012	19/03/2012	24h
6	20/03/2012	31/05/2012	73h
7	01/06/2012	25/09/2012	21h
8	26/09/2012	05/12/2012	79h
9	06/12/2012	07/02/2013	51h
10	08/02/2013	11/06/2013	76h

Figura 4.3: Cronograma del desarrollo de las diferentes iteraciones. Se han marcado en naranja los periodos de nula dedicación.

4.3. Trabajo futuro

A continuación se proponen algunas posibles mejoras futuras, que se pueden dividir en varias temáticas: red, rendimiento, VESPA, IA y otros.

Sobre aspectos de red

- Usar la *Codificación Huffman* como método de comprimir los paquetes de red, al igual que se hace en el juego *Quake3* y el motor *Source* [13].

- Cambiar el uso del *TCP* «asíncrono» usado para el envío de datos fuera de orden por el protocolo UDP con características de *reliability* implementadas. Este cambio es muy deseable ya que durante la redacción de esta memoria, revisando artículos de las fuentes, se descubrió un problema de usar los protocolos TCP y UDP simultaneamente que había pasado desapercibido.
  Todos los artículos relacionados con los aspectos de la comunicación en red recomendaban encarecidamente usar el protocolo UDP (por los motivos comentados en el Capítulo 2.8.1) y además mencionaban que mezclar el uso de TCP y UDP podía causar problemas de sincronización [9].
  Después de haber leído esos artículos se desarrolló el envío de paquetes de red durante el *game loop* mediante el protocolo UDP y en la inicialización y finalización (ver Anexo B.5), en las que era la fiabilidad y no la velocidad lo primordial, mediante el protocolo TCP. Tiempo después, durante la optimización del código de red para reducir la cantidad de datos a enviar, sin recordar las advertencias de los artículos acerca de mezclar ambos protocolos, se ideó que ciertos datos que se enviaban solo cada mucho tiempo se enviasen por TCP para lograr así una ligera mejora en el tamaño de los paquetes enviados en cada ciclo. Los elementos que se decidió enviar por TCP son: las puntuaciones, la explicación de la ronda y los mensajes durante la partida (a veces referidos en este documento como *mensajes GUI*). Durante la revisión de los artículos de la bibliografía realizada durante la preparación de este documento se recordó la advertencia del peligro del uso de ambos protocolos simultáneamente, e investigando más sobre el asunto se descubrió un artículo [19] en el que se explicaba que al estar ambos protocolos implementados sobre la capa IP, el uso de TCP tiende a inducir pérdida de paquetes en UDP.
  Es por esta razón que se debe cambiar de nuevo el envío de esos tres elementos que actualmente se realiza mediante TCP para volverlo a realizar en los paquetes «Snapshot» UDP o seguir con el diseño actual pero cambiando el uso de TCP por UDP e incorporar funcionalidades que garanticen la fiabilidad de los envíos.

Sobre aspectos del rendimiento

- Implementar un método que posibilite calcular el rendimiento del ordenador, para por ejemplo decidir de forma autónoma los parámetros más apropiados para la partida o el uso o no de las pantallas estáticas mencionadas en el punto anterior. Una forma de realizar esta funcionalidad podría ser ejecutar un proceso pesado y medir el tiempo utilizado.

- Disponer de diferentes resoluciones gráficas, predefinidas de antemano y seleccionables por el usuario.

Sobre VESPA

- Realizar el cálculo de la *Encounter Probability (EP)* mediante el uso de mapas digitales en lugar de mapas geográficos.

- Usar la implementación real de VESPA como implementación de los interfaces, en lugar de hacer uso de la interfaz desarrollada para el juego.

- Hacer una vista general del juego donde se vea todo el área de juego en miniatura de forma que se pueda observar el comportamiento de VESPA (cómo se envían los eventos, quien los reenvía, etc.). Esta vista general se visualizaría desde un cliente especial que se conectase al servidor. Una funcionalidad adicional podría ser que se pudiera grabar la secuencia para posteriormente poder revisionarlo como si de un vídeo se tratara.

- Realizar una metodología para automatizar la recogida y procesado a gran escala de los ficheros de estadísticas de explotación, así como realizar una evaluación en otros escenarios (con otros tipos de eventos, etc.). Esta extensión podría ser objeto de un Proyecto Fin de Carrera que continuara con el trabajo en este sentido.

Sobre la IA

- Dividir el escenario en regiones para mejorar el rendimiento del algoritmo de detección de colisiones (para que compruebe las posibles colisiones solo con los elementos del terreno de tu región) y del algoritmo que averigua cuál es tu nodo más cercano (usado por la IA).

- Hacer que los vehículos del tráfico respeten los sentidos de circulación en los caminos de doble sentido. Para lograr este objetivo, el algoritmo de *path-finding* debe poder diferenciar los sentidos de las calles (ya está así hecho) y se debe idear algún método para que el vehículo circule siempre próximo

al borde derecho del camino. Un problema que se encontraría sería que al reducirse a la mitad el espacio por el que circulan, podrían surgir problemas de maniobrabilidad de la inteligencia de los vehículos.
Otra posibilidad sería utilizar un comportamiento similar al *Flow field following* descrito en [17], para asegurar que en cada mitad del camino la «fuerza» que guiará a los vehículos sea en distinto sentido.

Otros aspectos

- Actualmente cada tipo de terreno tiene asociadas unas propiedades (infranqueable, ralentizar, causar daño, etc.). Sería deseable poder controlar las propiedades que tendrá cada terreno según un fichero de texto de configuración.

- Mostrar textos con colores y formato, en lugar de texto plano, en los textos durante el juego. Por ejemplo para mostrar el color de un equipo en los *mensajes GUI* o en la explicación de la ronda. Para lograr esta función se podría usar la clase *AttributedText*.

- Actualmente, las unidades de medida espacio-tiempo en torno a las cuales está diseñado el juego son los pixeles, los ciclos de juego y en menor medida los segundos. Se han utilizado éstas por motivos de sencillez pero sería conveniente cambiarlo de modo que se usen únicamente las respectivas unidades del S.I (metros y segundos). De esta forma garantizaríamos que la velocidad de los vehículos sea la misma aunque la velocidad del juego (*FPS*) disminuya.

- Usar *SandMark*[3] para ofuscar el código fuente del juego y así dificultar que se puedan hacer trampas en el juego.

- Hacer que la música de la partida cambie dinámicamente según la situación actual. Por ejemplo una música con un ritmo más rápido en situaciones de peligro. En [4] se muestra un método de crear música adaptativa mediante el uso de músicas *MIDI*.

- Sería recomendable que existiese una interfaz web con un diseño similar a los menús del juego desde la que se pudiesen modificar remotamente los ficheros «config» y «paramConfig.txt» del servidor dedicado.

[3] `http://sandmark.cs.arizona.edu/`

4.4. Valoración personal

El trabajo realizado ha sido muy satisfactorio, ya que me ha permitido cumplir el deseo de elaborar enteramente un videojuego, y además me ha aportado muchos conocimientos íntimamente ligados a dicho ámbito, así como muchos otros que seguro me son de gran utilidad en el ejercicio de mi carrera profesional.

Durante la elaboración de este Proyecto Fin de Carrera me encontré con diversas dificultades que me supusieron un empleo de tiempo mayor de lo esperado. Las más importantes fueron:

> (1) la comprobación de que la implementación de VESPA desarrollada funcionaba de forma correcta,
>
> (2) la adaptación del cálculo de la *Encounter Probability* (EP) de VESPA a partir del simulador, poco documentado, enteramente en francés, y con varios fallos (que costó encontrar) ya que no se trataba de la versión final, y la más importante,
>
> (3) el empleo de mucho tiempo de análisis, diseño e implementación de aspectos y características que en siguientes iteraciones se terminaron descartando, como por ejemplo buscar la forma de que el sonido del motor del coche fuera dinámico (con cambios de las marchas) o tratar el problema de los edificios que invadían la calzada y que dificultaban los algoritmos de la *IA*).

A estas dificultades habría que añadir la excesiva dilatación en el tiempo de la realización del proyecto, y su amplitud, que en ocasiones hacía difícil mantener la visión del conjunto, a pesar de la documentación desarrollada.

Debido a estas dificultades y a la cantidad de errores iniciales a causa de la poca documentación existente acerca de algunos temas, en los que fui aprendiendo a base de errores, el proyecto se dilató excesivamente en el tiempo y hubo algunos momentos en los que me planteé si la elección del proyecto había sido acertada, pero la motivación que me suponía realizar un videojuego y el apoyo de mis tutores me permitió sobrellevar esos momentos de desánimo.

A pesar de esto, considero muy útil toda mi experiencia en la realización del proyecto, tanto por lo aprendido como por lo trabajado, y me ha supuesto una gran satisfacción personal ver la evolución del desarrollo del videojuego hasta lo que es ahora.

Bibliografía

[1] J. E. Michael Behrisch, Laura Bieker, and D. Krajzewicz. SUMO – Simulation of Urban MObility: An overview. In *The Third International Conference on Advances in System Simulation (SIMUL'11)*, pages 63–68. IARIA, 2011.

[2] Yahn W. Bernier. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. In *Game Developers Conference*, 2001. Available at `http://web.cs.wpi.edu/~claypool/courses/4513-B03/papers/games/bernier.pdf`.

[3] Benedikt Bitterli. A Verlet based approach for 2D game physics. `http://www.gamedev.net/page/resources/_/technical/math-and-physics/a-verlet-based-approach-for-2d-game-physics-r2714`. Last accessed August 23, 2013.

[4] David Brackeen, Bret Barker, and Lawrence Vanhelsuwe. *Developing Games in Java*. New Riders Publishing, 2003.

[5] N. Cenerario, T. Delot, and S. Ilarri. A Content-Based Dissemination Protocol for VANETs: Exploiting the Encounter Probability. *IEEE Transactions on Intelligent Transportation Systems*, 12(3):771–782, 2011.

[6] T. Delot and S. Ilarri. Data gathering in vehicular networks: The VESPA experience (invited paper). In *Fifth IEEE Workshop On User MObility and VEhicular Networks (LCN ON-MOVE 2011)*, pages 801–808. IEEE Computer Society, 2011.

[7] T. Delot, S. Ilarri, S. Lecomte, and N. Cenerario. Sharing with caution: Managing parking spaces in vehicular networks. *Mobile Information Systems*, 9(1):69–98, 2013.

[8] S. Esper, S. R. Foster, and W. G. Griswold. On the nature of fires and how to spark them when you're not there. In *44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*, pages 305–310. ACM, 2013.

[9] Glenn Fiedler. UDP vs. TCP. http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/. Last accessed August 23, 2013.

[10] Glenn Fiedler. What every programmer needs to know about game networking. http://gafferongames.com/networking-for-game-programmers/what-every-programmer-needs-to-know-about-game-networking/. Last accessed August 23, 2013.

[11] J. Harri, F. Filali, C. Bonnet, and M. Fiore. VanetMobiSim: Generating realistic mobility patterns for VANETs. In *Third International Workshop on Vehicular Ad Hoc Networks (VANET'06)*, pages 96–97. ADM, 2006.

[12] Brian Hook. Introduction to Multiplayer Game Programming. http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/IntroductionToMultiplayerGameProgramming. Last accessed August 23, 2013.

[13] Brian Hook. The Quake3 networking model. http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking. Last accessed August 23, 2013.

[14] M. Lehn, C. Leng, R. Rehner, T. Triebel, and A. Buchmann. An online gaming testbed for peer-to-peer architectures. *ACM SIGCOMM Computer Communication Review*, 41(4):474–475, 2011.

[15] R. Mangharam, D. S. Weller, and R. Rajkumar. GrooveNet: A hybrid simulator for vehicle-to-vehicle networks. In *Second International Workshop Vehicle-to-VehicleCommunications (V2VCOM'06)*, pages 1–8, 2006.

[16] M. Piorkowski, M. Raya, A. L. Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. TraNS: Realistic joint traffic and network simulator for VANETs. *SIGMOBILE Mobile Computing and Communications Review*, 12(1):31–33, 2008.

[17] Craig Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, pages 763–782, 1999.

[18] Fabien Sanglard. Quake Engine code review. http://fabiensanglard.net/quakeSource/quakeSourcePrediction.php. Last accessed August 23, 2013.

[19] Hidenari Sawashima. Characteristics of UDP Packet Loss: Effect of TCP Traffic. http://www.isoc.org/INET97/proceedings/F3/F3_1.HTM. Last accessed August 23, 2013.

[20] C. Sommer, R. German, and F. Dressler. Bidirectionally coupled network and road traffic simulation for improved IVC analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, 2011.

[21] Nguonly Taing. TCP UDP and RMI Performance Evaluation. `http://lycog.com/performance-evaluation/tcp-udp-rmi-performance-evaluation/`. Last accessed August 23, 2013.

[22] VALVE. Source Multiplayer Networking. `https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking`. Last accessed August 23, 2013.

[23] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'04)*, pages 319–326. ACM, 2004.

[24] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[25] T. Yan, M. Marzilli, R. Holmes, D. Ganesan, and M. Corner. mCrowd: A platform for mobile crowdsourcing. In *Seventh ACM Conference on Embedded Networked Sensor Systems (SenSys'09)*, pages 347–348. ACM, 2009.

Anexos

Anexo A

Análisis

Para la elaboración de este Proyecto Fin de Carrera se ha seguido una metodología de desarrollo basada en diferentes iteraciones del producto, de forma que en cada reunión se establecían los objetivos del siguiente prototipo. Debido a que se realizaron muchas iteraciones con relativamente pocos cambios entre ellas, en este anexo se muestran únicamente los diferentes aspectos del análisis correspondiente a la última iteración (salvo que se indique lo contrario).

A.1. Requisitos

A continuación se muestran los requisitos de la última iteración del juego.

1. Generales

R1.1. La aplicación se tratará de un juego de coches

R1.2. Podrán jugar varios usuarios en una misma partida a través de la red

R1.3. Se elaborarán diversos modos de juego (pruebas de carácter competitivo)

R1.4. Se elaborará una IA para el control de los vehículos no humanos

2. Jugabilidad

R2.1. Se tomará como base para el juego el clásico videojuego Rally-X (Namco 1980)[1]

R2.1.1. Existirán vehículos enemigos que nos persigan

[1] `http://en.wikipedia.org/wiki/Rally-X`

R2.1.2. Existirán banderas que haya que recolectar

R2.1.3. Los jugadores podrán hacer uso de «bombas de humo» que despistarán a los enemigos

R2.1.4. Los jugadores tendrán una cantidad de combustible limitada que se recargará conforme se avance de nivel

R2.1.5. Los jugadores tendrán una cantidad de salud limitada que se recargará conforme se avance de nivel

R2.1.6. El juego dispondrá de diversos niveles en los que se irá avanzando hasta ser eliminado

R2.2. Se añadirán elementos para demostrar las ventajas del uso de *VANETs*

R2.2.1. Existirán otros vehículos no humanos cumpliendo la función de tráfico

R2.2.2. Existirán plazas de aparcamiento en las cuales podrán aparcar tanto los vehículos del tráfico como los jugadores

R2.2.3. Existirán vehículos de servicios de emergencia

R2.3. El jugador podrá abandonar el vehículo y avanzar andando

R2.4. El jugador, mientras esté muerto, podrá mover la cámara libremente por el escenario o ver lo que hacen los otros jugadores, con el objetivo de amenizar la espera hasta que sea revivido

3. Escenarios

R3.1. Se podrá jugar en diferentes escenarios reales

R3.2. Los escenarios se obtendrán mediante un servicio de mapas online

R3.2.1. Los escenarios se obtendrán a través del servicio *OpenStreetMap*

R3.3. Se podrán previsualizar los escenarios descargados

R3.4. Se almacenarán y consultarán localmente los mapas y las imágenes de previsualización de los mismos

4. Interfaz

R4.1. Todos los textos del juego se elaborarán en inglés

R4.1. La navegación por los menús de la aplicación se realizará mediante una interfaz gráfica

R4.2. El usuario podrá crear una partida nueva

R4.3. El usuario podrá unirse a una partida en red

R4.4. El usuario podrá gestionar los mapas almacenados: añadir, previsualizar y eliminar

R4.5. El juego dispondrá de música tanto durante la navegación por los menús como durante la partida

R4.6. El usuario podrá subir y bajar el volumen de música, así como también desconectarla

R4.7. Se creará una pantalla en la que se indiquen datos sobre el autor, los directores del proyecto y se agradezcan los usos de librerías y melodías utilizadas.

R4.8. Existirán varias configuraciones de dificultad cerradas: alta, media y baja

R4.9. Se notificará a los demás jugadores cuando un jugador se haya desconectado

R4.10. Se mostrará una barra de progreso durante el proceso de carga de la partida

5. *VANETs*

R5.1. Se permitirá la integración en el juego de un sistema de gestión de datos (*VANET*)

R5.2. En concreto, se integrará el sistema VESPA

R5.3. Se permitirán simular situaciones reales en las que se puedan evaluar el efecto que podría tener la utilización de un sistema de gestión de datos

R5.4. Se definirán los «interfaces» Java básicos (o clases abstractas) necesarias para desde el videojuego poder interaccionar con VESPA

R5.5. Se implementará una instanciación de dichos interfaces para poder probar VESPA en el juego

6. Entorno

R6.1. La aplicación se realizará sobre Java

R6.2. La aplicación podrá ejecutarse como aplicación de escritorio y también como *Applet*

R6.3. La aplicación debe funcionar en Windows XP, Linux y Mac OS X

7. Técnicas

R7.1. Se usará el protocolo UDP para la comunicación habitual entre el cliente y el servidor

R7.2. El cliente y el servidor estarán acoplados en una sola entidad, de forma que el jugador solo tenga que abrir una instancia para poder jugar

R7.3. La aplicación será multi-hilo

8. Otros

R8.1. El comportamiento de los vehículos no humanos, la IA del juego, deberá estar completamente aislado de todo lo demas, de forma que se pueda cambiar el comportamiento incluso sin reprogramar nada o muy poco (depende del cambio)

R8.2. El usuario se podrá unir a la partida sobre la marcha, durante el trascurso de una partida

A.2. Casos de uso

En esta sección se mostrarán los diagramas de casos de usos analizados.

El análisis se ha dividido entre, por un lado, la navegación por los menús hasta iniciar la partida (figura A.1) y por otro lado la partida en sí.
Además, el análisis de la partida se ha separado en diferentes diagramas: uno general (figura A.2), en el que se han simplificado todas las acciones del jugador y otros en los que se detallan dichas acciones según el modo de juego escogido.

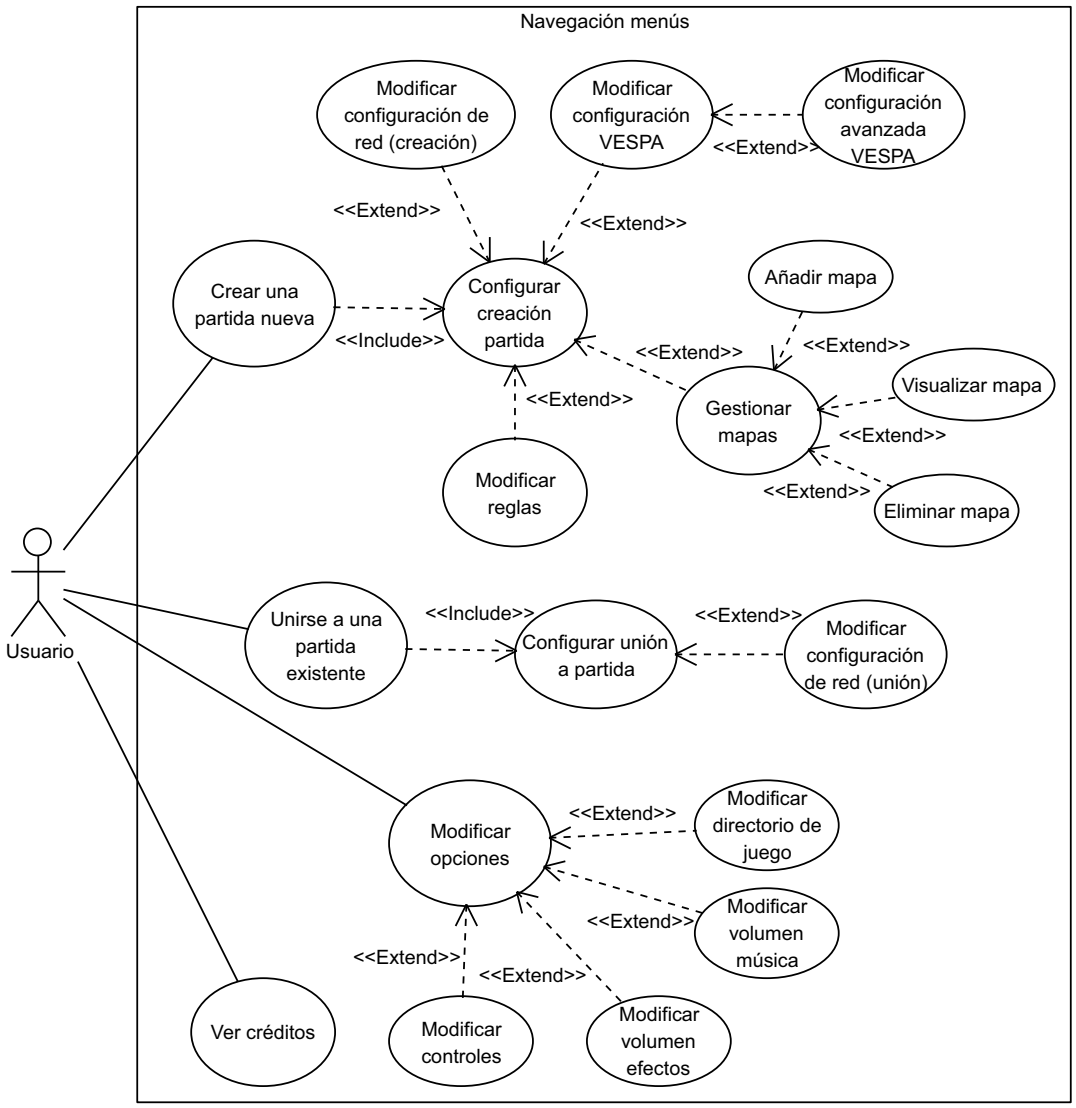


Figura A.1: Casos de uso: navegación menús

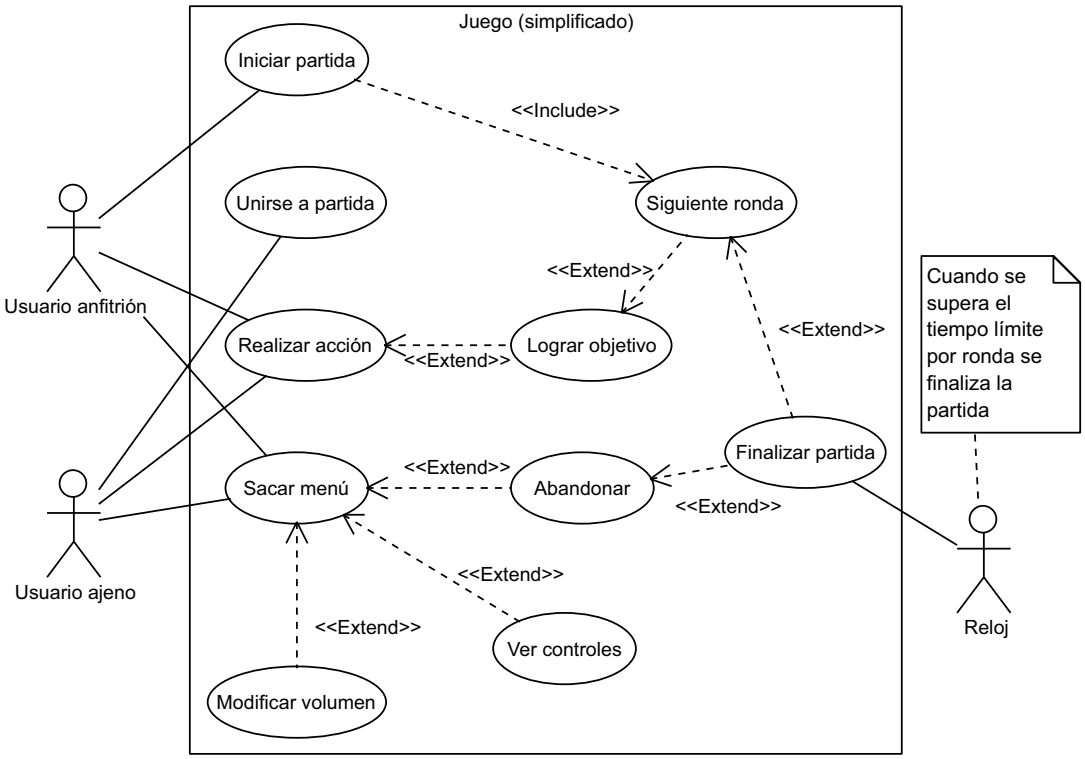


Figura A.2: Casos de uso: partida

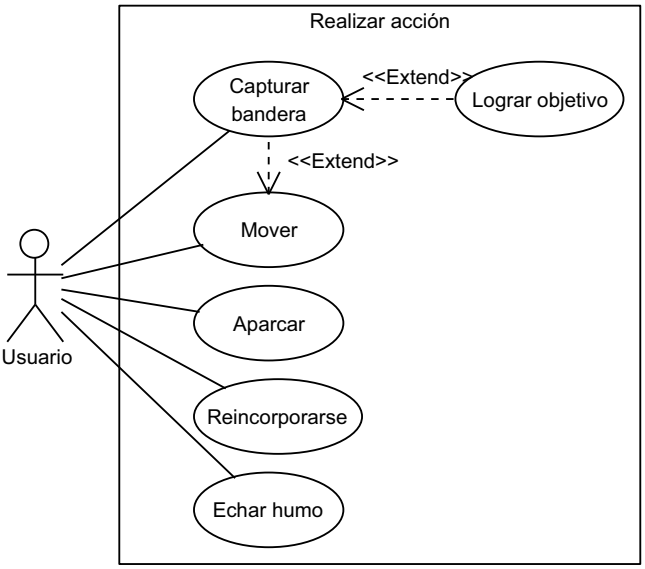


Figura A.3: Casos de uso: detalle del modo de juego «capture the flags»

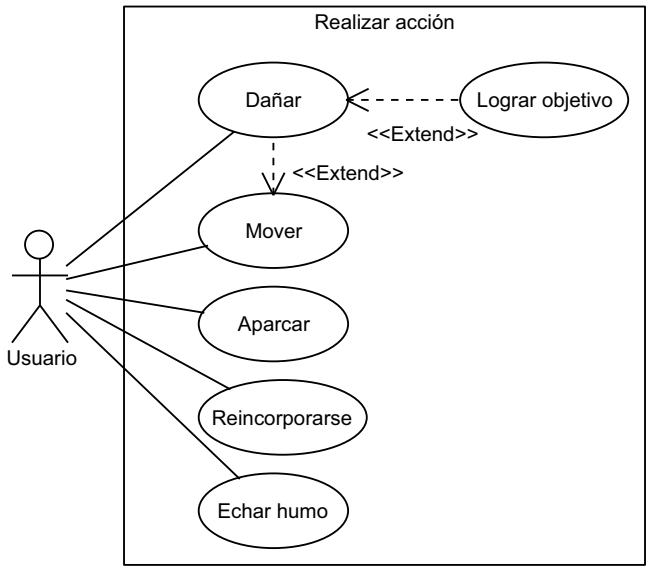


Figura A.4: Casos de uso: detalle del modo de juego «capture the red cars»

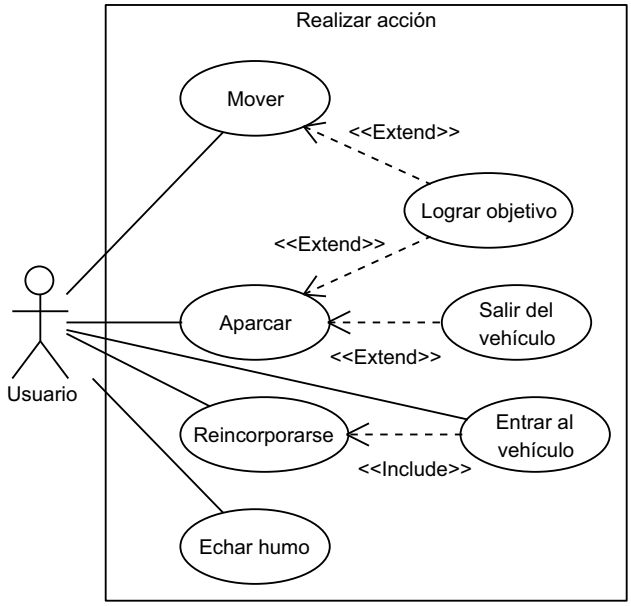


Figura A.5: Casos de uso: detalle del modo de juego «solve the task» y «task endurance survival»

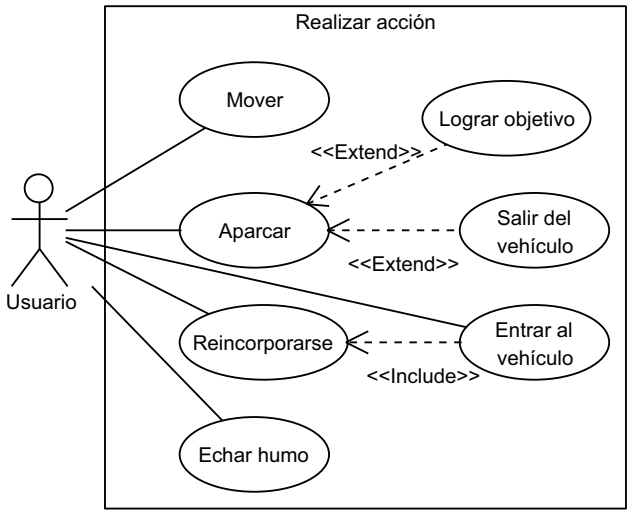


Figura A.6: Casos de uso: detalle del modo de juego «parking special mode»

Menús

Nombre:	**Ver créditos**
Descripción:	El usuario visualiza la pantalla que contiene información sobre el juego, el autor y sobre VESPA
Precondiciones:	El usuario se encuentra en la pantalla «inicial» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «inicial» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla «créditos» 2. El sistema muestra la pantalla «créditos» al usuario 3. El usuario demanda volver a la pantalla anterior

Nombre:	**Modificar opciones**
Descripción:	El usuario visualiza la pantalla que permite cambiar la configuración del volumen, controles y directorio de juego.
Precondiciones:	El usuario se encuentra en la pantalla «inicial» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «inicial» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla «créditos» 2. El sistema muestra la pantalla «opciones» al usuario 3. El usuario realiza las modificaciones deseadas a la configuración 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	3a. El usuario desea modificar el volumen de la música 1. Caso de uso «Modificar volumen música» 3b. El usuario desea modificar el volumen de los efectos 1. Caso de uso «Modificar volumen efectos» 3c. El usuario desea modificar el directorio de juego 1. Caso de uso «Modificar volumen efectos» 3d. El usuario desea modificar los controles 1. Caso de uso «Modificar controles» 4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados

Nombre:	**Modificar controles**
Descripción:	El usuario visualiza la pantalla que permite visualizar y modificar los controles del juego.
Precondiciones:	El usuario se encuentra en la pantalla «opciones» del menú

−continúa en la siguiente página−

–continúa de la página anterior–

Postcondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla «controles» 2. El sistema muestra la pantalla «controles» al usuario 3. El usuario realiza las modificaciones deseadas a la configuración 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados *a. El usuario desea restaurar los valores por defecto 1. Todos los campos de la pantalla se restablecen a sus valores por defecto.

Nombre:	**Modificar volumen música**
Descripción:	El usuario modifica el volumen de la música del juego.
Precondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Flujo normal:	1. El usuario realiza las modificaciones deseadas a la configuración

Nombre:	**Modificar volumen efectos**
Descripción:	El usuario modifica el volumen de los efectos del juego.
Precondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Flujo normal:	1. El usuario realiza las modificaciones deseadas a la configuración

Nombre:	**Modificar directorio de juego**
Descripción:	El usuario elige en qué carpeta se guardarán los datos del juego.
Precondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «opciones» del menú
Flujo normal:	1. El usuario demanda la carga de la ventana emergente de selección de archivos 2. El sistema muestra la pantalla emergente de selección de archivos al usuario

–continúa en la siguiente página–

–continúa de la página anterior–

	3. El usuario elige la carpeta deseada 4. El usuario demanda volver a la pantalla anterior aceptando los cambios
Flujo alternativo:	4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior cancelando los cambios realizados

Nombre:	**Unirse a una partida existente**
Descripción:	El usuario se une a una partida en red en curso
Precondiciones:	El usuario se encuentra en la pantalla «inicial» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla «unirse» 2. El sistema muestra la pantalla «unirse» al usuario 3. Caso de uso «Configurar unión a partida» 4. El usuario demanda unirse a la partida seleccionada 5. El sistema muestra al usuario la pantalla en la que aparece información de la partida en curso y sus jugadores. 6. El usuario selecciona a qué equipo unirse. 7. El usuario demanda incorporarse a la partida. 8. El sistema cierra el sistema de menús e incorpora al jugador a la partida en curso.
Flujo alternativo:	3a, 4a, 6a, 7a. El usuario desea volver a la pantalla anterior 1. El usuario demanda volver a la pantalla anterior cancelando los cambios realizados 5a, 8a. La partida no existe o está completa o no se tiene suficiente pericia para unirte 1. El sistema muestra al usuario una pantalla de error indicándole los motivos del mismo 2. El usuario vuelve a la pantalla «inicial» del menú

Nombre:	**Configurar unión a partida**
Descripción:	El usuario configura los parámetros básicos de unión a la partida: IP, apodo, activación del DMS
Precondiciones:	El usuario se encuentra en la pantalla «unirse» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «unirse» del menú
Flujo normal:	1. El usuario modifica los parámetros deseados
Flujo alternativo:	1a. El usuario desea volver a la pantalla anterior 1. El usuario demanda volver a la pantalla anterior

–continúa en la siguiente página–

–continúa de la página anterior–

	cancelando los cambios realizados 1b. El usuario desea modificar los parámetros avanzados: configuración de red 1. Caso de uso «Modificar configuración de red (unión)»

Nombre:	**Modificar configuración de red (unión)**
Descripción:	El usuario configura los parámetros avanzados de unión a la partida: puerto del anfitrión y puerto local.
Precondiciones:	El usuario se encuentra en la pantalla «unirse» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «unirse» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla de configuración avanzada de red (en unión) 2. El sistema muestra dicha pantalla al usuario. 3. El usuario modifica los parámetros deseados 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados *a. El usuario desea restaurar los valores por defecto 1. Todos los campos de la pantalla se restablecen a sus valores por defecto.

Nombre:	**Crear una partida nueva**
Descripción:	El usuario se inicia una nueva partida
Precondiciones:	El usuario se encuentra en la pantalla «inicial» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla «iniciar» 2. El sistema muestra la pantalla «iniciar» al usuario 3. Caso de uso «Configurar creación partida» 4. El usuario demanda comenzar la partida 5. El sistema cierra el sistema de menús e incorpora al jugador a la partida que se crea.
Flujo alternativo:	3a, 4a. El usuario desea volver a la pantalla anterior 1. El usuario demanda volver a la pantalla anterior cancelando los cambios realizados

–continúa en la siguiente página–

–continúa de la página anterior–

	5a. La configuración seleccionada no es compatible o el apodo no es válido o no se ha seleccionado ningún mapa o el usuario no tiene pericia suficiente para crear este tipo de partida 1. El sistema muestra al usuario una ventana emergente de error indicándole los motivos del mismo 2. Se continúa en el paso 3 del flujo normal.

Nombre:	**Configurar creación partida**
Descripción:	El usuario configura los parámetros básicos de creación de la partida: apodo, activación del DMS, mapa y modo de juego.
Precondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Flujo normal:	1. El usuario modifica los parámetros deseados
Flujo alternativo:	1a. El usuario desea volver a la pantalla anterior 1. El usuario demanda volver a la pantalla anterior cancelando los cambios realizados 1b. El usuario desea modificar la configuración de red 1. Caso de uso «Modificar configuración de red (creación)» 1c. El usuario desea modificar la configuración del sistema VESPA 1. Caso de uso «Modificar configuración VESPA» 1d. El usuario desea modificar la configuración de las reglas del juego 1. Caso de uso «Modificar reglas» 1e. El usuario desea gestionar los mapas 1. Caso de uso «Gestionar mapas»

Nombre:	**Modificar configuración de red (creación)**
Descripción:	El usuario configura los parámetros de red de creación de la partida: puerto del anfitrión, puerto local y adaptador de red deseado.
Precondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú

–continúa en la siguiente página–

–continúa de la página anterior–

Flujo normal:	1. El usuario demanda la carga de la pantalla de configuración avanzada de red (en creación) 2. El sistema muestra dicha pantalla al usuario 3. El usuario modifica los parámetros deseados 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados *a. El usuario desea restaurar los valores por defecto 1. Todos los campos de la pantalla se restablecen a sus valores por defecto.

Nombre:	**Modificar configuración VESPA**
Descripción:	El usuario configura los parámetros del sistema VESPA.
Precondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla de configuración de VESPA 2. El sistema muestra dicha pantalla al usuario 3. El usuario modifica los parámetros deseados 4. Caso de uso «Modificar configuración avanzada VESPA» 5. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	5a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados *a. El usuario desea restaurar los valores por defecto 1. Todos los campos de la pantalla se restablecen a sus valores por defecto.

Nombre:	**Modificar configuración avanzada VESPA**
Descripción:	El usuario configura los parámetros avanzados del sistema VESPA.
Precondiciones:	El usuario se encuentra en la pantalla de configuración de VESPA

–continúa en la siguiente página–

–continúa de la página anterior–

Postcondiciones:	El usuario se encuentra en la pantalla de configuración de VESPA
Flujo normal:	1. El usuario demanda la carga de la pantalla de configuración avanzada de VESPA 2. El sistema muestra dicha pantalla al usuario 3. El usuario modifica los parámetros deseados 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados *a. El usuario desea restaurar los valores por defecto 1. Todos los campos de la pantalla se restablecen a sus valores por defecto.

Nombre:	**Modificar reglas**
Descripción:	El usuario configura los parámetros de dificultad, número de vehículos, rondas y equipos, y también los tiempos de espera y de límite de duración de la ronda.
Precondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla de configuración de las reglas del juego 2. El sistema muestra dicha pantalla al usuario 3. El usuario modifica los parámetros deseados 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	4a. El usuario no desea guardar los cambios realizados 1. El usuario demanda volver a la pantalla anterior desechando los cambios realizados *a. El usuario desea restaurar los valores por defecto 1. Todos los campos de la pantalla se restablecen a sus valores por defecto.

Nombre:	**Gestionar mapas**
Descripción:	El usuario gestiona los mapas disponibles para la creación de la partida.

–continúa en la siguiente página–

–continúa de la página anterior–

Precondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Postcondiciones:	El usuario se encuentra en la pantalla «iniciar» del menú
Flujo normal:	1. El usuario demanda la carga de la pantalla de gestión de mapas 2. El sistema muestra dicha pantalla al usuario 3. El usuario modifica los parámetros deseados 4. El usuario demanda volver a la pantalla anterior guardando los cambios realizados
Flujo alternativo:	3a. El usuario desea descargar un nuevo mapa 1. Caso de uso «Añadir mapa» 3b. El usuario desea visualizar un mapa descargado 1. Caso de uso «Visualizar mapa» 3c. El usuario desea eliminar un mapa descargado 1. Caso de uso «Eliminar mapa»

Nombre:	**Añadir mapa**
Descripción:	El usuario añade un nuevo mapa al juego.
Precondiciones:	El usuario se encuentra en la pantalla de gestión de mapas
Postcondiciones:	El usuario se encuentra en la pantalla de gestión de mapas
Flujo normal:	1. El usuario introduce las palabras clave de la dirección que desea buscar, un alias y el tamaño deseado y demanda al sistema la realización de la búsqueda 2. El sistema muestra al usuario una lista con todas las coincidencias de la búsqueda 3. El usuario elige el mapa deseado de dicha lista. 4. El usuario visualiza y/o descarga el mapa
Flujo alternativo:	2a. No existe ninguna coincidencia 1. Se le informa al usuario de ello. 4a. Se desea visualizar el mapa 1. El usuario demanda visualizar el mapa a descargar 2. El sistema muestra una vista previa de dicho mapa al usuario 4b. Se desea descargar el mapa 1. El usuario demanda la adición del mapa al juego 2a. El sistema descarga dicho mapa y lo añade a la lista de mapas disponibles. 2b. El alias ya existe 1. Se informa al jugador resaltando el campo

–continúa en la siguiente página–

–continúa de la página anterior–

	2c. La dirección elegida ya existe con ese mismo tamaño 1. Se le informa al usuario mediante una ventana emergente de error 1a, 3a, 4c. El usuario no desea realizar cambios 1. El usuario demanda volver a la pantalla anterior

Nombre:	**Visualizar mapa**
Descripción:	El usuario visualiza un mapa ya descargado
Precondiciones:	El usuario se encuentra en la pantalla de gestión de mapas
Postcondiciones:	El usuario se encuentra en la pantalla de gestión de mapas
Flujo normal:	1. El usuario selecciona de la lista el mapa que desea visualizar 2. El sistema muestra una vista previa de dicho mapa al usuario

Nombre:	**Eliminar mapa**
Descripción:	El usuario elimina un mapa ya descargado
Precondiciones:	El usuario se encuentra en la pantalla de gestión de mapas
Postcondiciones:	El usuario se encuentra en la pantalla de gestión de mapas
Flujo normal:	1. El usuario selecciona de la lista el mapa que desea eliminar 2. El sistema muestra una ventana emergente pidiendo la confirmación del usuario 3. El usuario aprueba la eliminación 4. El sistema elimina dicho mapa de los ficheros de datos del juego y de la tabla.
Flujo alternativo:	3a. El usuario cancela la eliminación 1. El usuario demanda la cancelación de la operación

Partida (simplificado)

Nombre:	**Iniciar partida**
Descripción:	Se inicia una nueva partida y el jugador comienza a jugar
Actores:	Usuario anfitrión
Precondiciones:	Menú en pantalla «iniciar» con todos los parámetros ya configurados de forma correcta

–continúa en la siguiente página–

–continúa de la página anterior–

Postcondiciones:	El sistema cierra el sistema de menús e incorpora al jugador a la partida que se crea.
Flujo normal:	1. Caso de uso «Crear una partida nueva» 2. Caso de uso «Siguiente ronda»

Nombre:	**Unirse a partida**
Descripción:	El jugador se une a una partida en red ya existente
Actores:	Usuario ajeno
Precondiciones:	Menú en pantalla «unirse» con todos los parámetros ya configurados de forma correcta
Postcondiciones:	El sistema cierra el sistema de menús e incorpora al jugador a la partida deseada.
Flujo normal:	1. Caso de uso «Unirse a una partida existente»

Nombre:	**Sacar menú**
Descripción:	Muestra en pantalla el menú de pausa
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	El usuario está jugando una partida
Flujo normal:	1. El usuario presiona la tecla encargada de hacer aparecer el menú 2. El sistema muestra dicho menú. 3. El usuario realiza las operaciones deseadas 4. El usuario vuelve a la partida pulsando de nuevo la tecla establecida o mediante la pulsación de la opción del menú.
Flujo alternativo:	3a. La operación deseada es modificar el volumen 1. Caso de uso «Modificar volumen» 3b. La operación deseada es ver los controles 1. Caso de uso «Ver controles» 3c. La operación deseada es abandonar la partida 1. Caso de uso «Abandonar»

Nombre:	**Modificar volumen**
Descripción:	Modifica el volumen de los efectos y/o de la música
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	El usuario está con el menú de pausa desplegado
Postcondiciones:	El usuario está con el menú de pausa desplegado

–continúa en la siguiente página–

–continúa de la página anterior–

Flujo normal:	1. El usuario elige la opción «options» del menú 2. El sistema le muestra la pantalla de dicha opción 3. El usuario sube o baja los niveles del volumen de los efectos y de la música mediante los botones habilitados 4. El usuario acepta los cambios 5. El sistema muestra la pantalla inicial del menú

Nombre:	**Ver controles**
Descripción:	Muestra al usuario las teclas asociadas con los diferentes controles
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	El usuario está con el menú de pausa desplegado
Postcondiciones:	El usuario está con el menú de pausa desplegado
Flujo normal:	1. El usuario elige la opción «show controls» del menú 2. El sistema le muestra la pantalla de dicha opción 4. El usuario acepta volver a la pantalla anterior 5. El sistema muestra la pantalla inicial del menú

Nombre:	**Abandonar**
Descripción:	El usuario abandona la partida actual
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	El usuario está con el menú de pausa desplegado
Postcondiciones:	El usuario está con el menú de pausa desplegado
Flujo normal:	1. El usuario elige la opción «end game» del menú 2. El sistema le muestra un mensaje de confirmación 3. El usuario acepta abandonar la partida 4. El sistema abandona la partida y se carga de nuevo el sistema de menús mostrándose la pantalla de resumen de la partida
Flujo alternativo:	3a. El usuario no abandona la partida 1. El usuario elige la opción de continuar jugando 5a. El usuario es el anfitrión de la partida 1. Caso de uso «Finalizar partida»

Nombre:	**Finalizar partida**
Descripción:	El usuario finaliza la partida actual

–continúa en la siguiente página–

–continúa de la página anterior–

Actores:	Usuario anfitrión, reloj
Precondiciones:	El usuario anfitrión acaba de abandonar la partida
Postcondiciones:	El usuario finaliza la partida para todos los jugadores
Flujo normal:	1. El sistema cierra el servidor de la partida, de forma que todos los jugadores presentes vuelven a cargar el sistema de menús y se les muestra la pantalla de resumen de la partida

Nombre:	**Siguiente ronda**
Descripción:	Se avanza a la siguiente ronda de la partida
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso
Postcondiciones:	Se aumenta el nivel de ronda de la partida
Flujo normal:	1. El sistema comprueba cuántas rondas se han superado
Flujo alternativo:	2a. número rondas superadas $\geq$ número límite 1. Caso de uso «Finalizar partida»

Nombre:	**Lograr objetivo**
Descripción:	Un usuario ha logrado un objetivo del modo de juego
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso
Postcondiciones:	Se elimina un objetivo de la lista de objetivos
Flujo normal:	1. Se elimina el objetivo de la lista de objetivos 2. El sistema comprueba cuantos objetivos quedan
Flujo alternativo:	2a. No quedan objetivos 1. Caso de uso «Siguiente ronda»

Nombre:	**Realizar acción**
Descripción:	El usuario, mediante un evento de teclado, modifica el estado del actor que le representa en el mundo de juego
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso
Postcondiciones:	La acción seleccionada se ve reflejada en el mundo de juego (salvo pérdida de paquetes de red)
Flujo normal:	1. El usuario pulsa la tecla asignada a la acción que desea realizar

–continúa en la siguiente página–

–continúa de la página anterior–

	2. El sistema refleja dicha acción en el mundo de juego.
Flujo alternativo:	3a. Mediante dicha acción se ha logrado completar un objetivo 1. Caso de uso «Lograr objetivo»

Partida (detalle de «Realizar acción»)

Nombre:	**Echar humo**
Descripción:	El usuario, mediante un evento de teclado, crea una nube de humo situada tras el vehículo que le representa.
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. Cantidad de nubes de humo restantes mayor que cero.
Postcondiciones:	El vehículo que representa al jugador crea una nube de humo tras él.
Flujo normal:	1. El usuario pulsa la tecla asignada a la acción de echar humo. 2. El sistema refleja dicha acción en el mundo de juego.

Nombre:	**Aparcar**
Descripción:	El vehículo representado por el usuario estaciona en una plaza de aparcamiento.
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. Actor del usuario situado sobre una plaza de aparcamiento libre.
Postcondiciones:	El usuario estaciona en la plaza de aparcamiento. La plaza de aparcamiento cambia a estado «ocupada».
Flujo normal:	1. El usuario pulsa la tecla asignada a la acción de aparcar 2. El vehículo del jugador se queda inmóvil 3. La plaza de aparcamiento pasa a estar «ocupada»
Flujo alternativo:	4a. Dicha plaza es uno de los objetivos de la ronda 1. Caso de uso «Lograr objetivo» 5a. Modo de juego «solve the task», «task endurance survival» o «parking special mode» 1. Caso de uso «Salir del vehículo»

Nombre:	**Reincorporarse**

–continúa en la siguiente página–

–continúa de la página anterior–

Descripción:	El vehículo representado por el usuario abandona la plaza de aparcamiento en la que se encuentra, reincorporándose a la circulación
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. Actor del usuario estacionado en una plaza de aparcamiento.
Postcondiciones:	El usuario abandona la plaza de aparcamiento en la que se encontraba. La plaza de aparcamiento cambia a estado «libre».
Flujo normal:	1. El usuario pulsa la tecla asignada a la acción de aparcar 2. El vehículo del jugador recupera la movilidad 3. La plaza de aparcamiento en la que el jugador estaba aparcado pasa a estar «libre»

Nombre:	**Salir del vehículo**
Descripción:	El actor que representa al usuario abandona el vehículo y prosigue a pie.
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. Actor del usuario estacionado en una plaza de aparcamiento. Modo de juego «solve the task», «task endurance survival» o «parking special mode».
Postcondiciones:	–El vehículo del usuario continúa en la misma posición estacionado sobre la plaza de aparcamiento. El jugador pasa a controlar un hombre a pie, con distintas características de velocidad, giro, salud, etc.
Flujo normal:	1. El usuario pulsa la tecla asignada a la acción de aparcar 2. El usuario pasa a controlar al conductor del vehículo, que se mueve a pie.

Nombre:	**Entrar al vehículo**
Descripción:	El actor que representa al usuario entra al vehículo
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. Actor del usuario próximo a su vehículo estacionado. Modo de juego «solve the task», «task endurance survival» o «parking special mode».

–continúa en la siguiente página–

–continúa de la página anterior–

Postcondiciones:	El vehículo del usuario continúa en la misma posición estacionado sobre la plaza de aparcamiento. El jugador pasa a controlar un hombre a pie, con distintas características de velocidad, giro, salud, etc.
Flujo normal:	1. El usuario pulsa la tecla asignada a la acción de aparcar 2. El usuario pasa a controlar al vehículo 3. Caso de uso «Reincorporarse»

Nombre:	**Mover**
Descripción:	El usuario realiza un desplazamiento por el mundo de juego, dado por su actual velocidad y ángulo
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. El usuario no está estacionado en una plaza de aparcamiento.
Postcondiciones:	El actor que representa al usuario se habrá desplazado (salvo que haya colisionado con los límites de la calzada)
Flujo normal:	1. El usuario pulsa una de las teclas asignadas al control del vehículo 2. El actor que representa al usuario se desplaza en el mundo de juego
Flujo alternativo:	2a. El resultado del desplazamiento sitúa al actor fuera de los límites de la calzada transitable 1. El actor se desplaza únicamente la cantidad suficiente para no salirse de los límites 3a. El resultado del desplazamiento es una colisión contra otro vehículo al que persigue 1. Caso de uso «Dañar» 3b. El resultado del desplazamiento es una colisión contra otro vehículo al que no persigue y la vida del usuario no es infinita 1. El actor sufre una cantidad establecida de daños. 3c. El resultado del desplazamiento es una colisión con una bandera 1. Caso de uso «Capturar bandera» 3d. El resultado del desplazamiento es coincide con el radio de un lugar objetivo 1. Caso de uso «Lograr objetivo»

Nombre:	**Capturar bandera**
Descripción:	El usuario captura una bandera en el mundo de juego desplazándose sobre ella.
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso
Postcondiciones:	La bandera desaparece y se suma al usuario la puntuación correspondiente
Flujo normal:	1. El actor que representa al usuario colisiona con la bandera 2. El sistema elimina dicha bandera del mundo de juego 3. El sistema incrementa la puntuación del usuario en la cantidad establecida.

Nombre:	**Dañar**
Descripción:	El usuario causa daño a otro vehículo colisionando con él
Actores:	Usuario anfitrión, usuario ajeno
Precondiciones:	Existe una partida en curso. Modo de juego «capture the red cars»
Postcondiciones:	El vehículo contra el que colisiona el usuario recibe una cantidad de daño establecida.
Flujo normal:	1. El actor que representa al usuario colisiona contra otro vehículo 2. El sistema disminuye la cantidad de vida de dicho vehículo
Flujo alternativo:	2a. La vida del otro vehículo es menor que la cantidad a sustraer 1. El sistema elimina al otro vehículo del mundo de juego 2. El sistema incrementa la puntuación del usuario 3a. El otro vehículo es un objetivo de la ronda 1. Caso de uso «Lograr objetivo»

A.3. Diagrama de navegación

En esta sección se muestran el diagrama de pantallas elaborado durante la primera iteración del juego (Figura A.7) y el elaborado durante la última iteración (Figura A.8).
Como se puede apreciar en dichas figuras, se realizaron diversos cambios ya que cada nueva iteración desarrollada a veces requería de nuevas funcionalidades del

menú. Los principales cambios desde la versión inicial a la final fueron:

- Eliminación de las salas de espera, ya que al principio el juego se ideó e implementó de forma que los jugadores solo se podían unir al comienzo de la partida, y éstas salas eran necesarias para que el jugador anfitrión controlase la cantidad de jugadores esperando y el resto de jugadores tuviesen una pantalla en la que esperar al comienzo del juego. En el momento en que se vio que esta forma de unión, basada en los juegos más clásicos (como las sagas *Age of Empires* o *Empire Earth*) no era la óptima para un juego de estas características, se cambió por el método de unión sobre la marcha, utilizado en la gran mayoría de juegos actuales.
- Se incrementó el número de pantallas desde las cuales se podía derivar a una pantalla de error si fuera necesario.
- Se añadieron nuevas pantallas para los créditos y las opciones globales, cuyo contenido estaba incluido inicialmente en otras pantallas menos accesibles, y también una pantalla secreta, a la que solo se puede acceder con el conocimiento de una combinación de teclas específica, para cambiar durante la implementación y las pruebas diversos valores que hasta el momento requerían de una recompilación de todo el código lo cual era muy farragoso.

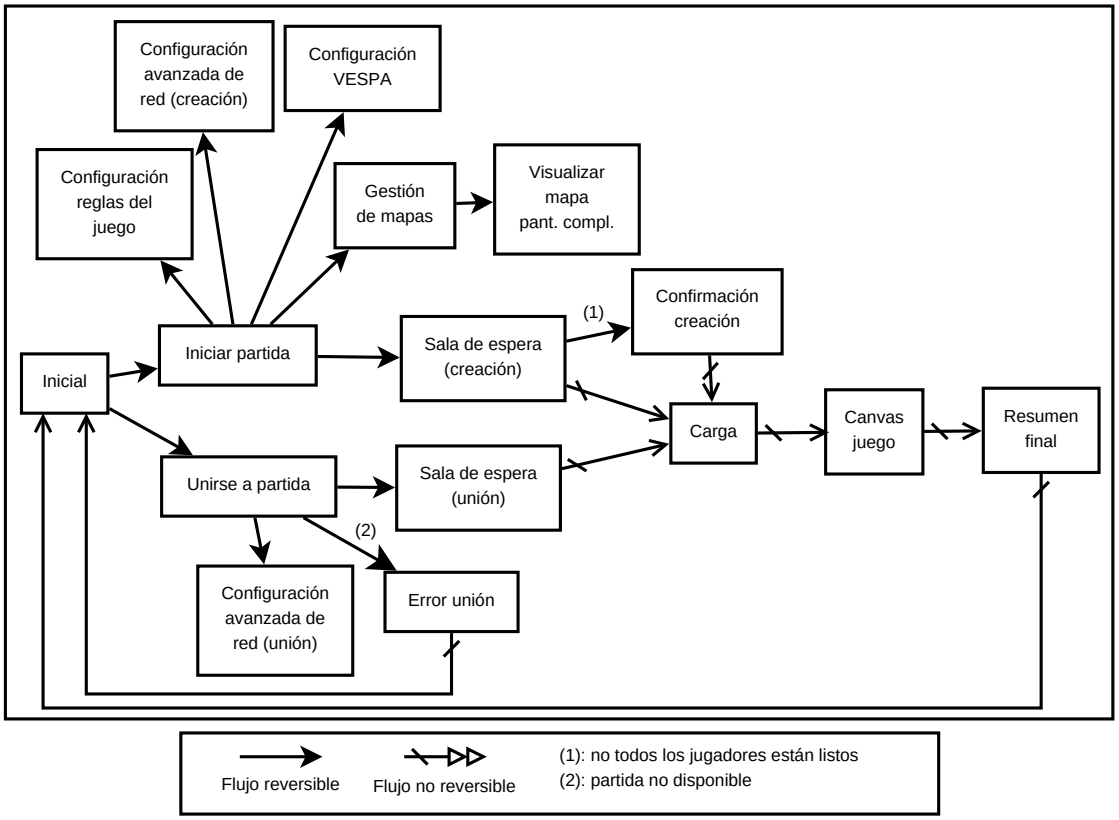


Figura A.7: Diagrama de navegación (primera iteración)

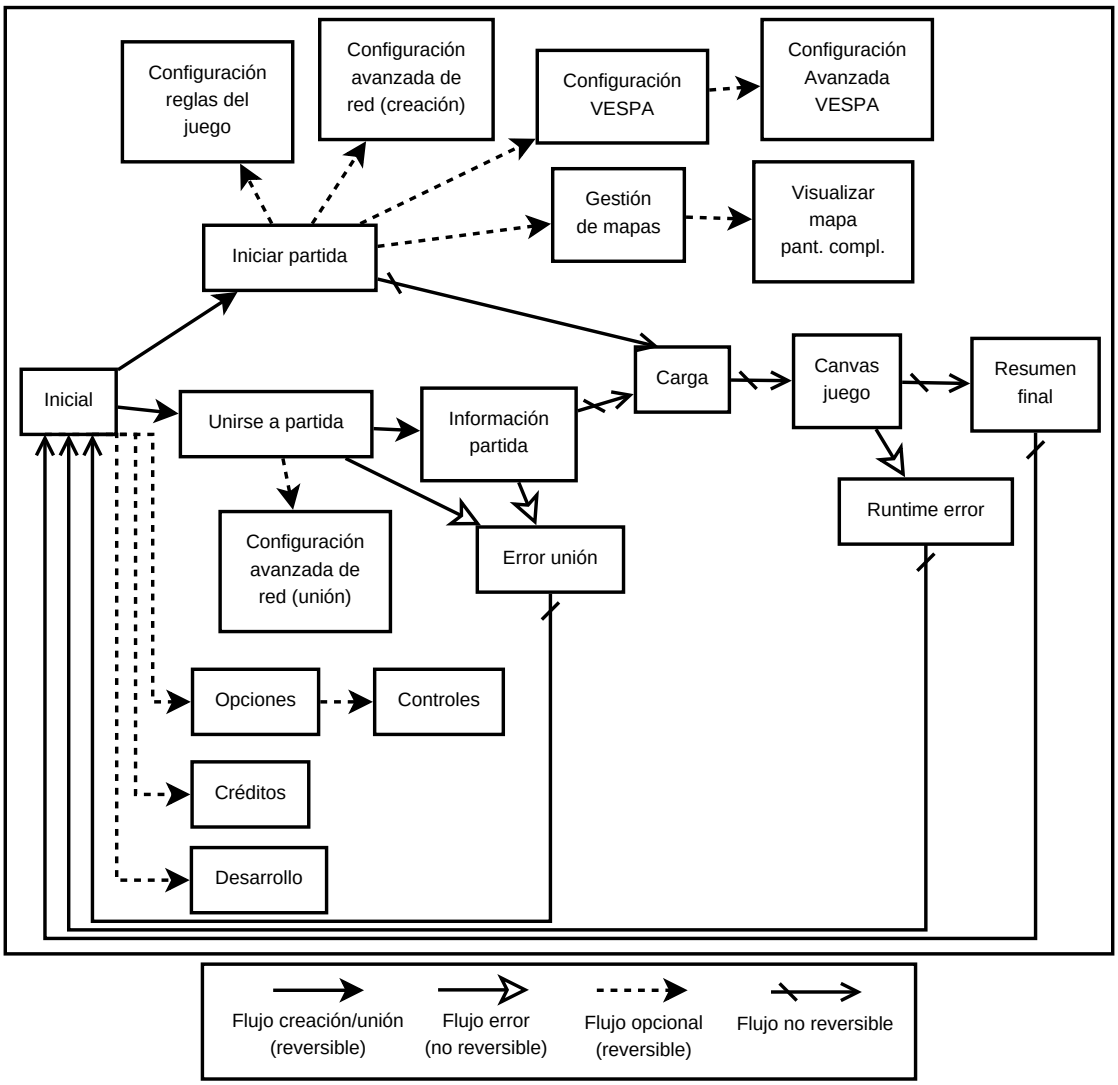


Figura A.8: Diagrama de navegación (última iteración)

A.4. Prototipado de ventanas

En las siguientes figuras se muestran los prototipos de las ventanas de menús realizados durante la primera iteración del juego en la que se consideró el uso de un sistema menús. Para diseñar dichos menús se realizó un análisis de otros videojuegos para reconocer cuales son las características fundamentales de sus menús, obteniéndose dos fundamentales que no suelen estar presentes en otro tipo de aplicaciones: sonido cuando el ratón se sitúa encima de un botón y fondo del menú dinámico.

Nótese que durante el desarrollo se usaron RALLY-X y NetRALLY-X como títulos provisionales del juego.

Figura A.9: Prototipo de ventana inicial

Figura A.10: Prototipo de ventana de unión a partida existente

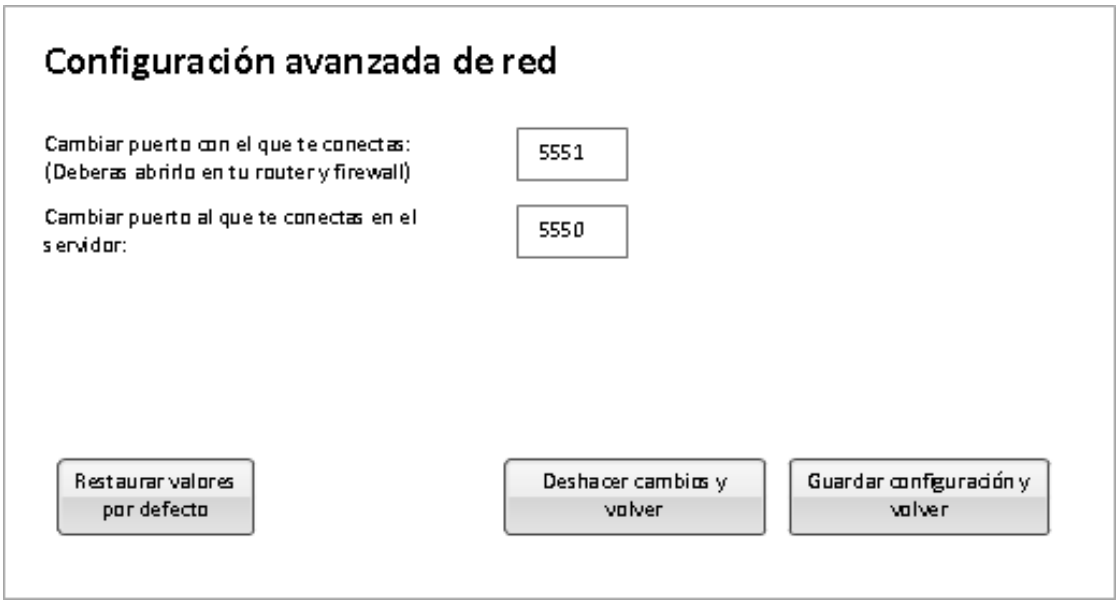


Figura A.11: Prototipo de ventana modificación configuración de red (en unión)

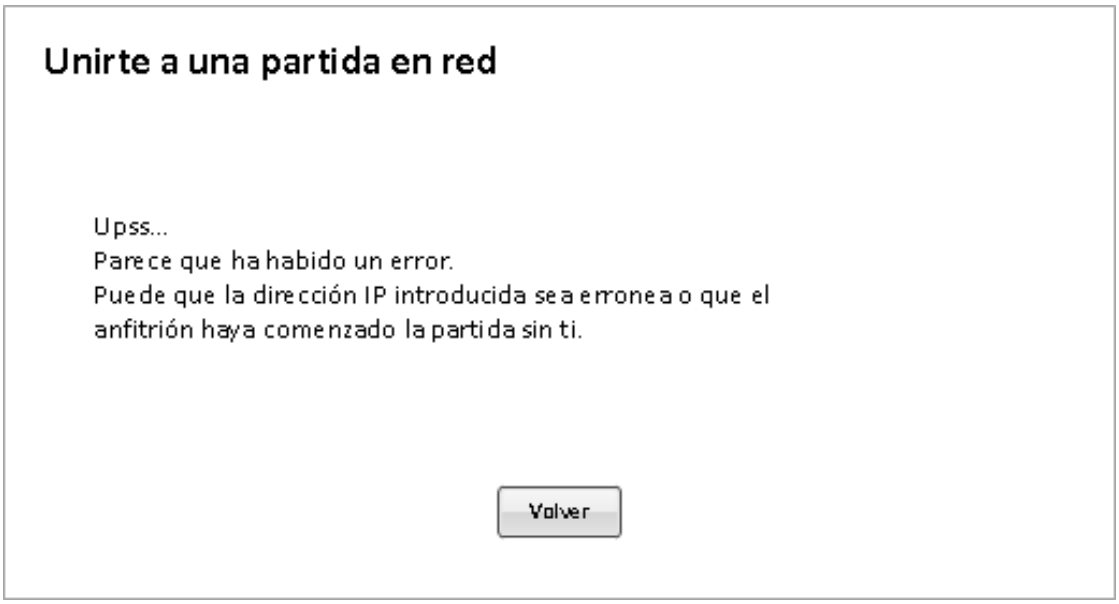


Figura A.12: Prototipo de ventana de error conectando a partida

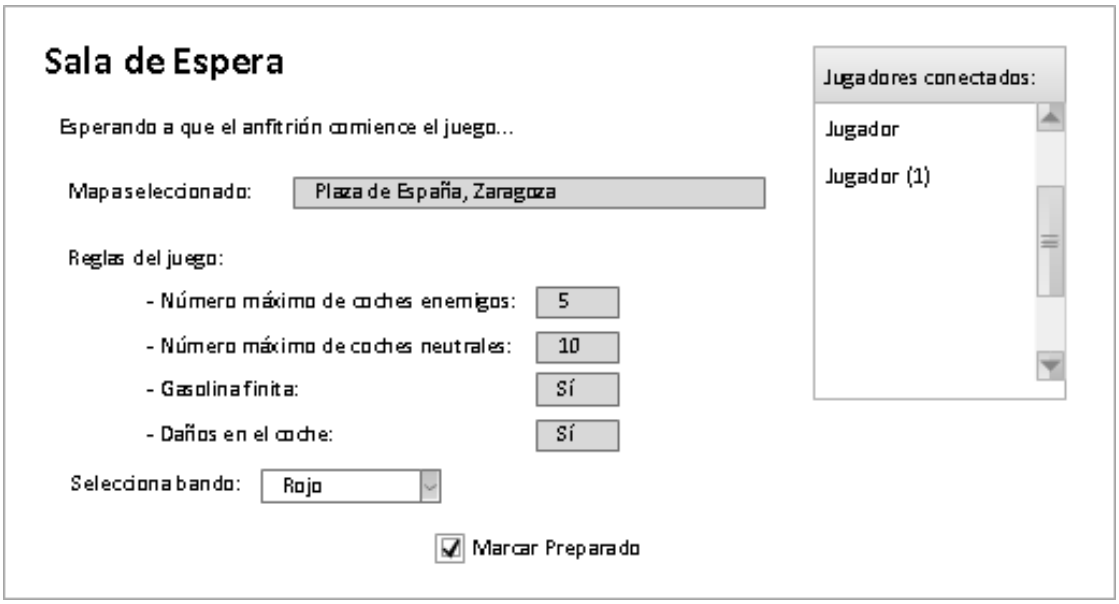


Figura A.13: Prototipo de ventana de sala de espera (en unión)

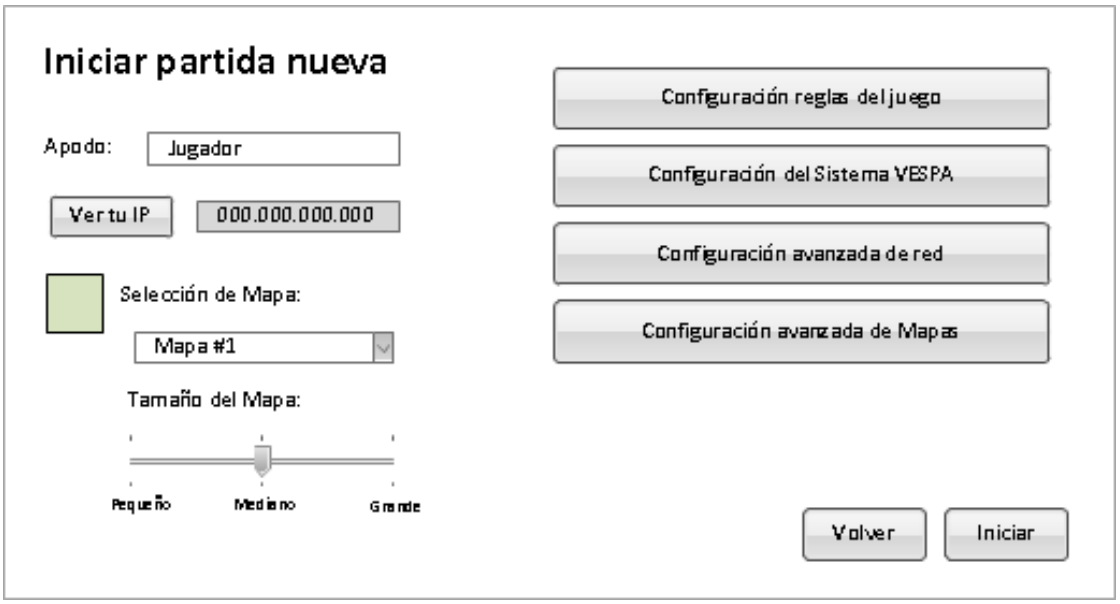


Figura A.14: Prototipo de ventana de crear una nueva partida

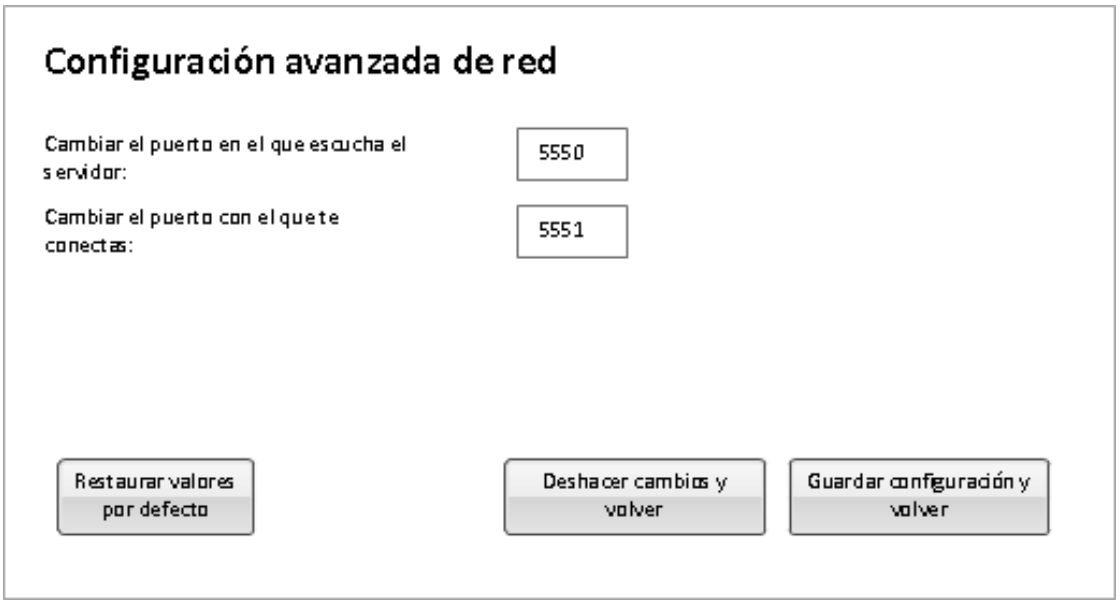


Figura A.15: Prototipo de ventana modificar configuración red (en creación)

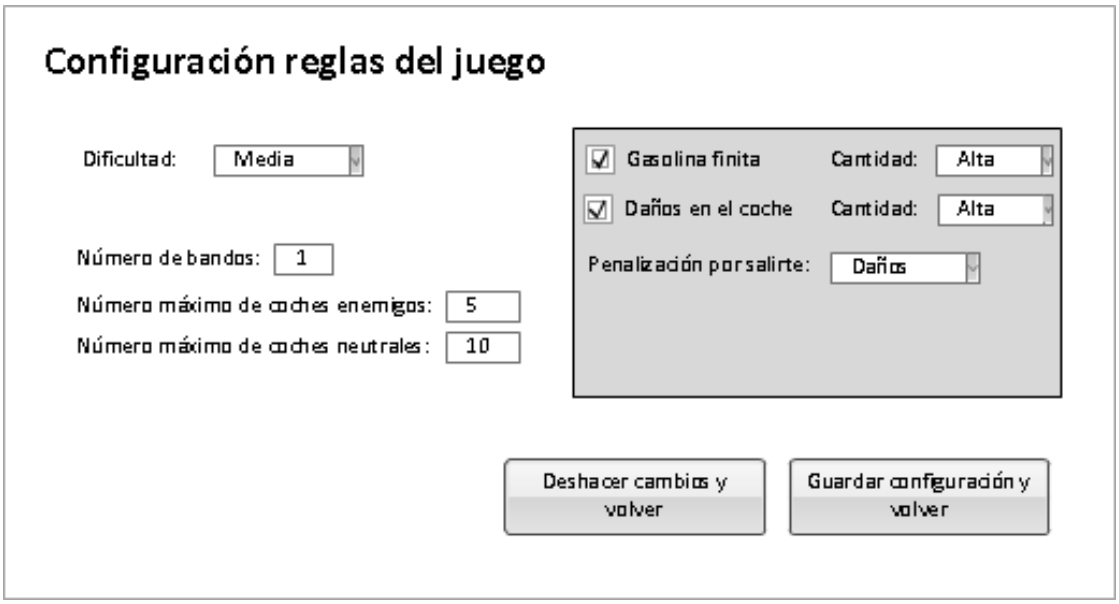


Figura A.16: Prototipo de ventana de reglas del juego

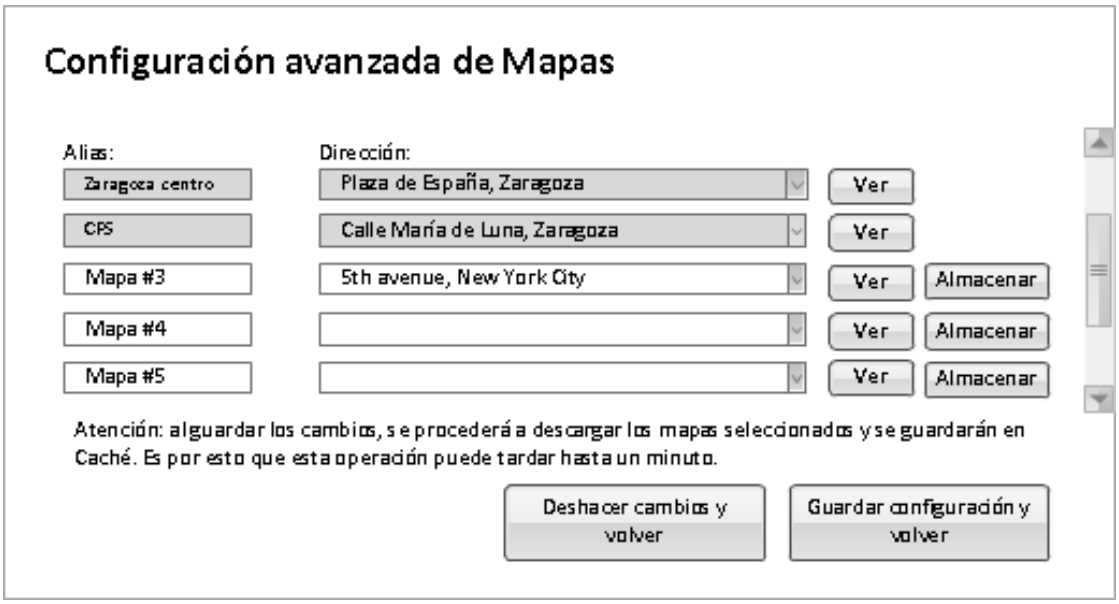


Figura A.17: Prototipo de ventana de gestión de mapas

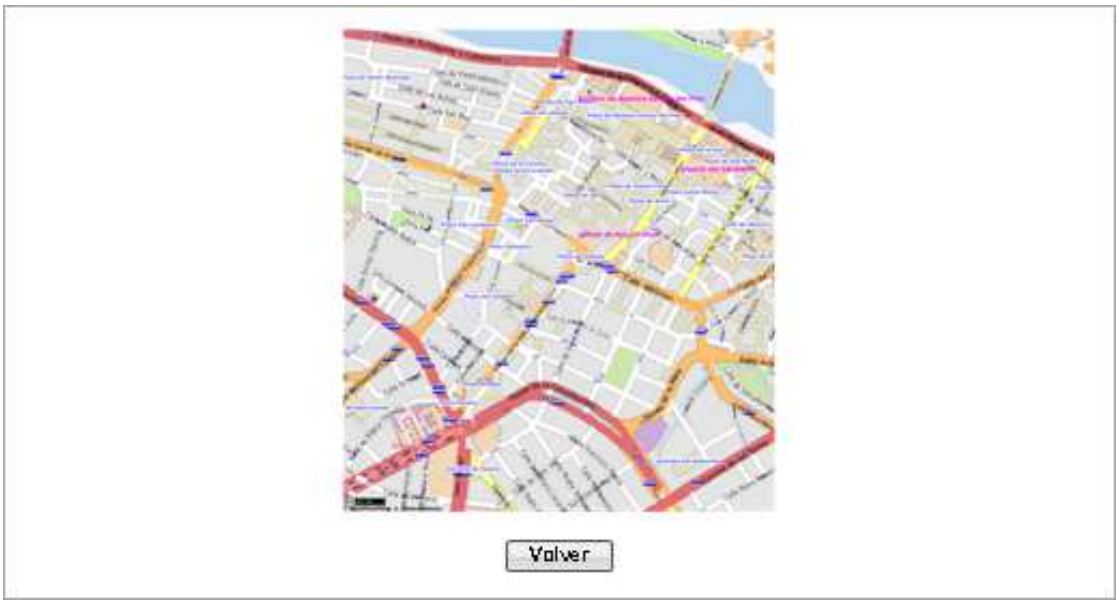


Figura A.18: Prototipo de ventana vista previa mapa

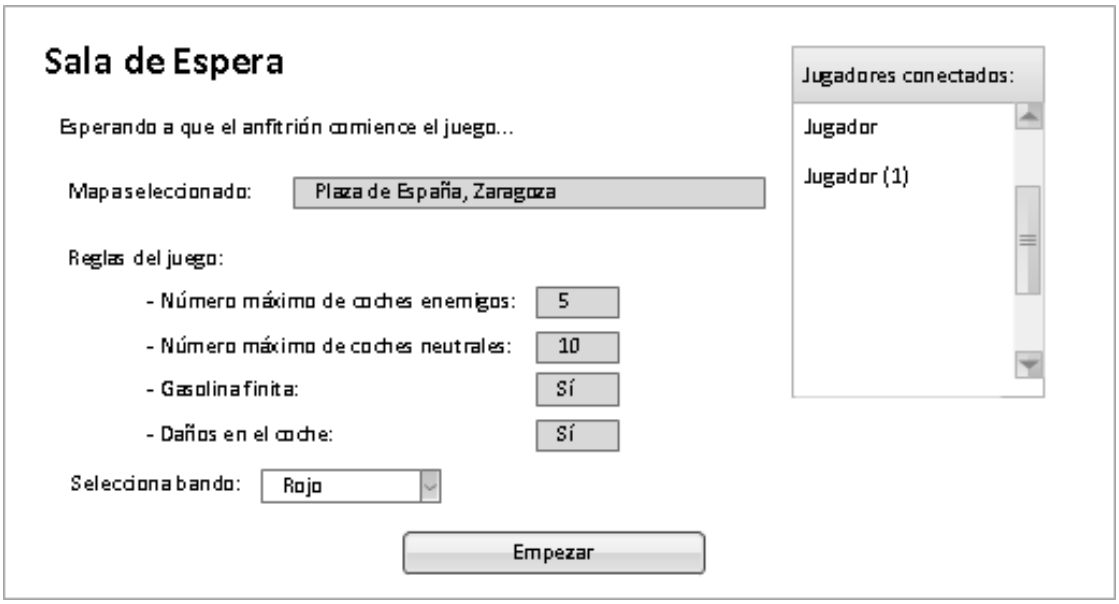


Figura A.19: Prototipo de ventana sala de espera (en creación)

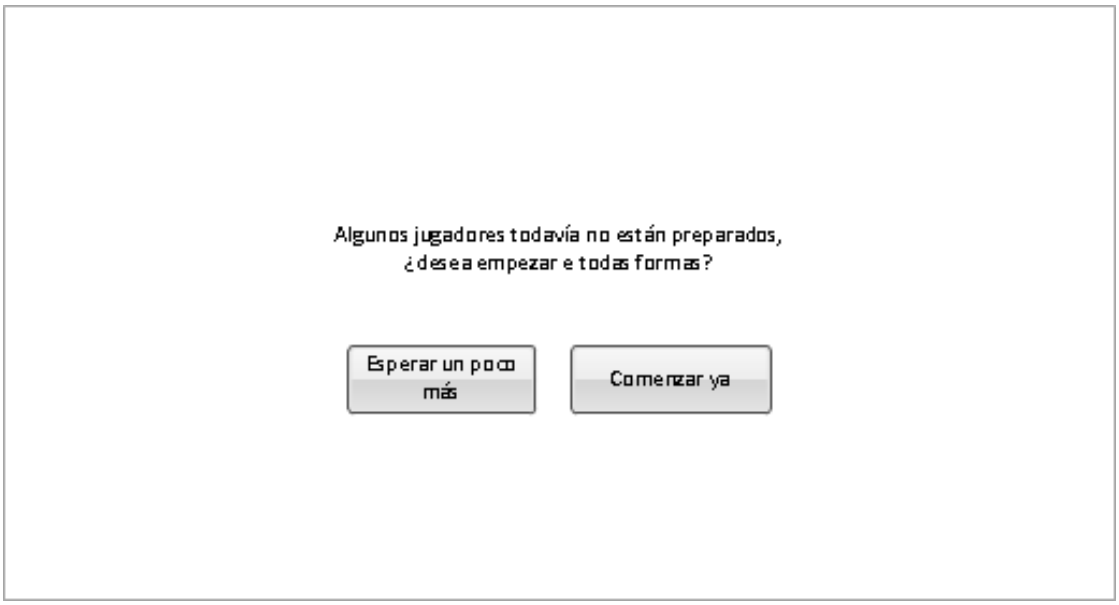


Figura A.20: Prototipo de ventana confirmación creación

Figura A.21: Prototipo de ventana cargando partida

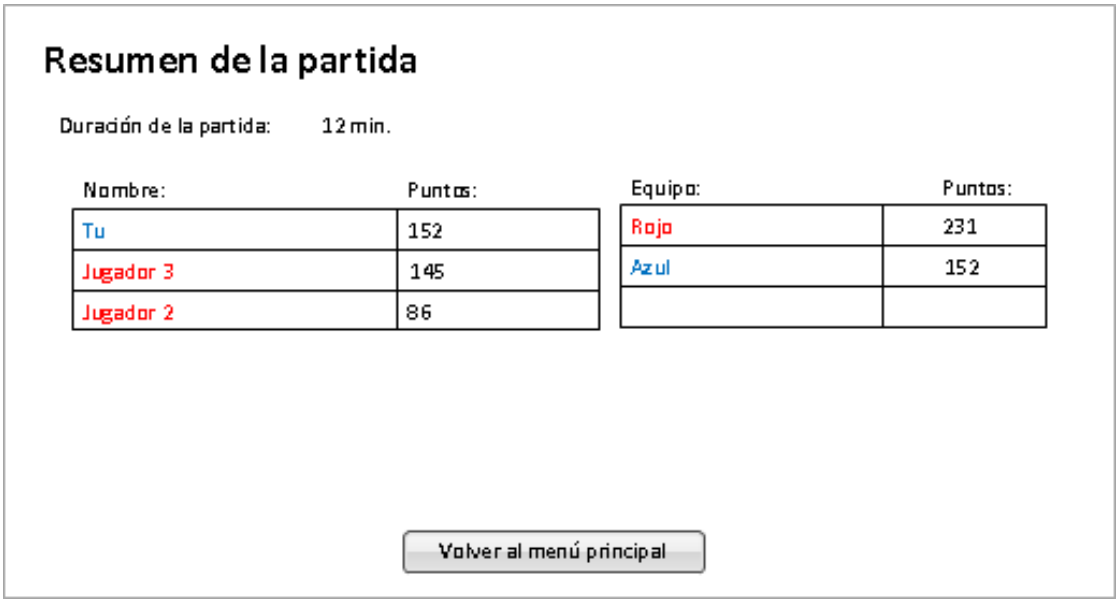


Figura A.22: Prototipo de ventana resumen después de partida

A.5. Modos de juego

A lo largo de diferentes reuniones y conversaciones por correo electrónico se analizó que diferentes modos de juego debían desarrollarse. Las tablas que contienen las conclusiones de dichas reuniones se pueden observar en las figuras A.23 y A.24.

TIPO	Jugadores	Equipos	Cuando se acaba	Objetivo	Enemigos IA	Competidores humanos	Comportamiento enemigos
Capturar banderas o coches enemigos	1..n	1	por tiempo o por objetivo	Banderas	si	no	Perseguir
				Enemigos			Huir
	2..n	2..n		Banderas	si	si	Perseguir
					no		-
				Enemigos	si		Huir
Seguir un plan finito	1	1		*	si	no	Perseguir
	2..n	2..n				si	
Seguir un plan infinito	1	1	por agotarse el dinero	*	si	no	Perseguir
	2..n	2..n				si	

Figura A.23: Modos de juego. Los objetivos marcados como asterisco (*) son tareas que pueden ser de los diferentes tipos desglosados en la figura A.24

Como se puede observar en la figura A.23, todos los modos de juego finalmente implementados corresponden a los de esta tabla a excepción del modo de juego

Tipo	Donde	Alias	Explicacion
ir a	dirección	en coche	llegar con el coche hasta el objetivo
		aparcar	aparcar en una plaza libre cercana
		a pie	aparcar y llegar a pie hasta el objetivo
	negocio	en coche	llegar con el coche hasta el objetivo
		aparcar	aparcar en una plaza libre cercana
		a pie	aparcar y llegar a pie hasta el objetivo

Figura A.24: Tipos de tarea de los modos de juego

de realizar tareas de aparcamiento «parking special mode», esto es así ya que ese modo de juego no estaba previsto inicialmente y se desarrolló durante la etapa final del proyecto, con el objetivo de tener un modo de juego que facilitase la realización de los experimentos asociados a la explotación. Éste modo de juego es una variación del modo «seguir un plan finito» con la particularidad de que cada ronda se divide en dos «subrondas» de forma que en la primera de ellas la tarea es siempre de tipo «llegada en coche» y la segunda de tipo «lograr aparcamiento».

Anexo B

Diseño

En este capítulo se mostrarán las diferentes capas y módulos que componen la arquitectura diseñada, junto con las clases más importantes de cada módulo. También se explicará el despliegue de la aplicación cuando se usan los servidores dedicados y el servidor de recogida de estadísticas. Por último, se finalizará explicando el llamado *game loop* (bucle de juego) y los hilos existentes en la ejecución.

También es importante anotar la autoría externa del diseño visual y sonoro de los vehículos y otros elementos del juego, si bien en su mayor parte han tenido que ser modificados para ajustarlos a las necesidades del videojuego.

B.1. Arquitectura de la aplicación

La arquitectura de la aplicación permite obtener un diseño a alto nivel del sistema, identificando los módulos de los que consta y las relaciones entre dichos módulos. El diseño que se ha realizado no se ha basado en ningún modelo específico existente, sino que se ha realizado un diseño específico para el videojuego desarrollado.

En la figura B.1 se muestran los distintos módulos de los que se compone la aplicación y sus relaciones. El papel realizado por cada uno de los módulos se expone a continuación.

- **Menús**: es el módulo que controla los menús del juego y todo lo necesario para su funcionamiento: obtención de dirección IP, carga de la configuración y los parámetros, etc.

- **Gestor de escenarios**: controla la gestión de los escenarios, desde lo relativo a su almacenamiento hasta su obtención a través de las APIs de OpenStreetMap.

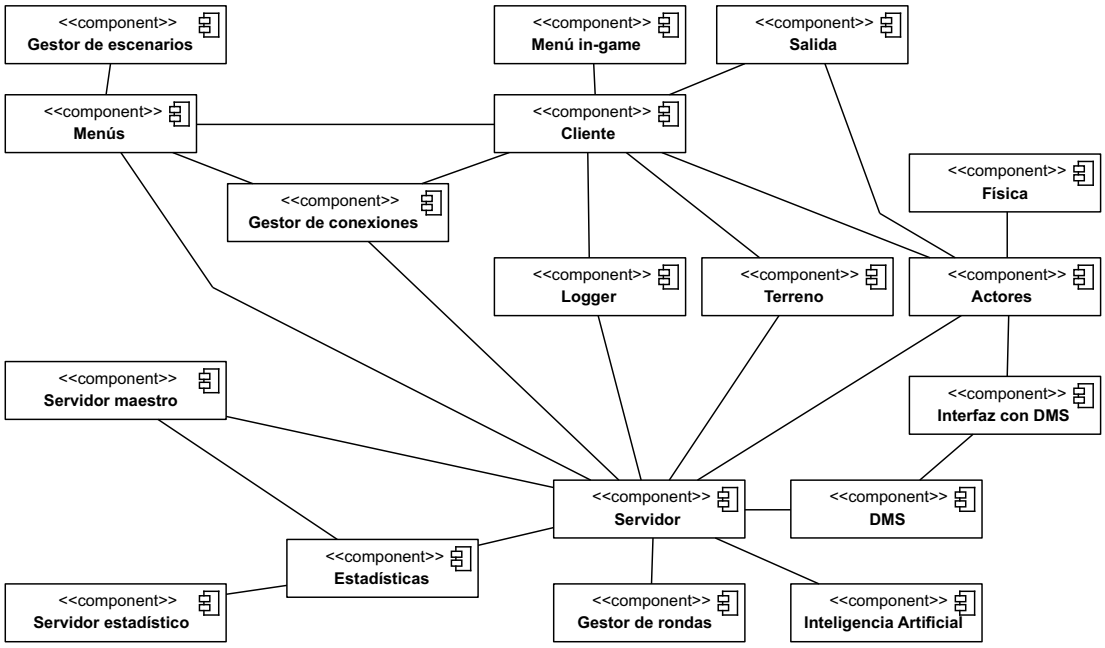


Figura B.1: Diagrama de componentes

- **Cliente**: el núcleo con la parte cliente del juego, que es la representación visual de la partida y las funciones necesarias para obtener a través de la red los datos del mundo de juego.
- **Salida**: contiene todo lo relacionado con el sonido(gestor de música mp3 y gestor de efectos de sonido, con sus correspondientes filtros) y con la gestión y representación de *sprites*.
- **Menú in-game**: es el módulo que controla el menú de pausa que se habilita durante la partida.
- **Servidor**: el núcleo con la parte servidor del juego. Contiene todo lo relativo a la partida a excepción de su representación.
- **Gestor de rondas**: gestiona las rondas y los objetivos pendientes y controla el modo de juego aplicado.
- **Inteligencia artificial**: contiene las clases referidas al manejo autónomo de los vehículos y también lo referente a los algoritmos de *path-finding*.
- **Terreno**: es el módulo en el que se definen los diferentes tipos de entidades usadas para la representación de los escenarios.

- **Actores**: contiene a todos los actores de la partida.
- **Física**: se encarga de la simulación física de la partida.
- **Gestor de conexiones**: es el módulo que contiene todos los elementos que intervienen en la transmisión de información a través de la red.
- **DMS**: contiene los interfaces del sistema DMS y la implementación de la estrategia utilizada (en este caso VESPA).
- **Interfaz con DMS**: contiene los interfaces que deben implementar los actores para ser accesibles desde el DMS.
- **Logger**: es el módulo que controla la presentación en la consola de diferentes trazas de ejecución.
- **Estadísticas**: contiene todo lo necesario para la obtención de estadísticas del juego y del DMS.
- **Servidor maestro**: permite el uso de un servidor dedicado para las partidas y contiene también el proceso que se usa para la comunicación remota con dicho servidor.
- **Servidor estadístico**: contiene todo lo relacionado con el servidor de recogida de estadísticas.

B.2. Capas de la arquitectura

Se pueden distinguir dos capas diferenciadas en la arquitectura: la capa del núcleo y la capa de los suplementos.

- **Núcleo**: Esta capa contiene todos los elementos necesarios para el funcionamiento del juego. Cualquier cambio dentro de esta capa alterará el funcionamiento del juego. Dentro de esta capa se controla tanto lo referente a la partida como a la navegación por los menús.
- **Suplementos**: Esta capa contiene los elementos que no son imprescindibles para el funcionamiento del juego y que pueden ser alterados, sustituidos o eliminados sin afectar al resto de módulos. Un ejemplo es el fácil reemplazamiento del módulo DMS (Data Management Strategy) que es explicado con detalle en el anexo D.1.

En la figura B.2 se representa la clasificación de los distintos módulos del diagrama de componentes diferenciados según su capa (en color blanco los módulos de la capa núcleo y en gris los módulos de la capa suplementos).

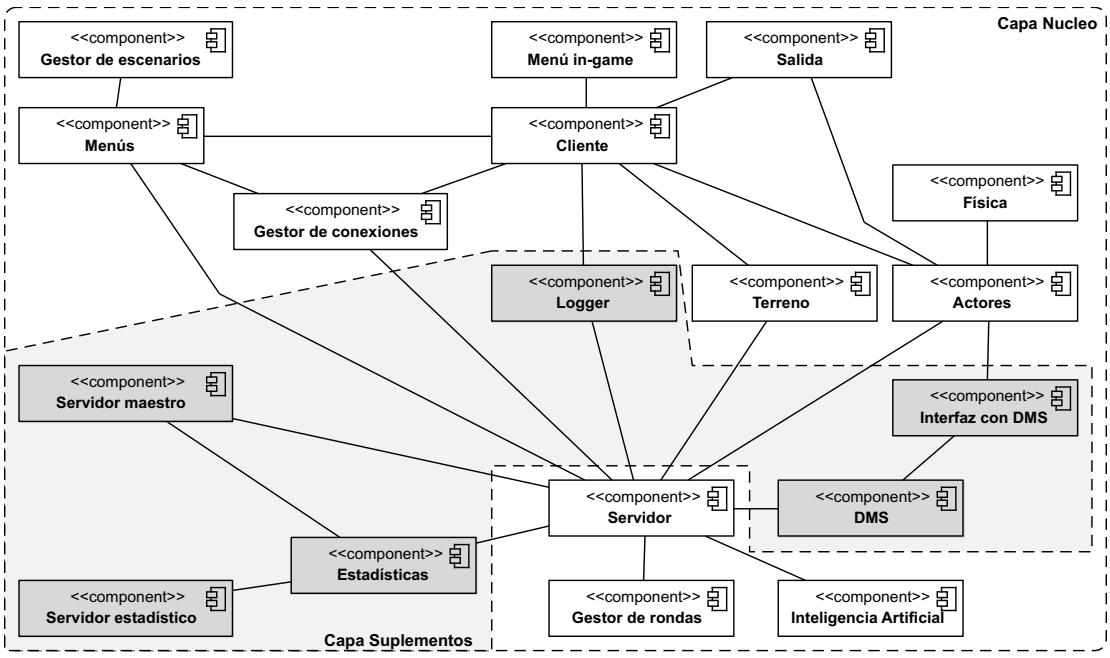


Figura B.2: Capas de la arquitectura

B.3. Despliegue

En esta sección se puede ver el despliegue de los diferentes componentes, desglosado en dos figuras (figura B.3 y figura B.4) según si se usa o no un servidor dedicado.

Es importante anotar que todos los componentes de la aplicación se distribuyen conjuntamente empaquetados en un fichero JAR, por lo que estos diagramas muestran únicamente los componentes de los que se hace uso desde cada ubicación.

En el caso de no usar un servidor dedicado (figura B.3), el jugador anfitrión crea la partida conectándose la parte cliente de la aplicación con la parte servidor (a través de la red, de igual forma que si la conexión se realizara entre diferentes computadores). El resto de jugadores se conectarán únicamente sus partes cliente de la aplicación con la parte servidor del jugador anfitrión. Cuando finalice la partida, las estadísticas recogidas durante la partida se almacenarán en ficheros locales del computador del jugador anfitrión y también realizará una conexión con el servidor de recogida de estadísticas, para que en el caso de que éste responda, enviarle dichas estadísticas.

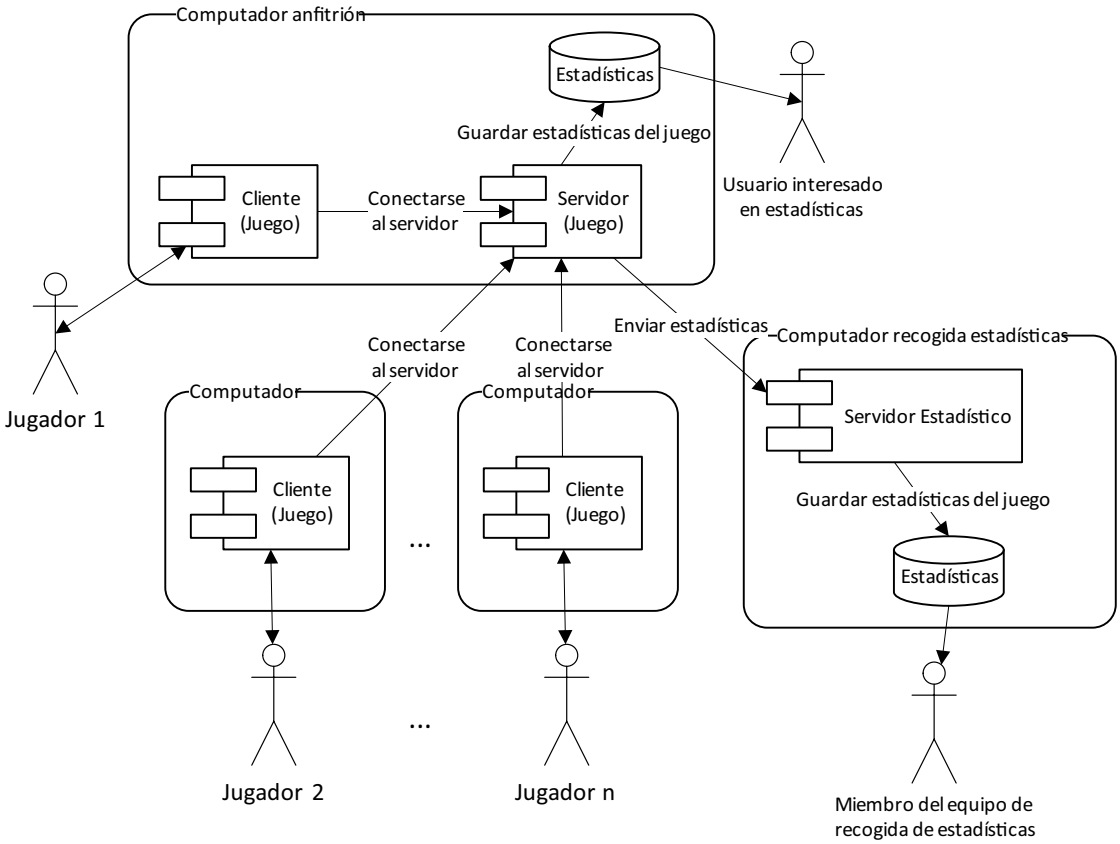


Figura B.3: Pseudo diagrama de despliegue sin servidor dedicado

La principal diferencia cuando se hace uso de un servidor dedicado (figura B.4) radica en que no existe un jugador anfitrión de la partida, sino que todos los jugadores son iguales. Estos jugadores no se conectarán directamente con el servidor de la partida, sino que realizarán una petición al servidor maestro, que convenientemente estará recibiendo mensajes en el mismo puerto que lo haría un servidor normal, y éste les contestará a los clientes enviándoles el puerto en el que se encuentra funcionando el servidor de la partida (que se creará en el caso de que no estuviese en funcionamiento). Conociendo este puerto, los clientes se conectarán al servidor de la partida de la misma forma que se conectarían en el caso de no usar servidor dedicado. Todo este proceso se habrá realizado de forma transparente para el usuario, que habrá de seguir el mismo método que el utilizado para la conexión con un servidor no dedicado.

Usando servidor dedicado hace acto de presencia un componente más que es el terminal, el cual consiste en una interfaz que permite modificar la configuración del servidor dedicado de forma remota.

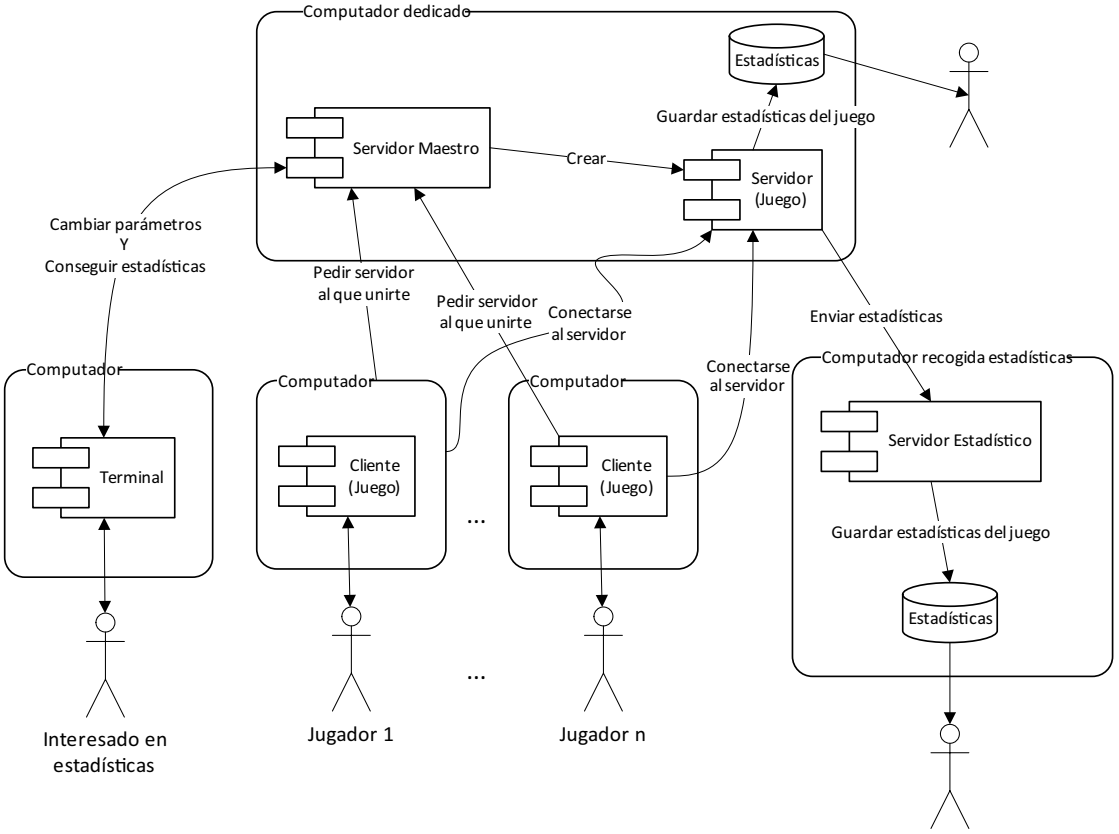


Figura B.4: Pseudo diagrama de despliegue con servidor dedicado

B.4. Diagramas de clases

En esta sección pretende establecer una definición más específica de cada uno de los componentes anteriormente descritos, estableciendo para cada uno de ellos las clases que los componen y cuál es su cometido.

A continuación se detallan todos los módulos mencionado anteriormente (ver sección B.1) a excepción del módulo «DMS» y el módulo «Interfaz con DMS», ya que éstos ya han sido explicados con todo detalle en el anexo D.1.

B.4.1. Módulo de salida

El módulo de salida actúa como controlador de los dispositivos gráfico y sonoro del computador, facilitando al resto de módulos representar de forma visual y sonora los elementos del juego.

Este módulo consta dos partes diferenciadas, una dedicada a la representación gráfica y otra dedicada a los aspectos sonoros.

La parte dedicada a los gráficos esta formada por la clase «SpriteCache» que hereda de «ResourceCache», la cual es la versión genérica de un recurso que deba ser utilizado de forma recurrente.
La clase «ResourceCache» permite almacenar recursos de cualquier tipo en una estructura *HashMap* de forma que posteriormente sean accedidos con facilidad. La case «SpriteCache» es la clase derivada para el caso de almacenamiento de recursos de tipo *BufferedImage*, que son los que nos interesan para la representación visual de las entidades.

La otra parte del módulo es la dedicada al sonido, la cual a su vez se divide en las clases relacionadas con la reproducción de sonidos *WAV*, que se han obtenido del libro «Developing Games in Java» [4] y en las clases relacionadas con la reproducción de sonidos *MP3*, para lo cual se ha usado la librería *JLayer*[1] modificándola para que soporte la reproducción en bucle, las listas de reproducción y el cambio dinámico de volumen. De ésta parte se ha representado en el diagrama únicamente la clase «MiPlayer» ya que es la modificación respecto a la librería. El resto de clases del paquete «sonido» están dedicadas a la parte de sonido *WAV*.

De la parte dedicada al sonido *WAV* cabe destacar las clases «Filter3d» y «FilterVolume» ya que son las únicas nuevas desarrolladas respecto a las mostradas en [4]. Estas clases permiten aplicar a los sonidos un filtro de distancia y de volumen respectivamente, siendo el primero utilizado para los efectos sonoros ligados a los actores de la partida (p.ej. el sonido que emiten las ambulancias, que disminuye con la distancia) y el segundo utilizado para todos los sonidos, ya que es imprescindible poder controlar el volumen del juego.

[1] `http://www.javazoom.net/javalayer/javalayer.html`

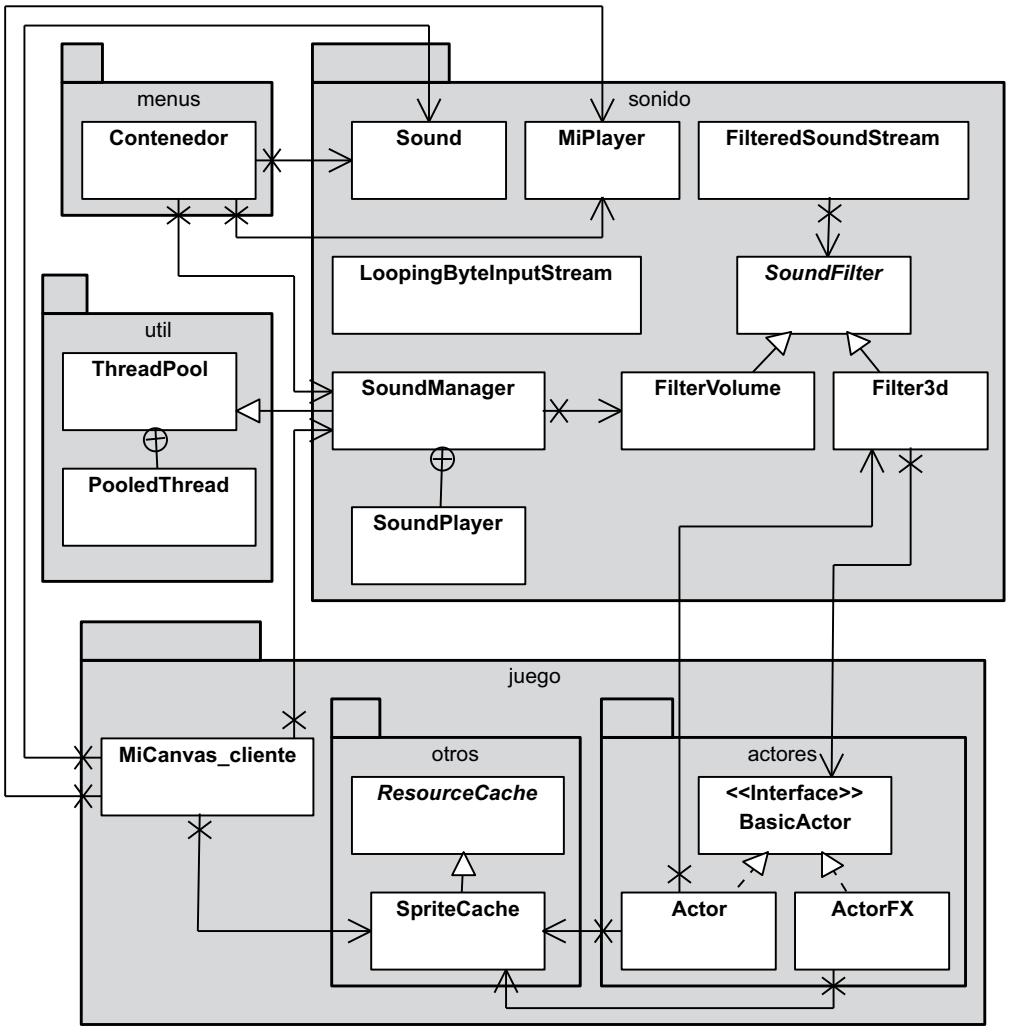


Figura B.5: Clases del módulo de salida

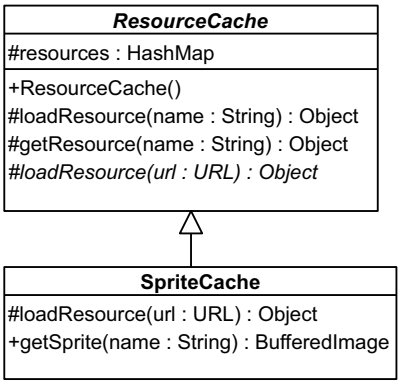


Figura B.6: Detalle de las clases «SpriteCache» y «ResourceCache»

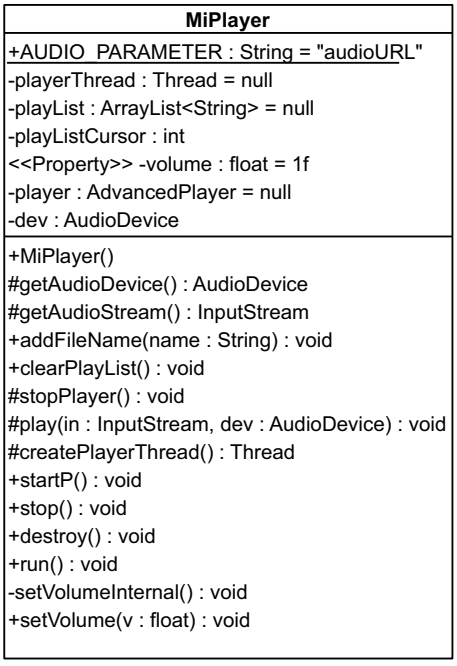


Figura B.7: Detalle de la clase dedicada al sonido *MP3*

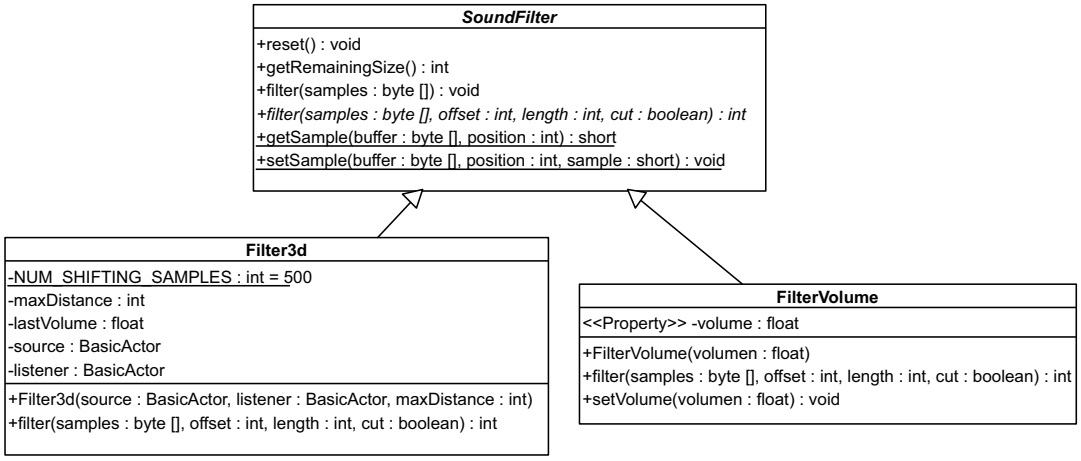


Figura B.8: Detalle las clases dedicadas al sonido *WAV*

B.4.2. Módulo de menús

El módulo de menús es el que contiene todas las clases utilizadas para la representación y el manejo del menú del juego.

Las clases principales son «Contenedor» y «PantallaAnimada».
La clase «Contenedor» es instanciada desde «VentanaPPal», que es la clase inicial del juego cuando se ejecuta como *Applet* (cuando se ejecuta como aplicación de escritorio la clase inicial es «StartClass», que salvo que se requiera un servidor dedicado, servidor estadístico o terminal, crea una instancia de «VentanaPPal»).
La clase «Contenedor» contiene el *JPanel* en el que se mostrarán las pantallas de los menús y es la encargada de iniciar el cliente y el servidor.
La clase «PantallaAnimada» es la superclase de la que derivan todas las pantallas de los menús y contiene el comportamiento básico de las pantallas.

Dentro de este módulo también se incluyen una serie de clases necesarias para realizar ciertos cometidos del menú, como redimensionar una imagen *JPEG*, comprobar si dicha imagen no está dañada, simplificar una cadena de texto eliminando los caracteres no latinos, etc. Todas estas clases se sitúan en el paquete «util».

Otro paquete contenido es «workers», que contiene las clases derivadas de «SwingWorker» de las que hacen uso las pantallas para realizar tareas costosas en tiempo sin congelar el hilo de ejecución que las muestra.

El paquete «red» contiene las clases que se necesitan para obtener los datos de la información de la partida (durante su trascurso o en su finalización) para representarla en la pantallas de sala de espera (al unirte) y en la pantalla con el resumen de la partida (al finalizarse).

Por último, también existen las clases «Parametros» y «ParametrosConfig»

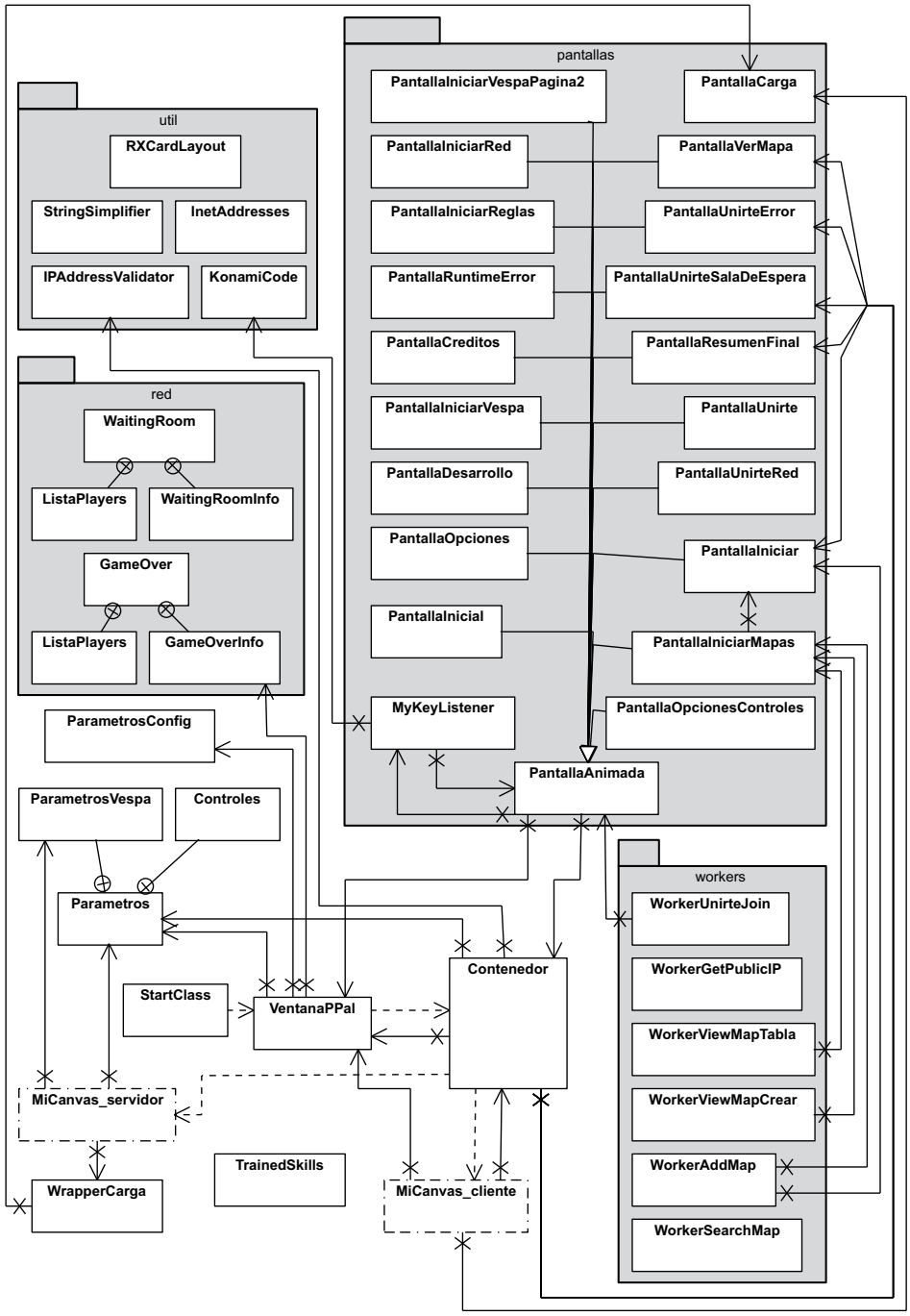


Figura B.9: Clases del módulo de menús

Figura B.10: Detalle de la clase «Contenedor»

Figura B.11: Detalle de la clase «PantallaAnimada»

que contienen los valores de configuración almacenados en las pantallas y los valores descritos mediante el fichero de texto *ParamConfig.txt* respectivamente. La clase «TrainedSkills» es la que contiene las funciones y estructura necesarias para el cálculo de la pericia del jugador (ver 3.3), y la clase «WrapperCarga» es la utilizada para poder transmitir el estado actual de carga de la partida desde la clase «Cliente» a la barra de progreso de la pantalla de carga.

B.4.3. Módulo gestor de escenarios

El módulo gestor de escenarios contiene las clases que permiten el manejo y almacenamiento de los mapas.

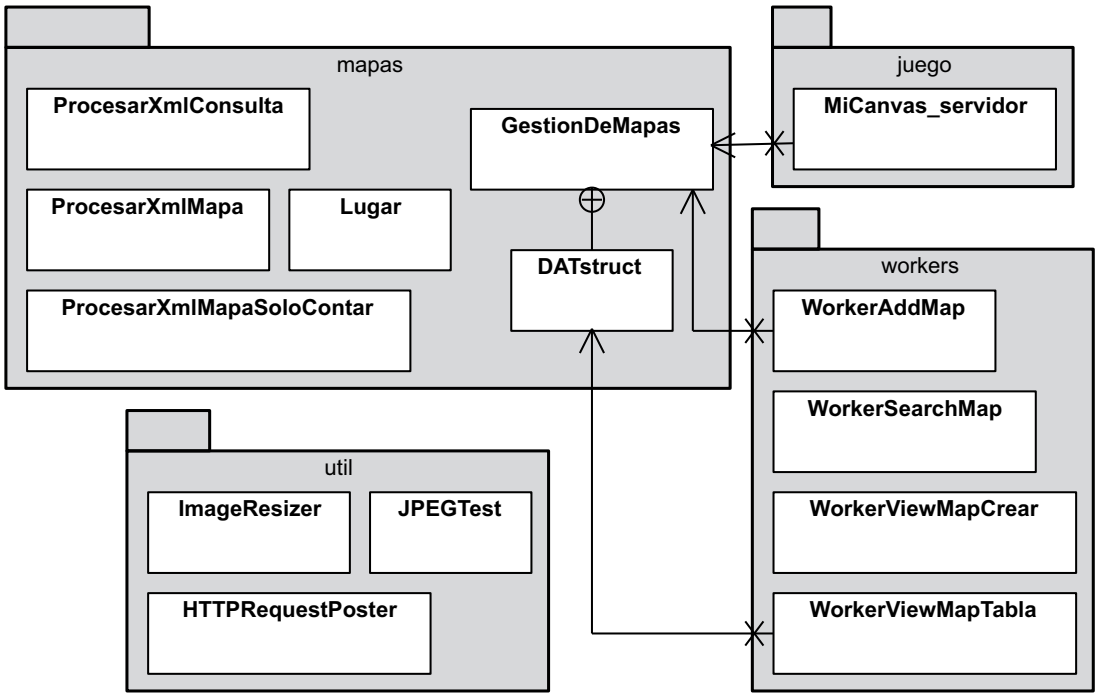


Figura B.12: Clases del módulo de gestor de escenarios

La clase «GestionDeMapas» es la principal y contiene funciones para crear los diferentes tipos de ficheros necesarios (ver anexo C.2.2) y procesarlos, así como todas las funciones que se requieren desde la pantalla del menú «Configuración avanzada de mapas» (listar los mapas, descargar uno nuevo, descargar la visualización, etc.). También contiene el listado de APIs que podrán ser usadas para la obtención de los ficheros.

Figura B.13: Detalle de la clase «GestionDeMapas»

La clase «DATstruct» contiene la estructura de almacenamiento de los mapas, mientras que la clase «Lugar» tiene la estructura de almacenamiento de las peticiones de obtención de dichos mapas.

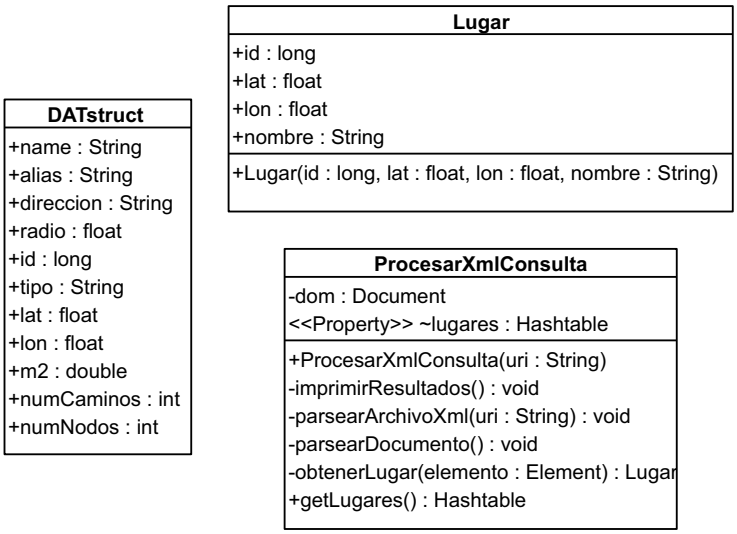


Figura B.14: Detalle de las clases «DATstruct», «Lugar» y «ProcesarXMLConsulta»

Las clases «ProcesarXML...» se utilizan para procesar el contenido de los ficheros *XML* descargados que contienen el listado de resultados de la búsqueda «ProcesarXMLConsulta» y los datos del escenario en formato *OSM*. La diferencia entre las clases «ProcesarXMLMapa» y «ProcesarXMLMapaSoloContar» radica en que esta última es la utilizada para calcular el número de nodos del escenario y mostrarlo en la tabla de la pantalla del menú, mientras que la primera es la utilizada para cargar los datos al mundo de juego.

B.4.4. Módulo de servidor maestro

El módulo de servidor maestro contiene las clases necesarias para el funcionamiento del servidor dedicado y del terminal cliente con el que se puede modificar sus parámetros.

La clase «MasterServer» es la encargada de la inicialización del servidor maestro y la configuración del directorio. Recibe conexiones TCP de los terminales e inicia el servidor de la partida cuando es requerido.
Para tratar con las conexiones de los terminales, con cada nueva conexión crea un hilo de ejecución con una instancia de la clase «HiloMasterServer». Esta clase tiene las funciones necesarias para modificar la configuración de la partida.

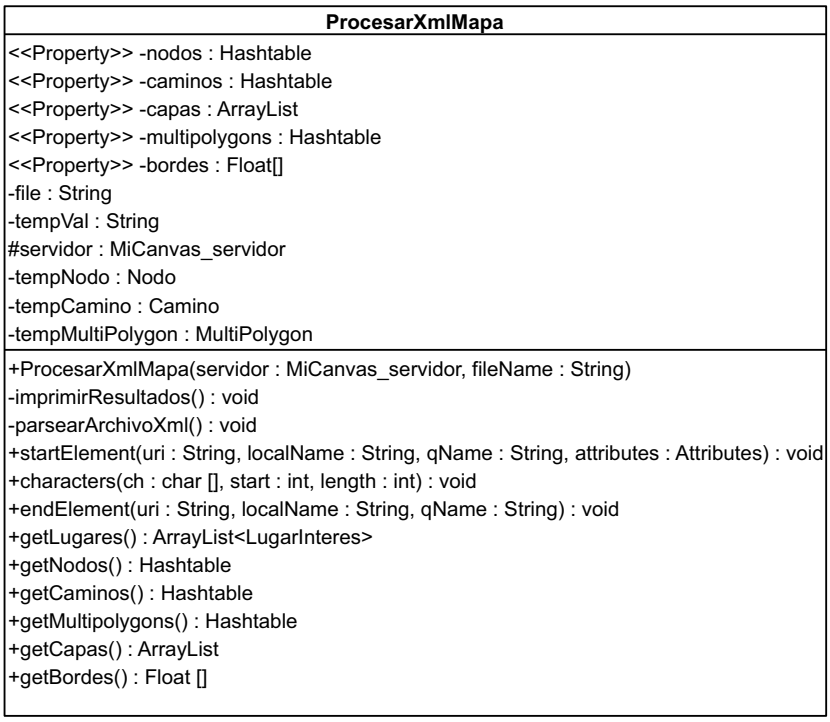


Figura B.15: Detalle de la clase «ProcesarXMLMapa»

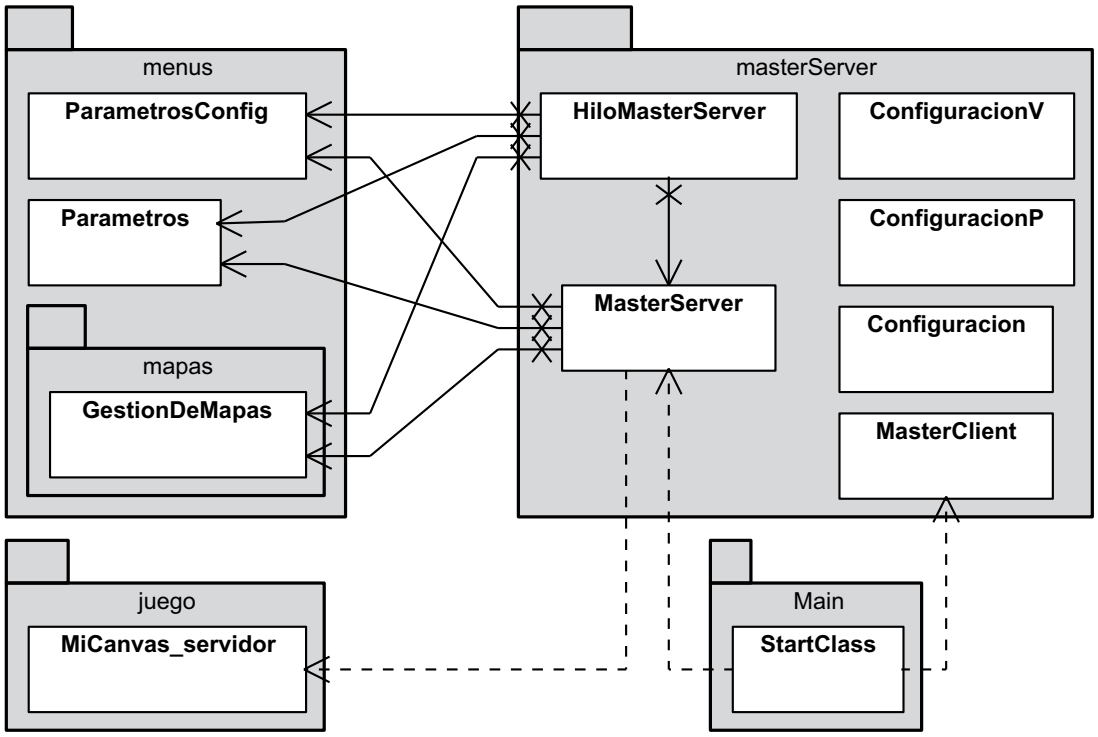


Figura B.16: Clases del módulo de servidor maestro

La clase «MasterClient» es la encargada de realizar la conexión mediante TCP con el servidor maestro y tiene las funciones necesarias para obtener y modificar la configuración actual de las partidas que inicie dicho servidor.

Si los parámetros usados así lo requieren, tanto «MasterServer» como «MasterClient» son instanciadas desde la clase «StartClass», que es la clase inicial de la aplicación cuando esta no se ejecuta como *Applet*.

B.4.5. Módulo de servidor estadístico

El módulo de servidor estadístico contiene las clases necesarias para el despliegue del servidor de recogida de estadísticas y el envío y recepción de éstas.

La clase «StatServer» se encarga de iniciar el proceso y escuchar peticiones de conexión, creando instancias de la clase «HiloStatServer» cuando dichas peticiones de conexión son aceptadas.

La clase «HiloStatServer» contiene funciones para recibir los diferentes tipos de estructuras de estadísticas por parte de los servidores.

Por último la clase «EnvíoEstadísticas» es una clase en la cual se definen métodos (estáticos) para que los servidores puedan enviar los diferentes tipos de estadísticas a los servidores de recogida de estadísticas.

B.4.6. Módulo de estadísticas

El módulo de estadísticas contiene las clases que permiten la creación de las diferentes estructuras de estadísticas (de aparcamiento, de juego y de VESPA) y la estructura que contiene el resumen de la configuración actual de la partida. Además también contiene las funciones necesarias para almacenar en ficheros dichas estructuras de forma legible al usuario.

Para cada tipo de fichero que se desea generar cuando las estadísticas estén activadas, se tiene una clase en la que se contiene su estructura «TADEstadisticasAparcamiento», «TADEstadisticasJuego», «EstadisticasVespa (EV)» y «CurrentConfigInfo (CCI)», para estadísticas de aparcamiento, de la partida, de VESPA y para el resumen de la configuración respectivamente.
Para cada uno de estos tipos se tiene una clase con las funciones necesarias para limpiar el directorio donde se almacenan los ficheros y crear el fichero, además de los métodos llamados desde el juego y mediante los cuales se calculan las estadísticas.

Las estadísticas de aparcamiento tienen dos clases asociadas «TADEstadisticasAparcamiento» y «TADEstadisticasAparcamientoTC» cuya diferencia es que la primera es llamada por los jugadores humanos mientras que la última está ideada para vehículos del tráfico, por lo que la función que incluyen para anotar un apar-

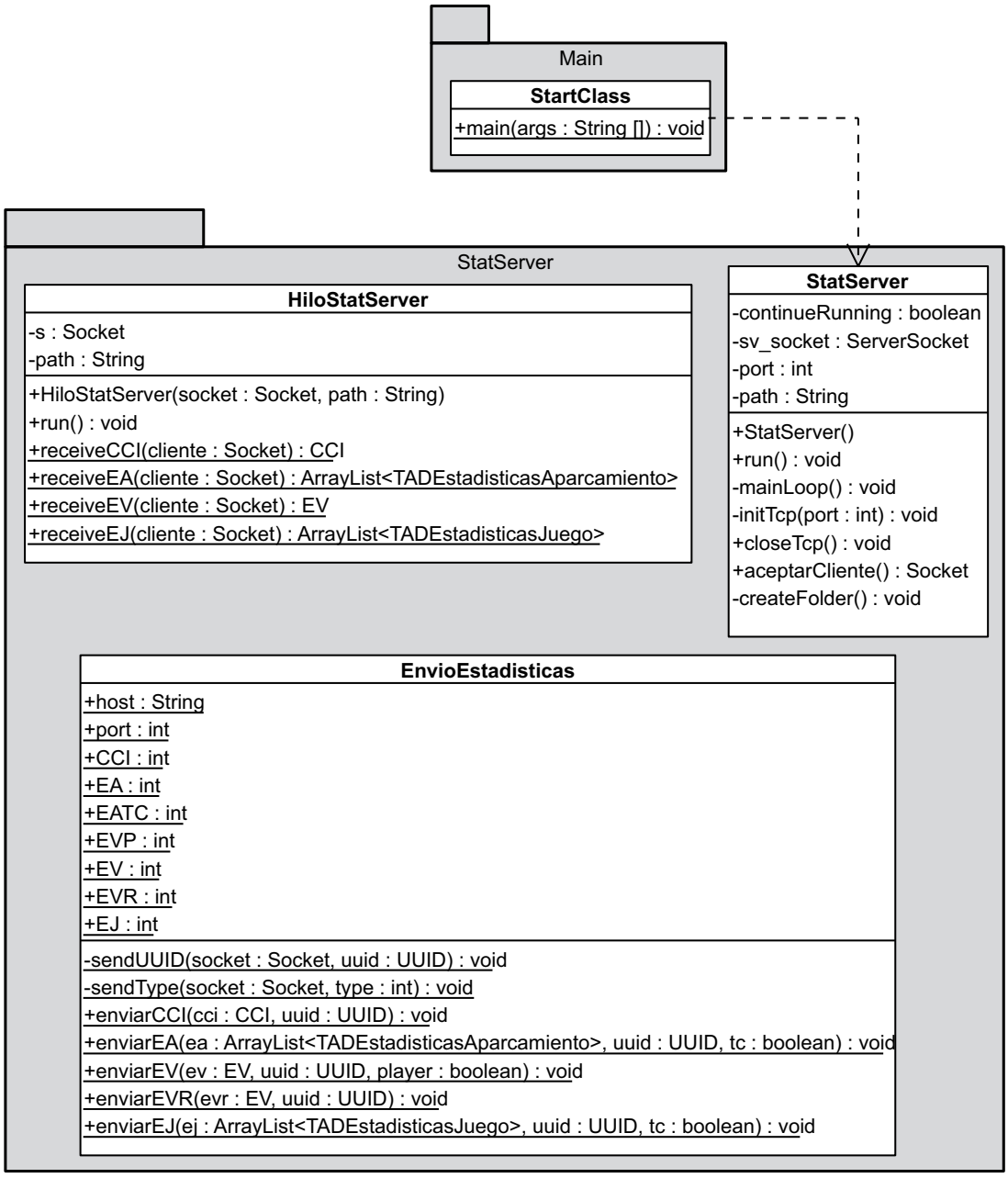


Figura B.17: Clases del módulo de servidor estadístico

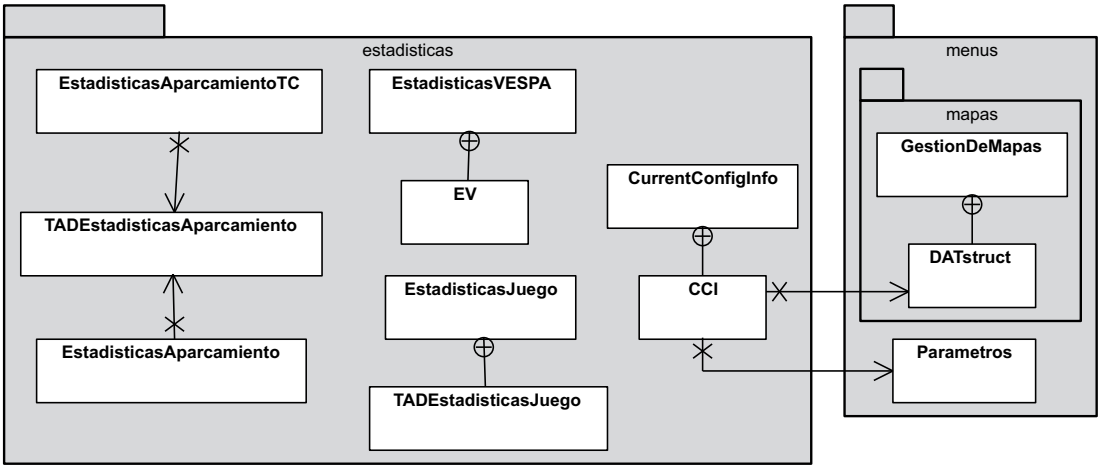


Figura B.18: Clases del módulo de estadísticas

CurrentConfigInfo
+StatisticsOn : boolean -pathName : String
+createCleanFolder(uri : String, date : String) : void +createFolder(path : String, uuid : String) : void +writeFile(p : Parametros, mDJ : byte, numPlayers : int, timeElapsed : int, mapData : DATstruct) : void +writeFile(cci : CCI, path : String, uuid : String) : void -protocol2String(p : int) : String +getPath() : String

Figura B.19: Detalle de la clase «CurrentConfigInfo (CCI)»

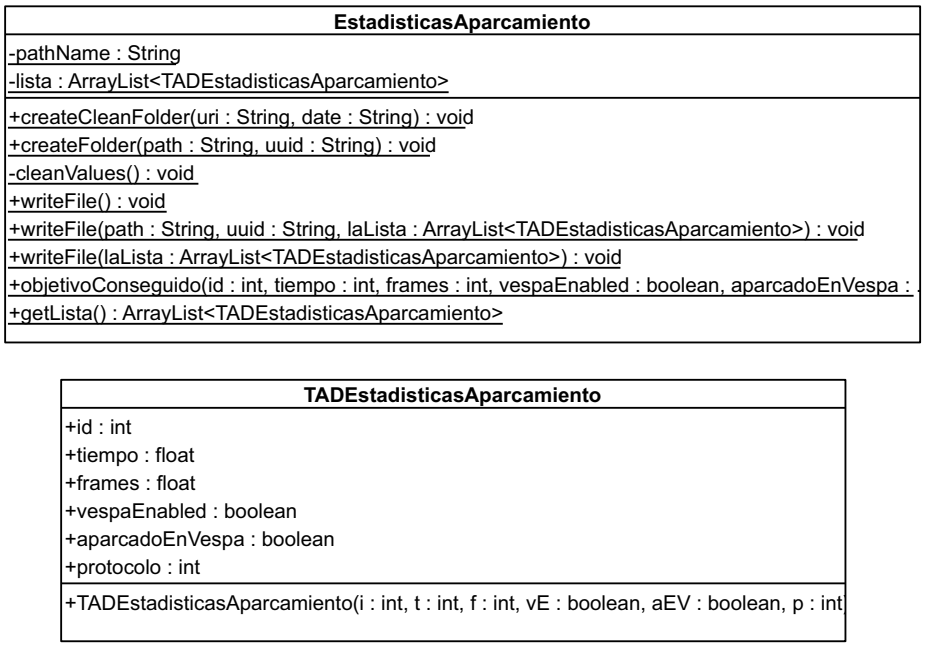


Figura B.20: Detalle de las clase «TADEstadisticasAparcamiento» y «EstadisticasAparcamiento»

camiento exitoso varía, así como también el formato con el que se representará en el fichero.

Las estadísticas de VESPA se almacenan tanto individualizadas para cada vehículo como de forma agregada para toda la partida, por lo que la clase «EstadisticasVESPA» se diferencia de las demás en que incluye un método para la agregación de los datos y también otro para la representación en fichero con diferente formato.

B.4.7. Módulo logger

El módulo logger permite la activación o desactivación de la impresión por pantalla de diversas trazas de ejecución.

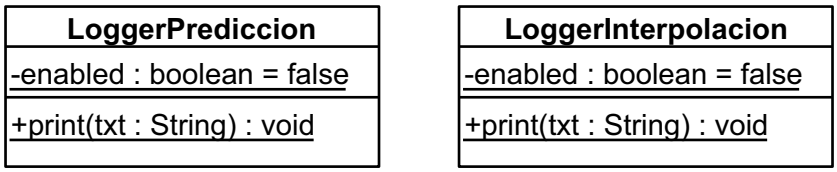


Figura B.21: Clases del módulo de logger

Consiste de únicamente dos clases, pensadas para las trazas referentes a la predicción y a la interpolación, pero es fácilmente expandible añadiendo una nueva clase para cada nuevo tipo de traza.

También es fácilmente expandible para que se seleccione si la traza se pintará por pantalla (como actualmente) o en fichero.

B.4.8. Módulo de menú in-game

El módulo de menú in-game contiene las clases necesarias para mostrar el menú de pausa y permitir navegar por él mediante los eventos del ratón y la tecla escape.

Este módulo se compone de cuatro clases («Principal», «Opciones», «Controles» y «SalirConfirmación») que contienen la estructura de las diferentes pantallas del menú (principal, opciones, controles y abandonar) y otras cuatro («MenuInGamePrincipal», «MenuInGameOpciones», «MenuInGameControles» y «MenuInGameSalirConfirmacion») con las respectivas acciones de pintado y manejo de dichas pantallas.
Además, la clase «MenuEscape» contiene la estructura de pantallas y es con la que se comunican las clases de manejo.

La clase «MiBoton» es la que contiene la estructura de la representación de los botones utilizados en las pantallas.

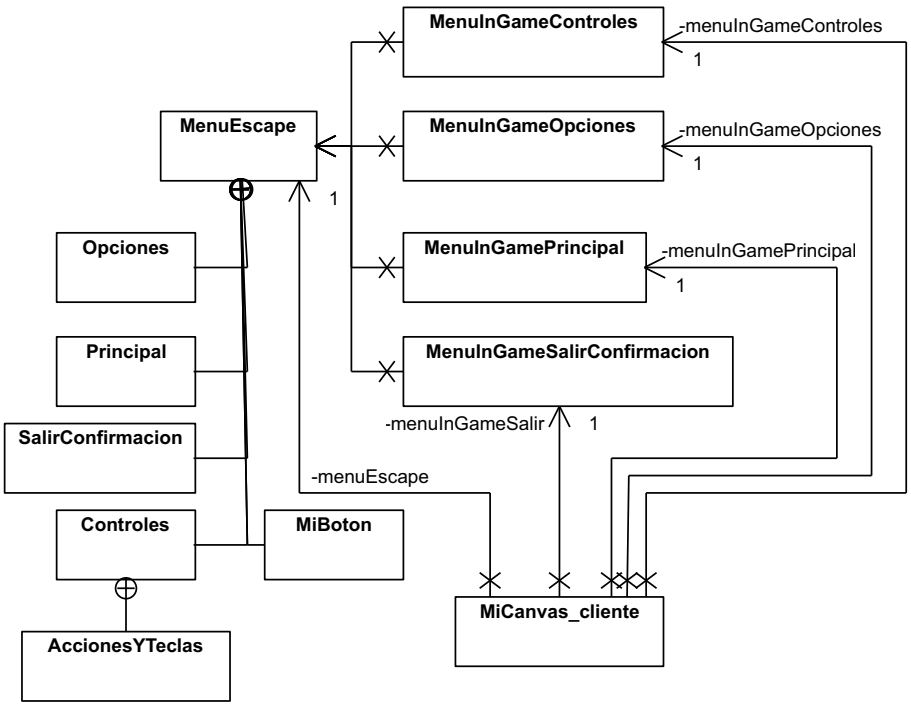


Figura B.22: Clases del módulo de menú in-game

MenuInGameControles
+ACCION_LENGTH : int +TECLA_LENGTH : int +BOTON_LENGTH : int -cliente : MiCanvas_cliente -menuEscape : MenuEscape
+MenuInGameControles(cli : MiCanvas_cliente, me : MenuEscape) +mouseClicked(e : MouseEvent) : void +mousePressed(e : MouseEvent) : void +mouseReleased(e : MouseEvent) : void +cl_paintMenuEscape(g : Graphics2D) : void -accionClicked(texto : String) : void

Figura B.23: Detalle de la clase «MenuInGameControles»

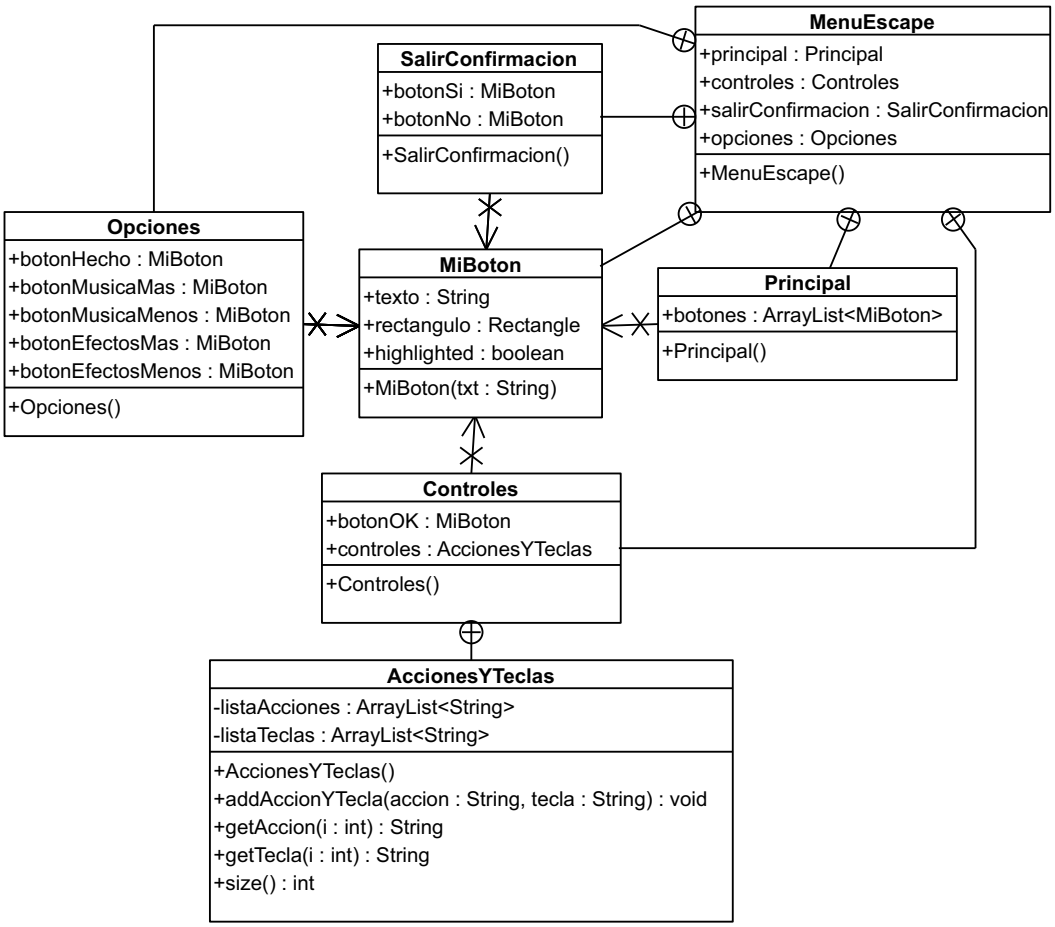


Figura B.24: Detalle de la clase «MenuEscape» y las relacionadas

B.4.9. Módulo de terreno

El módulo de terreno es el encargado de soportar la representación de los distintos elementos que conforman los elementos estáticos del escenario.

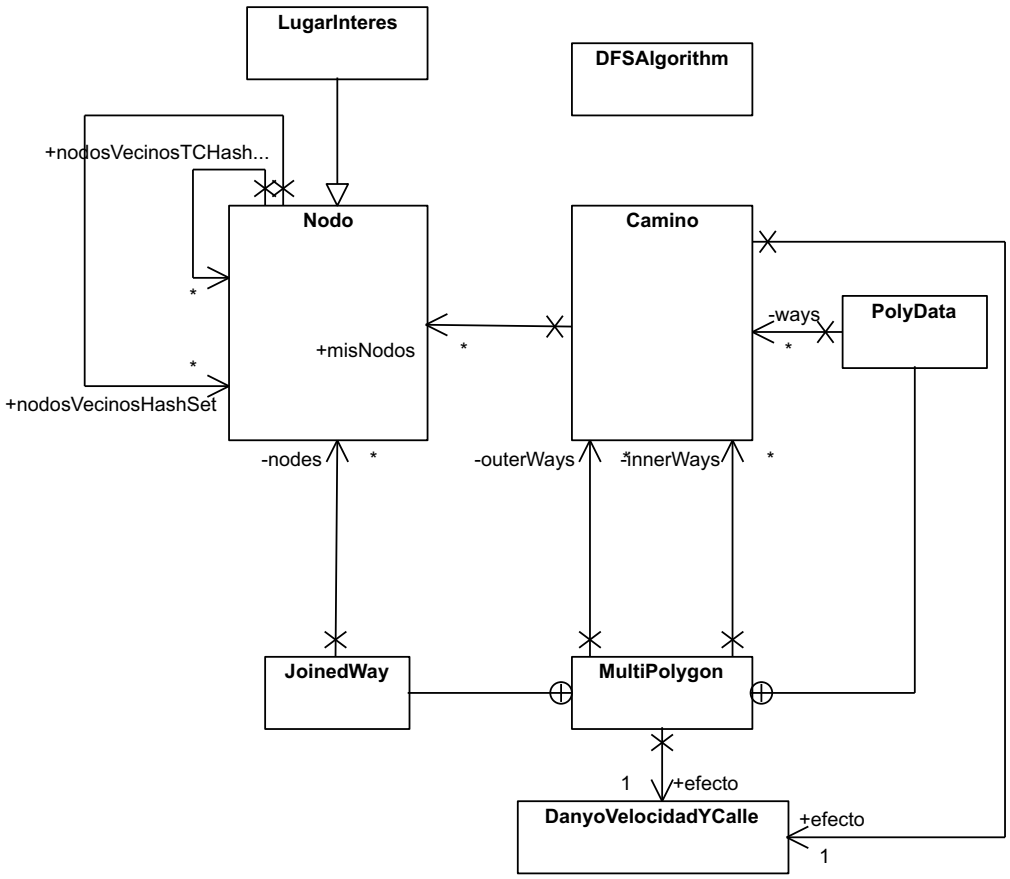


Figura B.25: Clases del módulo de terreno

En éste módulo hay que destacar tres clases: «Nodo», «Camino» y «MultiPolygon». Son las clases que representan a los respectivos tipos de datos de OpenStreetMap y contienen las funciones necesarias para su creación y manejo.

Hay que destacar que un multipolígono está formado por caminos y éste a su vez está formado por nodos. Además cada nodo tiene una lista con sus nodos vecinos (*nodosVecinosHashSet*) y otra lista casi idéntica (*nodosVecinosTCHashSet*) pero en la que solo aparecen los nodos accesibles con las restricciones de los vehículos del tráfico (que no pueden circular por los mismos tipos de terreno que los jugadores y enemigos).

La clase «LugarInteres» es una subclase de «Nodo» y contiene la estructura y los métodos necesarios para que pueda ser elegido como objetivo en los modos de juego que funcionan mediante tareas (por ejemplo pintar el radio del objetivo o calcular sus plazas de aparcamiento cercanas).

La clase «DanyoVelocidadYCalle» contiene el tipo de efecto causante por el terreno: ralentización de velocidad, infranqueable, causa daños, etc. y además también contiene la calle a la que pertenece. Esta clase se usa para no tener que recalcular los efectos sobre los vehículos cada vez que se detecte que un elemento del terreno está en colisión con un vehículo.

Por último también existe la clase «DFSAlgorithm» que contiene un algoritmo de búsqueda primero en profundidad usado para calcular los «sectores» en los que se divide el escenario, es decir, que nodos son accesibles desde otros nodos. Esto se usa para evitar representar nodos que pertenecen a caminos a los que es imposible que los vehículos accedan.

Para más información ver anexo C.3.

B.4.10. Módulo gestor de conexiones

El módulo gestor de conexiones contiene todas las clases implicadas en el envío y recepción de datos a través de los protocolos TCP y UDP.

Este módulo puede dividirse en dos partes: las clases de uso general en toda la aplicación (paquete «conexion») y las clases que solo se usan durante la partida (paquete «juego.red»).

Respecto a los primeros, las clases «TCP_servidor», «TCP_cliente» y «TCP_master_related» contienen las funciones usadas para la comunicación mediante el protocolo TCP para el servidor, el cliente y el servidor dedicado y su terminal respectivamente. Todas ellas implementan la interfaz «TCP_comun», que únicamente contiene los valores asociados a los tipos de mensaje que son comunes a todas las implementaciones.
Las clases «UDP_servidor» y «UDP_cliente» son las clases equivalentes para el protocolo UDP.

La clase «Cliente» es la encargada de almacenar en el servidor los datos de cada cliente conectado. No solo se almacena la IP y puerto sino también la secuencia del último paquete recibido y enviado y una lista donde se almacenan los últimos estados enviados para poder realizar la descompresión delta.

La clase «GestorTCPCliente» crea un hilo de ejecución en el cliente y se encarga de realizar la petición de unión a la partida y, más adelante, recibir los datos de cada nueva ronda, la información de finalización de la partida y también los *eventos GUI* que se muestran al jugador.

La clase «GestorConexiones» es la versión análoga para el servidor, dedicándose a enviar dichos datos también se encarga de gestionar la unión de jugadores a la

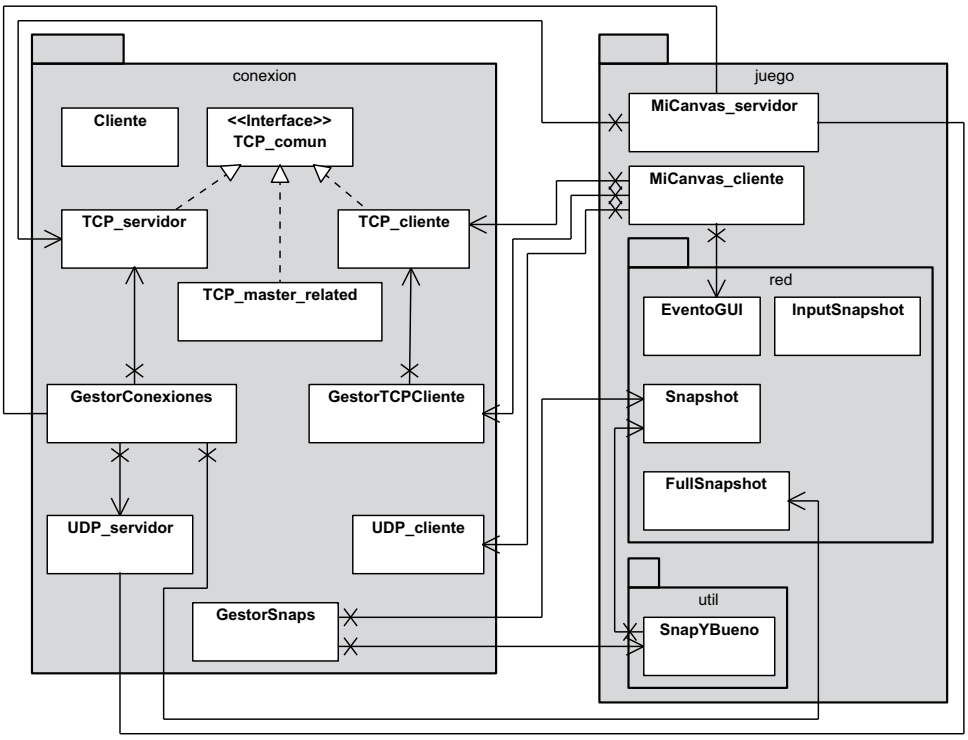


Figura B.26: Clases del módulo de gestor de conexiones (1 de 2)

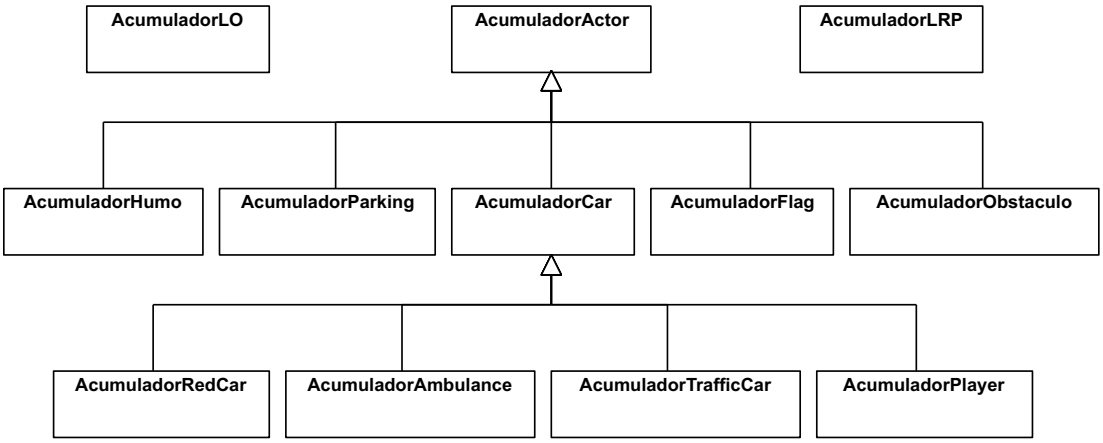


Figura B.27: Clases del módulo de gestor de conexiones (2 de 2)

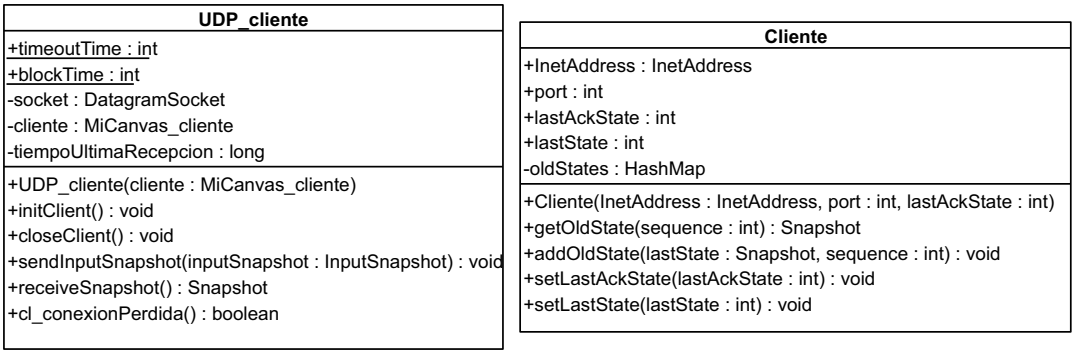


Figura B.28: Detalle de las clases «UDP_cliente» y «Cliente»

partida y de proporcionarles el estado inicial de la partida (*FullSnapshot*).

Respecto a las clases que se usan durante la partida, «FullSnapshot», «Snapshot» e «InputSnapshot» contienen la estructura de los paquetes de datos transmitidos entre clientes y servidor.
La primera contiene el estado inicial del juego, es decir, los elementos estáticos que no van a cambiar durante el trascurso de la partida y que por lo tanto solo serán necesarios de enviar cuando el jugador se conecte.
La segunda contiene el estado actual del juego: los elementos dinámicos que pueden variar (p.ej. los vehículos o los objetivos).
La tercera contiene los eventos de teclado recogidos en el cliente y que deben transmitirse al servidor en cada ciclo.

La clase «SnapYBueno» es una simple estructura formada por un «Snapshot» y un valor booleano, que indica si es correcto según el sistema de control de flujo. Se usa en el cliente para almacenar los *snapshots* pendientes de procesar. La clase «GestorSnaps» es la encargada de realizar este control de flujo de paquetes.

La clase «EventoGUI» contiene la estructura utilizada para los mensajes que se muestran en pantalla al jugador cuando ocurren ciertos eventos, y contiene también los métodos necesarios para su creación.

Dentro de las clases usadas durante la partida también se encuentran los *acumuladores*, que son clases creadas para solucionar el problema causado por el envío de únicamente actores cercanos (ver anexo C.6.5). Estas clases están contenidas en el paquete «juego.red.acumuladores» y se dispone de una clase por cada tipo de objeto que se ha de enviar solo en ciertas ocasiones.

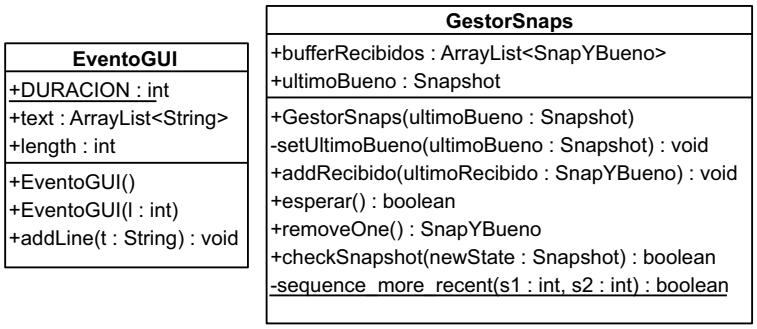


Figura B.29: Detalle de las clases «GestorSnaps» y «EventoGUI»

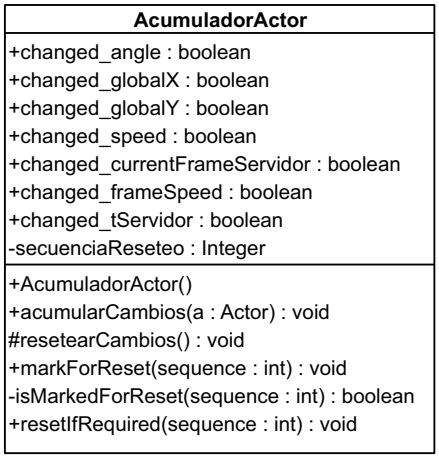


Figura B.30: Detalle de una clase acumulador

B.4.11. Módulo de física

El módulo de física contiene las funciones y estructuras que permiten realizar tareas como la detección de colisiones.

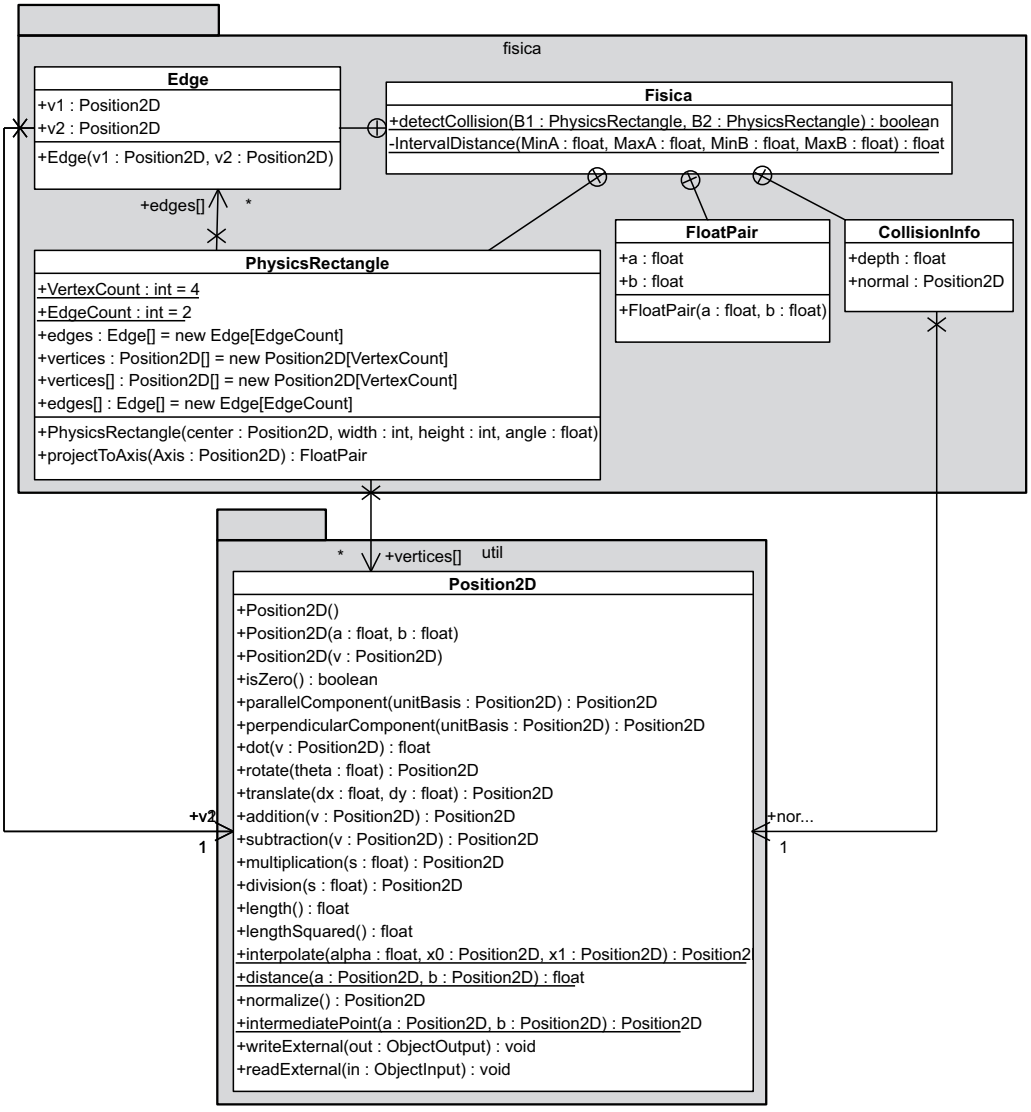


Figura B.31: Clases del módulo de física

La clase «Fisica» realiza la detección de colisiones entre actores (no con elementos del terreno) y hace uso de la clase «PhysicsRectangle» que contiene los ejes y vértices que representan al actor y tiene funciones para proyectar dicha representación sobre los ejes (necesario para calcular la colisión según el teorema *Separating Axis theorem* (ver anexo C.4.2).

B.4.12. Módulo de inteligencia artificial

El módulo de inteligencia artificial realiza los cálculos de path-finding (paquete «astar») y del control de la conducción del vehículo.

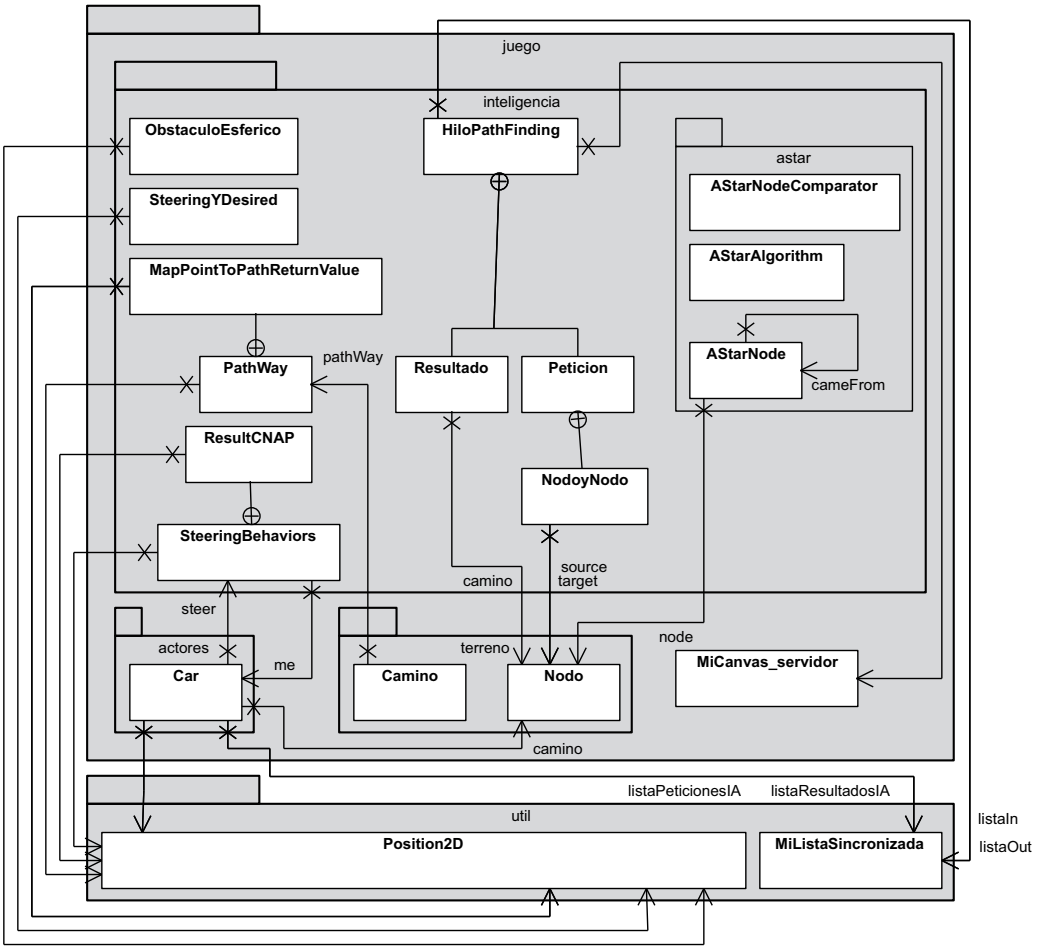


Figura B.32: Clases del módulo de inteligencia artificial

Dentro del paquete «astar» se encuentra la clase «AStarAlgorithm» que, mediante el uso de las clases «AStarNode» (representación de un nodo) y «AStarNodeComparator» (comparador de nodos), realiza una búsqueda $A*$ sobre los nodos que forman los caminos del terreno para encontrar la ruta más corta entre dos puntos.

Ésta clase es llamada desde la clase «HiloPathFinding», que tiene un hilo de

ejecución propio y recibe a través de la clase «MiListaSincronizada» peticiones (clase «Peticion») de path-finding, las computa y cuando el resultado está listo devuelve el resultado (clase «Resultado») a través del mismo medio.

Respecto al control de la conducción, las clase más importante de las que intervienen es «SteeringBehaviors», que contiene los métodos invocados desde el vehículo para la realización de los diferentes comportamientos indicados en [17] y en su mayor parte desarrollados en *OpenSteer*[2] (ver C.5.1).

«SteeringYDesired» es una estructura donde se tiene el vector que indica el giro a realizar junto al vector que indica la dirección a la que se encuentra el objetivo.

«ObstaculoEsferico» contiene una posición en dos dimensiones y un radio y se asocia con las entidades que deben ser esquivadas por los diferentes comportamientos desarrollados.

Por último «PathWay», «MapPointToPathReturnValue» y «ResultCNAP» son clases que se han obtenido de la implementación de la librería *OpenSteer* y son usadas por los métodos que implementan los diferentes comportamientos de control del vehículo.

B.4.13. Módulo cliente

Este módulo se compone de todos los elementos que intervienen en la parte cliente de la aplicación y no encajan en el resto de módulos.

En la figura B.33 se da una vista general de la interacción de la clase «MiCanvas_cliente» con el resto de clases del módulo y también las relaciones con el resto de módulos que no hayan sido vistas todavía.

La clase principal es «MiCanvas_cliente» y contiene métodos para el pintado del mundo de juego y la interfaz gráfica de usuario, el tratamiento de red (incluyendo la interpolación, extrapolación y predicción), la captura de eventos del teclado, el manejo de las entidades locales, las acciones realizadas en el cliente para liberar de gasto de procesamiento al servidor (cálculo de aparcamientos cercanos, etc.).

Dentro de este módulo hay muchas otras clases de apoyo, la mayoría en el paquete «juego.otros».

Una de estas clases es «RadarVespa», que permite representar el radar y contiene métodos para pintar en él diversos tipos de elementos.
Una clase asociada a ésta es «MiniMapEvent», que tiene la representación de los eventos proporcionados por VESPA con la mínima información necesaria (ya que será transmitida a través de la red).

La clase «Flecha» contiene la estructura y funciones necesarias para calcular y mostrar en pantalla las flechas que indican la posición de los objetivos y enemigos respecto del jugador.

[2] `http://opensteer.sourceforge.net/`

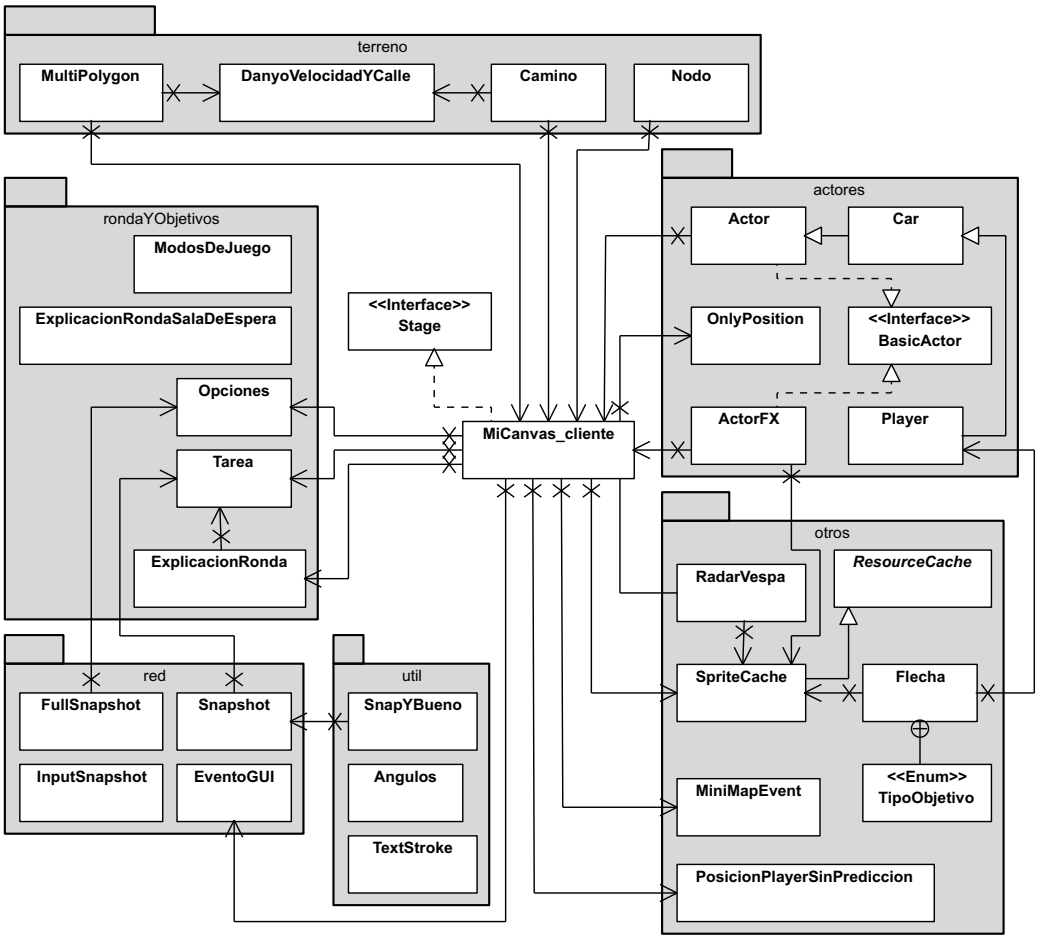


Figura B.33: Clases del módulo de cliente

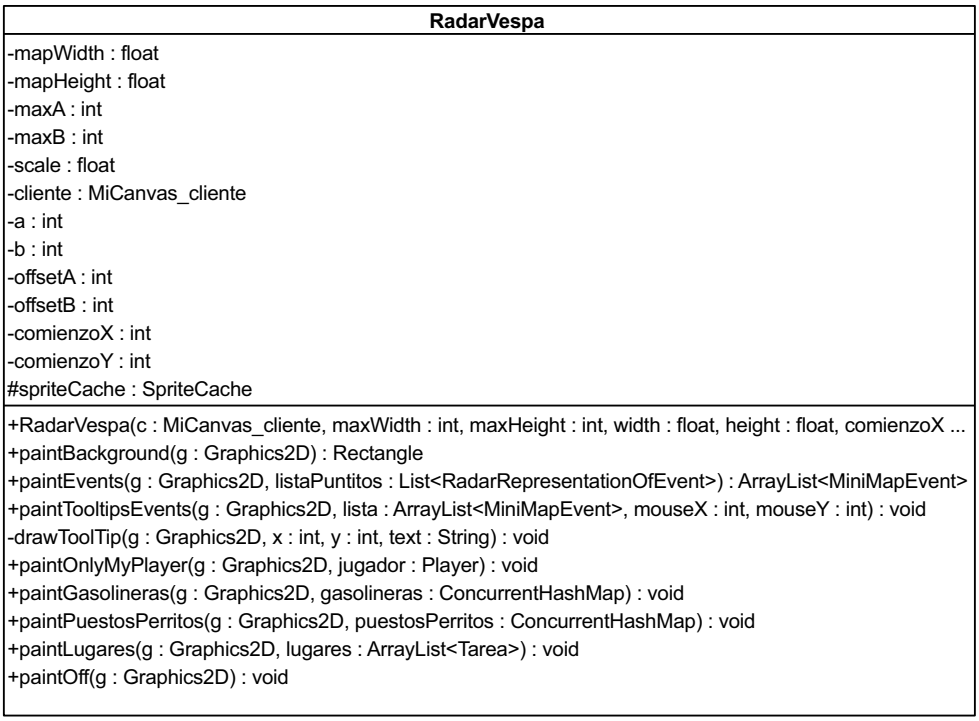


Figura B.34: Detalle de la clase «RadarVespa»

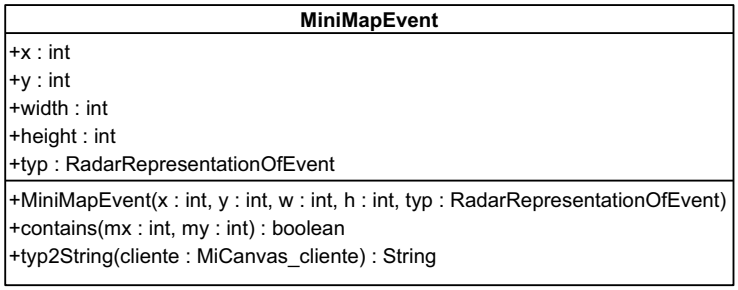


Figura B.35: Detalle de la clase «MiniMapEvent»

Figura B.36: Detalle de la clase «Flecha»

B.4.14. Módulo servidor

Este módulo se compone de todos los elementos que intervienen en la parte servidor de la aplicación y no encajan en el resto de módulos.

En la figura B.37 se da una vista general de la interacción de la clase «MiCanvas_servidor» con el resto de clases del módulo y también las relaciones con el resto de módulos que no hayan sido vistas todavía.

La clase principal es «MiCanvas_servidor», que contiene métodos para el tratamiento de red, inicialización del mundo de juego, cálculo de puntuaciones e inicialización y finalización del servidor.

Esta clase interactúa con el resto de módulos previamente citados.

B.5. Game Loop (bucle de juego)

El *game loop* es una secuencia presente en todos los juegos que generalmente consiste en obtener los comandos del jugador, actualizar el estado del juego, realizar las tareas de la IA, reproducir los efectos de sonido y pintar el juego.[3] Esta secuencia se ejecuta infinitas veces hasta que se acabe la partida.

[3] `http://www.koonsolo.com/news/dewitters-gameloop/`

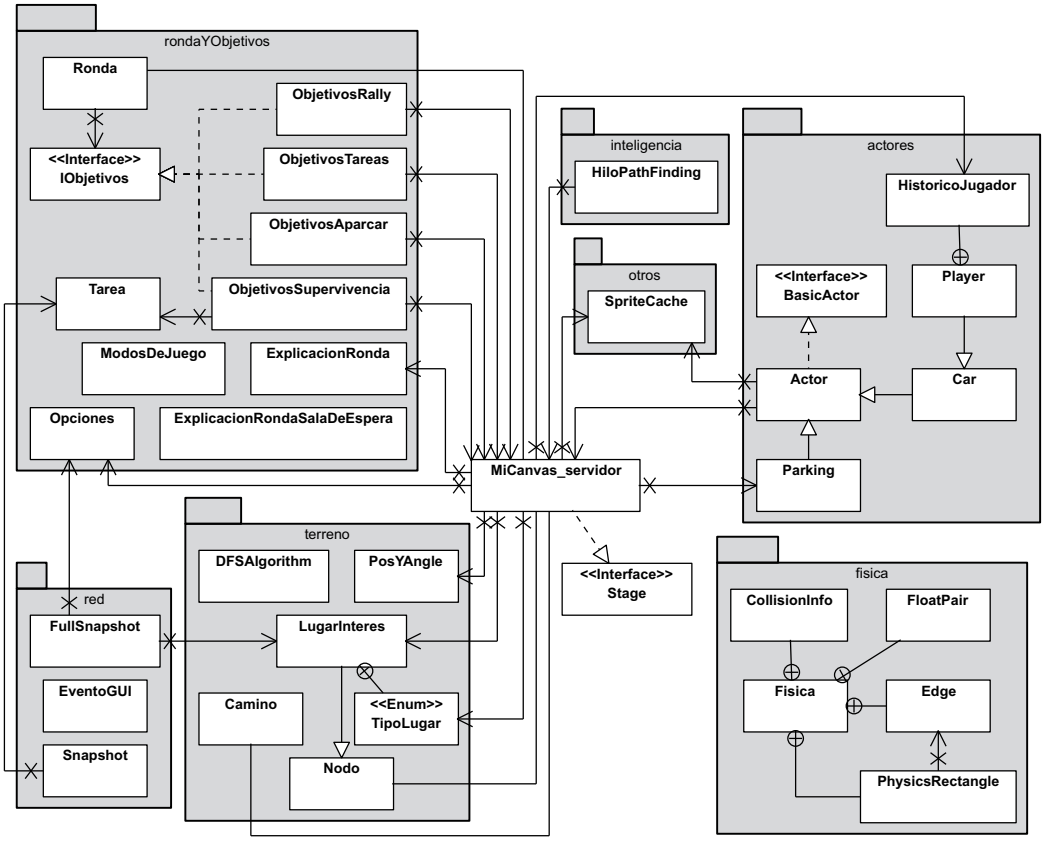


Figura B.37: Clases del módulo de servidor

En `http://www.koonsolo.com/news/dewitters-gameloop/` se muestran varios posibles diseños del *game loop*, de los cuales se escogió realizar el primero de ellos: *FPS dependientes de la velocidad del juego (constante)* por ser una solución fácil de implementar y que mantiene la sencillez del código, lo cual es importante ya que al tratarse de un juego en red y como también se van a introducir conceptos como la predicción e interpolación, se va a complicar mucho el código . Además, contando con que todos los ordenadores implicados (servidor y clientes) tienen potencia suficiente para conseguir los *FPS* establecidos, la velocidad en todos ellos será la misma.

En el Algoritmo B.1 se observa el código básico tanto en el cliente como en el servidor. En *actualizar juego* se realiza todo lo necesario para actualizar el mundo de juego (distinto según si es el cliente o el servidor) y en *mostrar objetos en pantalla* (solo lado cliente) se pinta el mundo de juego en pantalla. En las siguientes secciones (Anexos B.5.1 y B.5.2) se explica en detalle el contenido de estas dos funciones así como también de *inicialización* y *finalización*.

Algoritmo B.1: *Game loop*

```
inicializacion
while( el juego continua )
{
  actualizar juego
  mostrar objetos en pantalla <-- Solo en el cliente
}
finalización
```

B.5.1. Servidor

El servidor tiene el siguiente esquema de funcionamiento (ver Figura B.38):

***inicialización*:** se crea el escenario (nodos, caminos, aparcamientos, gasolineras...) y después se crea el *gestor de conexiones*, el cual es el encargado de recibir las peticiones de conexión de los clientes y enviarles el estado de juego actual para que se puedan unir. Esta fase de inicialización acaba cuando el primer cliente se une a la partida.

***actualizar juego*:** recibe los comandos de los jugadores (enviados mediante *UDP*) y los procesa. Después llama al *gestor de rondas* para que compruebe el estado de los objetivos y dictamine si se debe avanzar de ronda o si el juego se ha acabado. Finalmente envía a los clientes los datos actualizados de los

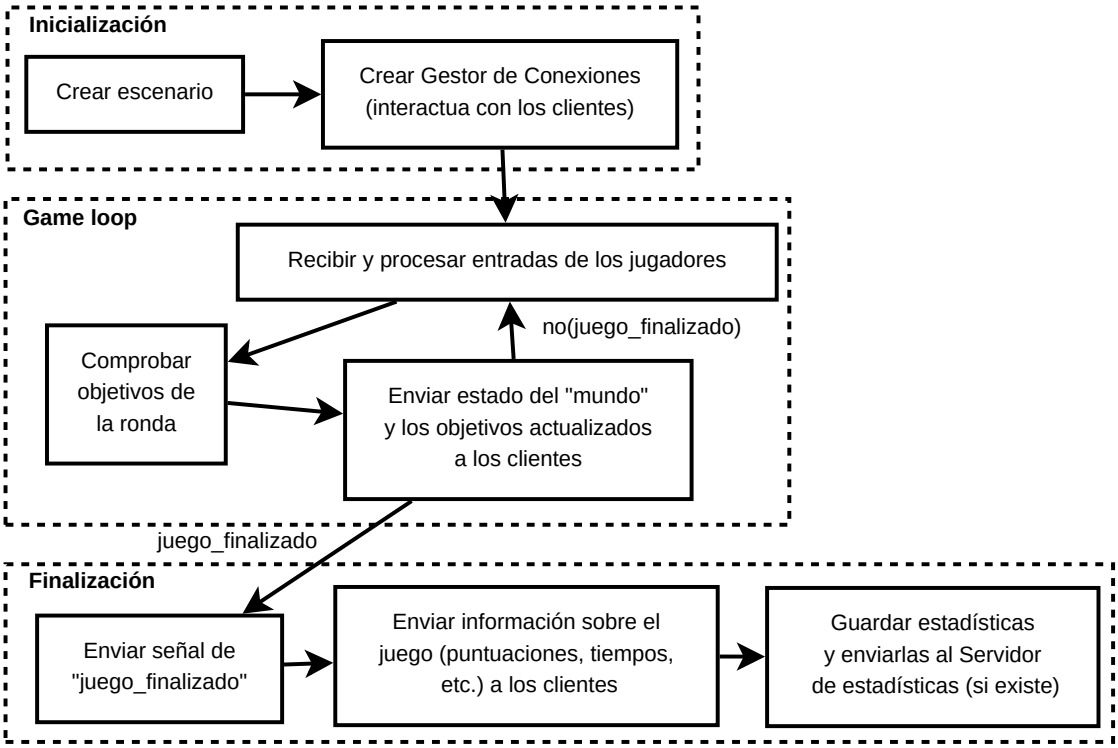


Figura B.38: Funcionamiento del servidor

actores en sus cercanías y, en el caso de haberse modificado, los objetivos de la ronda.

***finalización*:** se envía a los clientes una señal de que el juego ha acabado, y cuando se ha recibido el *ack* de todos ellos, se les envía las estadísticas finales de la partida (puntuación, tiempo...). Finalmente, si la configuración actual así lo requiere, se guardarán en ficheros las estadísticas de explotación recogidas, y si el servidor de recogida de estadísticas está operativo, también se le enviarán a él (ver Capítulo 3).

Hay que destacar que al tener cada actor un hilo de ejecución propio, la actualización de su estado no se realiza dentro de la lógica del hilo del servidor sino que se realiza de forma asíncrona.

Otro aspecto importante es que a cada cliente no se le envían los datos de todos los actores sino solamente de aquellos que por su cercanía tengan interés para él. Se profundizará más en este aspecto en el Anexo C.6)

B.5.2. Cliente

El cliente tiene el siguiente esquema de funcionamiento (ver Figura B.39):

***inicialización*:** primero se inicia la música de la partida (la del menú se ha finalizado al crear la clase *cliente*). Después se crea el *gestor de conexión TCP*, que es el que iniciará la petición de conexión al servidor. Se enviarán y recibirán todos los datos requeridos para unirte a la partida y se procesarán para que el estado de juego sea el mismo que del servidor.

***actualizar juego*:** se recogen de teclado los comandos de los jugadores y se envían al servidor. Se recibe del servidor el estado actualizado del juego y se almacena en un *buffer*. Se extrae el estado más antiguo de los almacenados en el buffer (se eliminan después de extraerlos) y se le aplica la descompresión delta para obtener los datos que contiene. Después se aplica la predicción e interpolación (ver Anexo C.6), se hace la comprobación de colisiones de los *actores FX* (son aquellos que solo existen en el cliente por representar efectos visuales o sonoros) y se actualiza su estado.

***mostrar objetos en pantalla*:** se pintan todos los elementos del juego en pantalla.

***finalización*:** se detiene la música y los efectos de sonido, pintas la animación de fin de la partida y esperas a recibir las estadísticas finales de la partida. Finalmente cargas las pantallas de los menús, mostrando inicialmente la pantalla de estadísticas o la de error en caso de que haya ocurrido alguno.

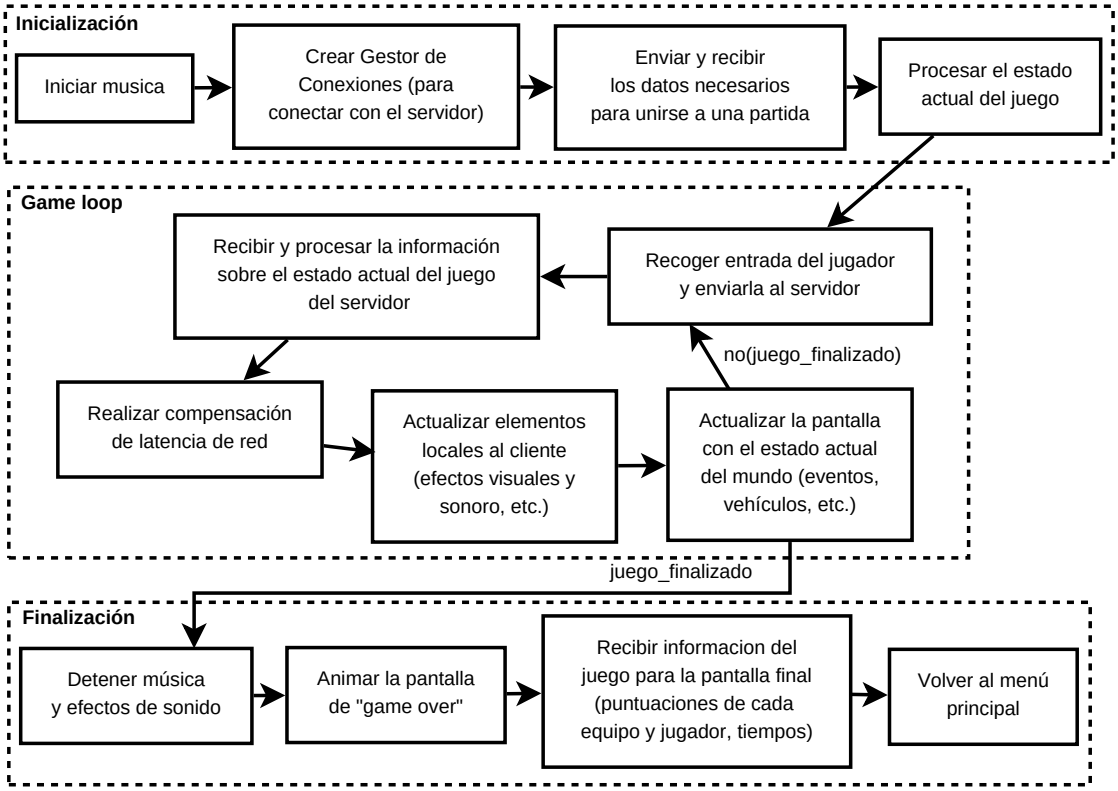


Figura B.39: Funcionamiento del cliente

B.5.3. Actor

Cada actor tiene el siguiente esquema de funcionamiento (ver Algoritmo B.2): Tiene un bucle similar al *game loop* del cliente y servidor, que finalizará cuando el actor muera o se acabe la partida, y que contiene un método *actualizar* al que se le llama una vez por ciclo (la misma velocidad que el bucle del servidor) y cuando se finaliza el bucle se llama a la función *eliminarse*, que elimina al actor de las listas en las que está incluido y si era uno de los objetivos de la ronda lo da por completado, además según su clase también detiene el hilo de path-finding y los de VESPA.

Algoritmo B.2: Bucle del actor

```
while(vivo & no(partida acabada))
{
  actualizar
}
eliminarse
```

El método *actualizar* tiene el funcionamiento que se puede ver en el Algoritmo B.3, y básicamente consiste en comprobar si el actor sigue vivo (se comprueba al comienzo de cada ciclo) y si lo está se llama a las funciones *actualizar* y *comprobar colisiones* que realizan las acciones del actor y comprueban si existe alguna colisión con otro actor respectivamente.

Algoritmo B.3: Método *actualizar* del actor

```
if (está marcado para eliminación)
{
  vivo = falso
}
else
{
  actuar
  comprobar colisiones
}
```

Este bucle del actor se ejecuta en el servidor, ya que en el cliente solo se reciben los datos calculados. Sin embargo, existen ciertos aspectos de los actores (normalmente relacionados con su pintado) que se calculan en el mismo cliente, por no necesitar un resultado que sea consistente en todos los clientes. Estas acciones

se ejecutan desde la función *actuar FX*. Ejemplos de estas acciones son los cambios de luces en la sirena de la ambulancia o el cambio de *sprite* del humo.

Existen dos tipos de actores: los actores de la clase *Actor* (p.ej. vehículos, humos, gasolineras...), que son los actores que tienen que existir en todos los clientes con su estado sincronizado con el servidor, y los actores de la clase *ActorFX*, que representan efectos visuales o sonoros y solo existen en el cliente que necesita representarlos (p.ej. explosiones, símbolos de mareo al colisionar con el humo...). Ambos tipos implementan el interfaz *BasicActor* ya que es la que usa el sonido (que puede ser generado por actores de ambas clases) (ver Figura B.40). El bucle de los *actores FX* es igual al de los actores normales exceptuando que no llaman a la función *comprobar colisiones*, ya que no tienen consistencia física (de tenerla dejarían de ser *actores FX* y necesitarían ser actores normales para que el servidor sincronice su estado con todos los clientes).

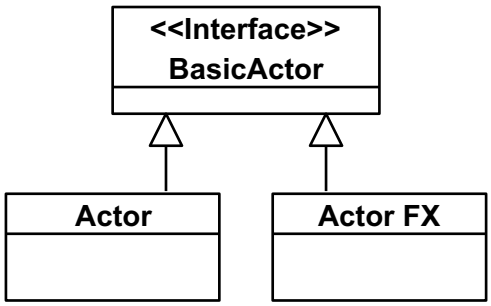


Figura B.40: Las clases *Actor* y *ActorFX* implementan la interfaz *BasicActor*

B.6. Hilos de ejecución

En esta sección se van a explicar que hilos existen durante la ejecución y cómo están comunicados unos con otros. Los diferentes hilos de ejecución se pueden dividir en hilos de la interfaz, hilos del cliente e hilos del servidor.

El esquema de ejecución es el siguiente (ver Figura B.41):
Cuando se inicia el juego, se crea la clase *VentanaPPal* (encargada de cargar la configuración almacenada y crear los directorios si es necesario), la cual termina pasando el control a la clase *Contenedor*, que es en la que se muestran las pantallas de los menús. La clase *Contenedor* crea un hilo adicional para el gestor de música

(clase *MiPlayer*), y un número de hilos determinado (dependiente de la configuración de audio) para el gestor de sonidos FX (clase *SoundManager*).
Cuando desde el menú se elija la opción de crear una partida nueva, el hilo principal creará la clase *Cliente* y se creará un hilo adicional con la clase *Servidor*. Además, el hilo del gestor de música llegará a su final y se creará uno nuevo con la música deseada (la forma de cambiar de canción es cerrar el hilo y crear uno nuevo).
Cuando la opción de menú elegida no sea crear una nueva partida sino unirse a una partida existente, se omitirá la creación del servidor y su hilo.

Dentro del cliente, se creará un nuevo hilo en el que se ejecutará el gestor de conexiones TCP (clase *GestorTCPcliente*), y cuando la partida llegue a su fin, el hilo principal (clase *Cliente*) enviará señales de interrupción a este hilo así como a todos los demás dependientes del cliente (gestor de sonidos y gestor de música). Finalmente, una vez interrumpidos todos, el cliente volverá a invocar al menú principal (clase *Contenedor*) sobre su mismo hilo.

Por su parte el servidor (ver Figura B.42) tiene como similitud la creación de un gestor de conexiones TCP (clase *GestorConexiones*), pero no tiene hilos de sonido ya que es una parte que se ejecuta únicamente en el cliente.
Además, a diferencia del cliente, en el servidor cada actor tiene su propio hilo de ejecución, los cuales seguirán funcionando hasta que el actor sea eliminado de la partida (por ejemplo por haberse destruido) o el valor del servidor que indica si la partida ha finalizado se vuelva cierto.
Algunos actores a su vez crean nuevos hilos, caso del hilo de *path-finding* y los hilos de VESPA (dependientes de la implementación, ver capítulo D.1), siendo los actores los responsables de enviarles una señal de interrupción cuando el hilo del actor vaya a ser eliminado.
A diferencia del cliente, que al acabar devuelve el hilo de ejecución al menú principal, el servidor al finalizarse termina su hilo de ejecución.

La comunicación entre los diferentes hilos, más allá del envío de interrupciones ya citado, se realiza siempre sobre variables comunes, utilizando sincronización de bloques.

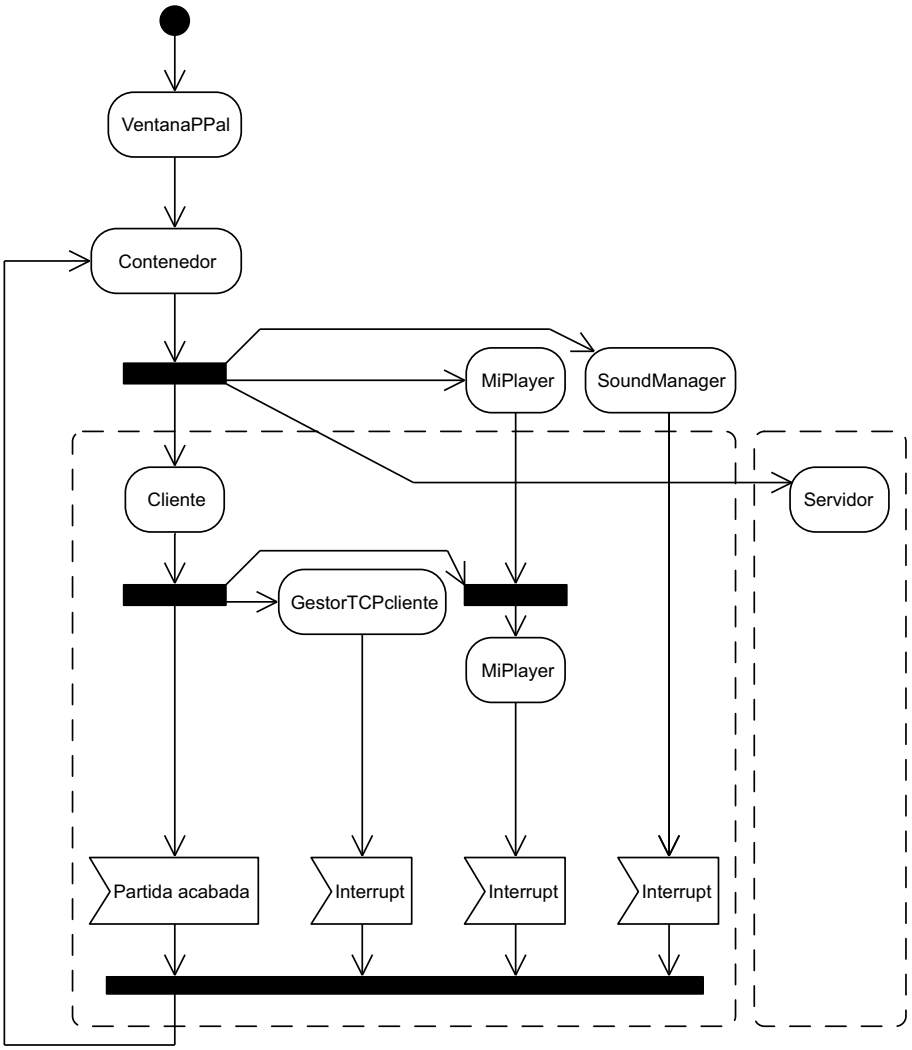


Figura B.41: Vista general de los hilos de ejecución

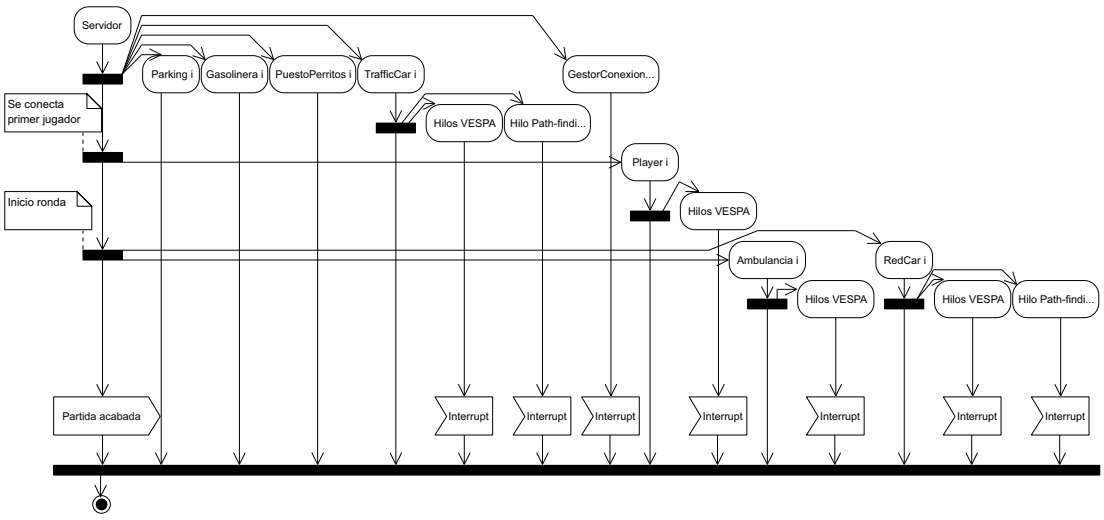


Figura B.42: Detalle de los hilos de ejecución del servidor

Anexo C

Sobre el videojuego

En este anexo se tratarán en detalle todos aquellos aspectos sobre el desarrollo del videojuego que no han podido ser tratados en el capítulo 2 o han sido tratados de forma resumida.

C.1. Menús del juego

En esta sección se verán todas aquellas cuestiones que por motivo de espacio no pudieron ser explicadas en el capítulo correspondiente (2.3).

C.1.1. Tipografía

Los menús del juego se diseñaron para utilizar la tipografía *OCR-B 10BT*, dado que ésta poseía el aspecto idóneo para la imagen que se buscaba transmitir. Sin embargo, debido a los derechos de autor, no es posible redistribuir dicha fuente tipográfica, y además incluso dándose la posibilidad de redistribuirla, no todos los usuarios desearían instalarla como requisito previo para jugar. Es por esto que al ejecutar el juego, si se detecta que dicha fuente está instalada en el sistema, se usará, sin embargo si no se encuentra instalada, Java elegirá una fuente alternativa (de la misma familia a ser posible) para sustituirla.

Por esta razón, y al ser la fuente escogida poco común, es probable que la mayoría de usuarios vean las pantallas de menús con una apariencia distinta a la diseñada. Para evitar esto, se pensó que una forma de solucionar este problema sería sustituir todos los textos por imágenes de dichos textos en la tipografía deseada, pero se descartó por la complicación que supone y por el inconveniente de tener tantas imágenes cargadas en memoria.
En lugar de esto, finalmente se decidió únicamente comprobar que no existan errores de diseño con la tipografía *Arial*, que es la tipografía sustituta en los tres

sistemas operativos para los que se diseñó el juego: *Mac OS*, *Windows* y *Ubuntu*.

C.1.2. Directorio del juego

La gran mayoría de los juegos requiere guardar distintos parámetros y configuraciones. Como ellos, `Vanet-X` usa un directorio de juego en el que se almacenan los mapas descargados y la configuración de los diferentes parámetros de juego.

Siempre que se inicia la aplicación se comprueba si existe el directorio de juego por defecto y en caso contrario crea el sistema de archivos necesario, mostrándose una ventana emergente en la que se informa al usuario que directorio se va a usar (y en el caso de que se haya creado nuevo, indicándoselo).
El directorio de juego por defecto está situado en la carpeta por defecto del usuario proporcionada por el sistema operativo. En sistemas Windows esta carpeta es *Mis Documentos*, mientras que en sistemas Unix se sitúa en el directorio *$HOME*.

Aunque siempre se inicia el juego con el directorio por defecto, en los menús se puede cambiar cual es el directorio en uso. Al cambiar de directorio se reinicia el interfaz por lo que se cargan los valores almacenados en los ficheros de configuración del fichero elegido.
Si el directorio elegido estaba vacío, se crea el sistema de archivos con los valores de configuración por defecto.

El sistema de archivos tiene la siguiente estructura (Figura C.1):

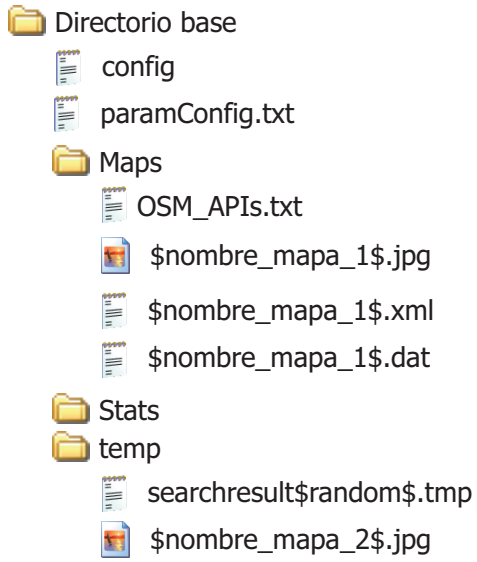


Figura C.1: Estructura del directorio de juego

- En el directorio base el fichero *config*, en el que se guardan todos los parámetros modificables mediante las pantallas de menú, y el fichero de texto *ParamConfig.txt*, que contiene otros parámetros, modificables por el usuario, relacionados con el rendimiento del juego y opciones de habilitación de diversas características como las estadísticas (ver Figura C.2).

```
#radioMaxMapa
0.0050
#radioPeqMapa
0.0030
#MaxBytesUDP
6000
#MaxExtrapolation
5
#MaxInterpolation
2
#NUM_PARKINGS_FIJOS
30
#NUM_PARKINGS
15
#NUM_FX
20
#timeoutWaitingPlayers
60
#framesPropagacionExplosion
5
#maxTC
50
#maxRC
8
#maxNodos
10000
⋮
⋮
#WOLFSON (ON=1)
0
#maxPlayers
8
#cochesBuscandoParking
30
#TIEMPO_CAMBIO
20000
#MARGEN_TIEMPO_CAMBIO
20000
#RADIO_PARKING_VALIDO
500
#PARKING_PROTOCOL (0=Time,1=Distance,2=EP,3=Null)
3
#DISTANCIA_CERCA
7000
#debugRadar (ON=1)
1
#VESPA_StatisticsOn (ON=1)
1
#Parking_Player_StatisticsOn (ON=1)
1
#Parking_Traffic_StatisticsOn (ON=1)
1
#Game_StatisticsOn (ON=1)
1
```

Figura C.2: Ejemplo de contenido del fichero «*ParamConfig.txt*»

- Una carpeta *Maps*, en cuyo interior se guardan los ficheros *.dat*, *.jpg* y *.xml* en los que se contienen los datos de cada mapa descargado, y también un fichero de texto *OSM_APIs.txt* (ver Figura C.3) que contiene las *url* de las *APIs* de descarga de mapas, para que el usuario pueda modificarlas.

```
http://jxapi.openstreetmap.org/xapi/api/0.6/map?bbox=
http://www.overpass-api.de/api/xapi?map?bbox=
```

Figura C.3: Ejemplo de contenido del fichero «*OSM_APIs.txt*»

- Una carpeta *Stats* en la que se guardan las estadísticas recogidas durante las partidas. En la sección D.2.3 se ve detalladamente los elementos de su interior.

- Una carpeta *temp* que contiene los ficheros temporales creados durante la ejecución, como son los resultados de las búsquedas y las previsualizaciones de los mapas no descargados, y que se eliminan automáticamente al cerrar la aplicación.

C.1.3. Prevención de errores

Se han implementado diversos métodos para impedir acciones en los menús que puedan causar errores en la ejecución del juego. El primero de ellos es la comprobación de los campos de texto.

Tanto al pulsar los botones de navegación de avanzar de pantalla y guardar cambios, como cuando un campo de texto pierde el *foco*, se hace uso de la clase *javax.swing.InputVerifier* para comprobar si los campos de la pantalla actual son correctos. En caso negativo el campo incorrecto se marca con fondo rojo y en algunos casos también se crea una ventana emergente avisando de dicha incorrección.

Algunas acciones tienen unos tiempos de espera considerables, y por ello en esos casos usando la clase *javax.swing.SwingWorker* se desligó la acción del hilo *EDT* (*Event Dispatch Thread*) para que no lo bloquease y no diese la apariencia de que la interfaz se ha quedado «congelada». Esto tiene el peligro de que cómo se tiene el control de la interfaz mientras se ejecutan las acciones deseadas, el usuario puede realizar acciones cuya ejecución antes de haber terminado la ejecución de la otra acción en curso puede causar errores.
Es por ello que mientras exista una ejecución en curso, se bloquean los botones que podrían causar comportamientos indeseados si son pulsados a la vez. Un ejemplo sería pulsar el botón de unirte a una partida y mientras esperas a que se ejecute volverle a pulsar (lo cual sería muy común debido a la característica impaciencia de la mayoría de usuarios).

Otro posible error es iniciar una partida sin tener seleccionado ningún escenario (por no tener ninguno descargado), posible error que se previene dirigiendo en ese caso al jugador a la pantalla *Configuración avanzada de mapas* para que se descargue un mapa.

Por último, existe un posible error que solo puede sucederle al creador de la partida, y que consiste en después de finalizar una partida, crear rápidamente otra nueva, sin dejar tiempo a que el «lado servidor» de la aplicación termine de enviar las estadísticas al resto de jugadores y finalice, en cuyo caso en la nueva partida aparecería un error indicando que los puertos escogidos están ya asociados.
Para evitar este problema, se establece un tiempo mínimo desde que abandonas

una partida hasta que el botón para crear una nueva responde a los eventos del usuario.

C.1.4. Pantallas de error

Cuando aparece un error crítico durante una partida, que hace que se tenga que finalizar y volver al menú, no solo se imprime la traza del error por la salida de error por defecto, sino que también se crea una pantalla de error, anterior a la pantalla de estadísticas, en la que se imprimen las diferentes trazas de error diferenciadas por pestañas según el elemento que las haya causado.

Si por el contrario, aparece un error pero no es crítico, se continúa jugando la partida pero se muestra un mensaje permanente en la esquina inferior izquierda de la pantalla avisando al jugador de la existencia de ese error y de que por esa razón pueden darse comportamientos extraños. Cuando finalmente el jugador abandone la partida, también aparecerá la pantalla de error anteriormente citada con los errores y los elementos en los que han tenido lugar.

También existe otra pantalla de error diferente que aparece en los casos en los que ha habido un error al intentar unirte a una partida.
Este error puede tener diferentes causas (la partida está completa, no existe, las versiones de juego del servidor y el cliente no coinciden...) y es por ello que esta pantalla tiene una zona de texto cuyo contenido es modificable dinámicamente y, en función de la causa del error de conexión, se mostraran diferentes textos.

C.1.5. Pantallas de mapas

En la pantalla *Configuración avanzada de mapas*, además de poder añadir y eliminar mapas, se ha elaborado una opción para poder previsualizar el mapa seleccionado o el área que se desea descargar. Esta previsualización se realiza obteniendo de una *API* la imagen en la que se representa el área seleccionada.
Esta imagen se presenta en tamaño reducido en dicha pantalla, y pulsando sobre ella se accede a una nueva pantalla en la que se visualiza la imagen a un mayor tamaño. Mientras que el tamaño reducido siempre es el mismo, en la pantalla de ampliación la imagen no se amplía mas allá de su tamaño original, solamente se reduce en caso de que sea necesario para que encaje en el espacio disponible.

Cuando se desea previsualizar un mapa descargado, como la vista previa se ha descargado a la vez que los datos *OSM XML*, no hay más que elegir esa imagen y mostrarla.
Sin embargo, cuando se desea previsualizar el área de la búsqueda actual, cómo existe un control deslizante para ajustar el tamaño a descargar, la forma de no tener que pedir una nueva imagen a la *API* cada vez que se varíe el tamaño seleccionado

es descargando la imagen con el tamaño máximo que se permite seleccionar y luego mediante una función de recorte, mostrar únicamente el área proporcional al tamaño elegido actualmente.

C.1.6. Otros aspectos importantes

Un aspecto importante de la implementación es que todas las pantallas de los menús se cargan en memoria al inicio de la ejecución, de forma que se evita tiempo de espera por carga cuando se navega entre los menús.
Cuando se inicia una partida, se descargan de memoria, ya que no van a ser usadas hasta que la partida finalice, momento en que se volverán a cargar.

Otro elemento importante de la implementación es que se ha añadido un número de versión al juego para que cuando te conectes a un servidor se compruebe que ambos funcionan bajo la misma versión de la aplicación. Esta comprobación se realiza inicialmente al pulsar el botón *Unirte* de la pantalla *Unirte a una partida existente*, ya que cuando se pide al servidor la información de la partida que se mostrará en la pantalla *Sala de espera*, la cual contiene información de la partida en curso y los jugadores conectados, también se aprovecha para comprobar que las versiones del juego sean idénticas.

La comprobación no solo se realiza en este momento, sino que se comprueba en todas las ocasiones en las que se vuelve a realizar alguna petición al servidor antes de unirte (botones *Actualizar* y *Unirte* de la pantalla *sala de espera*). Esto es así ya que, aunque es una posibilidad muy remota, podría darse el caso de que mientras esperas en la sala de espera, el servidor haya finalizado la partida, y se cree otra nueva en la misma dirección IP con otra versión diferente del juego.

También hay que anotar que en la sala de espera, en el caso de tratarse de una partida competitiva, se pide al usuario seleccionar el equipo al que desea unirse, estando siempre seleccionado por defecto el equipo que tiene menos jugadores, lo cual se ha podido calcular gracias a la información que el servidor ha proporcionado a petición del usuario.

Otro aspecto adicional es que durante el desarrollo del juego se requería poder cambiar diversos parámetros desde el mismo juego evitando así la necesidad de recompilar cada vez. Estos parámetros no deberían ser modificables en la versión final, ya que principalmente era para probar ciertos aspectos que estaban todavía en fase de pruebas. Una vez acabadas dichas pruebas se decidió que era buena idea mantenerlos, y por ello se ideó una pantalla que estuviese oculta y sólo se mostrase con fines de desarrollo.

Para esta tarea se ideó usar el *Konami Code*[1], que es usado en muchos videojuegos y sitios web[2] para acceder a trucos o características ocultas.

También es importante destacar que el diseño de los menús se ha realizado usando el *layout* nulo ya que como la ventana de la aplicación tiene un tamaño fijo, los componentes no variarán de posición ni de tamaño, y por esa razón se ha considerado innecesario usar otro *layout* más complejo.

C.2. Obtención de mapas

Como se ha explicado en el capítulo 2.4, para cumplir el segundo y tercer objetivo del Proyecto Fin de Carrera («desarrollar lo necesario para que se compita en escenarios creados a partir de datos reales obtenidos de algún sistema que proporcione mapas de carreteras» y «desarrollar una funcionalidad de descarga de mapas, de forma que introduciendo la localización en la que deseas jugar se descargue una porción de mapa alrededor del punto elegido») se escogió el servicio OpenStreetMap, por ser gratuito y tratarse de un proyecto colaborativo con un uso en expansión.

En esta sección se mostrarán los detalles del estudio del sistema OpenStreetMap, la implementación realizada y los problemas que se encontraron durante dicha implementación.

C.2.1. OpenStreetMap

Según la correspondiente entrada de la wiki de OpenStreetMap[3], existen varias fuentes donde conseguir los datos, los cuales se encuentran en formato *OSM XML*[4]. Estas fuentes son la propia API principal[5], Xapi (OSM Extended API)[6] y Overpass API[7], cuyas diferencias se explican a continuación:

API principal se trata del método de acceder a los datos usado por las aplicaciones que requieren capacidades de edición, ya que es el único método que no se realiza con acceso de solo lectura. Para evitar su sobrecarga, tiene la descarga de datos limitada a áreas menores de 0,25 grados cuadrados y se

[1] `http://en.wikipedia.org/wiki/Konami_Code`
[2] `http://konamicodesites.com/`
[3] `http://wiki.openstreetmap.org/wiki/Databases_and_data_access_APIs`
[4] `http://wiki.openstreetmap.org/wiki/OSM_XML`
[5] `http://wiki.openstreetmap.org/wiki/API`
[6] `http://wiki.openstreetmap.org/wiki/Xapi`
[7] `http://wiki.openstreetmap.org/wiki/Overpass_API`

recomienda que las aplicaciones que no tengan capacidades de edición usen los otros métodos en lugar de éste.

Xapi es un protocolo de API de solo lectura, muy similar a la API principal. Tiene diversas mejoras de rendimiento y tiene un mayor límite de descarga de datos: 10 grados cuadrados. Los datos que devuelve son compatibles con el protocolo de la API principal.

Overpass API al igual que Xapi, es un protocolo de API de solo lectura, ideado no para la edición sino para el consumo de datos. Su principal diferencia de los otros dos métodos es que posee un poderoso lenguaje de consulta aunque tiene una capa de compatibilidad con las consultas de Xapi.

A diferencia de la API principal, cuyo acceso se realiza siempre sobre la misma URL, los otros dos métodos tienen implementaciones funcionando en varios servidores, los cuales pueden variar con el paso del tiempo y pueden sufrir sobrecargas o caídas de servicio con mayor frecuencia. Por este motivo, para garantizar que se logra usar al menos un servicio que funcione, en `Vanet-X` se hace uso de una lista, modificable por el usuario, de *APIs* externas, las cuales se intentan usar de forma secuencial hasta hallar una que esté operativa, y finalmente, si ninguna lo estaba, se hace uso de la API principal.

Al igual que existen diferentes *APIs* para conseguir los datos en formato *OSM XML*, también existen diferentes servicios que te permiten lograr imágenes estáticas del mapa. En la correspondiente entrada de la wiki de OpenStreetMap[8] existe una tabla comparativa entre las características de estos servicios.

En este proyecto se ha usado *OSM Static maps API*[9] por ser el que más se adaptaba al resultado que se quería obtener, que era conseguir el mapa mediante una *caja* que delimite sus coordenadas y mediante el zoom que se quiere aplicar. En el momento de decidir qué servicio usar, este era el único que contaba con estas características, ya que los demás en lugar de delimitar el área por una *caja* de coordenadas, lo hacían solo mediante el zoom y estableciendo los pixeles de la imagen resultante.
El motivo de que se necesitase poder delimitar el área por coordenadas era que se quería conseguir mostrar en la imagen exactamente el área descargada en *OSM XML*, o al menos de forma lo más aproximada posible.

Cómo se ha mencionado anteriormente, los datos de OpenStreetMap se descargan en formato *OSM XML*, por lo que para tratarlos se necesita usar un *parser*

[8] `http://wiki.openstreetmap.org/wiki/Static_map_images#Comparison_Matrix`
[9] `http://pafciu17.dev.openstreetmap.org/`

XML.
Se eligió usar *XERCES Java Parser* dado que es muy completo, soporta completamente las APIs XML de Java *SAX*, *DOM* y *JAXP*, y se distribuye bajo la licencia Apache 2.0. La elección del API XML fue *SAX* ya que por eficiencia es la única opción si se desea manipular documentos XML demasiado grandes, como podría ser el caso. Además como no guarda el documento entero en memoria, es una solución muy rápida y eficiente.

Para usar los datos descargados, dado que usan coordenadas geográficas en formato *WGS84*, se ha necesitado realizar una conversión a coordenadas *UTM (Universal Transverse Mercator)*, que en lugar de expresarse en longitud y latitud se expresan en metros. Una vez realizada esta conversión, y después de aplicar un modificador para adaptarlas a las unidades de medida del juego, las coordenadas ya pueden ser utilizadas en nuestro plano de juego.

Como realizar esta conversión entre sistemas de coordenadas es una labor compleja, se hizo uso del código proporcionado abiertamente por *Chuck Taylor* en su sitio web[10]. Como este código esta realizado en lenguaje *JavaScript*, se realizó una conversión para adaptarlo al lenguaje *Java* en el que se ha desarrollado el resto del proyecto.

C.2.2. Implementación

El proceso de la obtención de un mapa se ha implementado de la siguiente forma:

1. Se hace una petición *HTTP GET* con las palabras clave de la dirección que deseas al servicio *OpenStreetMap Nominatim Tool*, el cual devuelve un listado de los lugares coincidentes con las búsqueda realizada. Cada lugar incluye el nombre, coordenadas y otros datos que no nos son relevantes.

2. Una vez seleccionada del listado la localización deseada, se hace una petición de mapa al API elegido con los datos de la «*bounding box*» que delimitará el área. Esta «caja delimitadora» se forma a partir de las coordenadas proporcionadas por el listado y el valor del radio deseado por el usuario, obtenido a través de los menús del juego. El API devolverá un documento con formato *OSM XML* que contendrá los datos requeridos. Si el API al que se ha realizado la petición no funciona correctamente, se repite el proceso con el siguiente API del listado de APIs.

[10] `http://home.hiwaay.net/~taylorc/toolbox/geography/geoutm.html`

3. Una vez obtenidos los datos en formato *OSM XML*, se *parsean* para introducirlos en la estructura de nodos, caminos y multipolígonos de `Vanet-X`. Además, mientras se introducen, se realizan también acciones asociadas como dividir los elementos del terreno en diversas capas para su pintado o calcular los bordes del escenario.

Es importante apuntar que a pesar de que el formato de los datos es el mismo independientemente del API utilizado, ciertas etiquetas opcionales pueden variar, como sucede con la etiqueta «*bounds*», que indica los bordes del área descargada pero que sólo está presente en los datos descargados de la API principal de OpenStreetMap. Por esta razón, cuando se *parsean* los datos, se comprueba la existencia de esta etiqueta y si no está presente se obtiene este dato del fichero *.dat* creado al descargar el mapa.

Los mapas descargados se almacenan y consultan localmente, de forma que una vez descargados ya no es necesario volver a conectarse a la API ni para obtener los datos en formato *OSM XML* ni para obtener la imagen de vista previa de la zona descargada. De esta forma, es posible jugar a `Vanet-X` aun sin tener conexión a internet, solo siendo necesario tener los mapas descargados en la carpeta correspondiente.

Cuando te descargas un nuevo mapa, se consigue mediante las correspondientes APIs los datos *OSM XML* y la imagen de la vista previa, y se guardan en la carpeta «Maps» del directorio de juego en los ficheros *nombre_del_mapa.xml* y *nombre_del_mapa.jpg* respectivamente. El nombre del mapa se obtiene mediante la función *hash* de la dirección concatenada con el radio del mapa, de esta forma se evita poder volver a descargar un mapa cuya área sea la misma que otro ya existente.

Además de estos dos ficheros, se crea *nombre_del_mapa.dat*, que tiene la estructura mostrada en la Tabla C.1 y con sus datos es posible volver a obtener la imagen de vista previa y los datos *OSM XML* en caso de haber sido eliminados. Los ficheros de este tipo son los usados por el menú para crear la tabla de mapas disponibles.

El sistema está diseñado de forma que cuando se requiere usar un mapa seleccionado, bien sea para visualizar su vista previa o para usarlo en una nueva partida, o se actualiza la lista de mapas del menú, si los ficheros *.xml* o *.jpg* no existen, se usan los datos del fichero *.dat* para descargarlos y almacenarlos de nuevo.

Una decisión de diseño tomada fue la inclusión de varios mapas predefinidos para que el usuario no estuviera obligado a descargarse un mapa para empezar a jugar, de forma que se disminuyese el intervalo de tiempo necesario para empezar a jugar desde que se inicia el juego por vez primera.

Alias
Dirección
Radio
Latitud
Longitud
Área (en m^2)
Número de elementos «way»
Número de elementos «node»

Tabla C.1: Estructura de archivo de mapas .dat

Los mapas elegidos lo fueron por tener una gama de diferentes tamaños y haberse realizado en ellos múltiples pruebas que garantizasen una buena jugabilidad.
También se decidió durante la fase de diseño que estos mapas no pudieran ser eliminados, objetivo que no se implementó de forma exacta sino que se optó por crear de nuevo los ficheros cada vez que se inicie el juego. Asi nos aseguramos que aunque hubiesen sido eliminados de forma manual desde el explorador del sistema operativo, los mapas estarían siempre presentes para empezar una nueva partida.

Otra de las decisiones de diseño fue establecer unos límites para la descarga de mapas. No límites de cantidad sino de tamaño de elementos y área del mapa descargado.

Como establecer un límite es una tarea muy difícil ya que depende mucho de la potencia del ordenador, se ideó mostrar en la tabla de mapas descargados el número de elementos y el tamaño de cada mapa y así el usuario, basándose en su experiencia previa con otros mapas, pueda comparar esos datos y predecir el rendimiento que experimentará.

A pesar de esto, finalmente se decidió establecer también un límite basándome en la potencia de un ordenador medio, pero fácilmente modificable mediante un fichero de texto de configuración. Este límite establecido es de una cantidad de elementos inferior o igual a $10,000$ y una extensión menor o igual a $0{,}005^{o}$ de longitud/latitud, lo cual varía dependiendo de la localización pero en la latitud de España es aproximadamente $1km^2$.

C.2.3. Problemas encontrados

Uno de los problemas encontrados fue que al final del desarrollo de este Proyecto Fin de Carrera, OpenStreetMap cambió el tipo de los identificadores de los elementos que forman la estructura *OSM XML* del tipo *Integer* a *Long*, causando que tuviera que modificar el *parser* XML y la estructura interna utilizada en

`Vanet-X` para adaptarla a la nueva realidad.

Otro problema encontrado fue que durante varias horas el servicio mediante el cual se obtienen las imágenes de previsualización de los mapas (*OSM Static maps API*) dejó de estar operativo. Gracias a esto se vio que el sistema desarrollado no estaba hecho a prueba de fallos y se mejoró añadiendo las siguientes características:

- Si la función de búsqueda por nombre (*OpenStreetMap Nominatim Tool*) no funciona, en el *combo box* en el que debería aparecer el listado de posibles coincidencias aparece un texto indicando el error y no deja añadir el mapa.
- Si la función de descarga del *OSM XML* no funciona, cuando pulsas el botón de agregar mapa aparece una ventana emergente informando del error. Además, cuando se actualiza la lista y falta un mapa e intenta descargarlo, se borra el mapa del listado para que deje de estar seleccionable y no se pueda iniciar la partida con él.
- Si lo que no funciona es la función de previsualizar el mapa, se genera una imagen corrupta. Cada vez que se debe mostrar una imagen, el sistema analiza si es correcto y en caso contrario se muestra una imagen de error (como la de la Figura C.4).

Figura C.4: Imagen que se muestra cuando no se ha podido previsualizar el mapa

C.3. Elementos del terreno

En `Vanet-X` los elementos del terreno están estructurados de forma muy similar a la estructura seguida por el proyecto OpenStreetMap para poder realizar una conversión sencilla.

Existen tres tipos de elementos: nodos, caminos y multipolígonos. Los caminos están formados por nodos y los multipolígonos están formados por caminos. Los únicos elementos renderizables son los caminos y los multipolígonos, y se dividen en capas (dieciséis) para que según el elemento al que representen (definido por sus etiquetas) se pinten por encima o por debajo de otros elementos. Así por ejemplo cualquier tipo de carretera o camino se pintará en las capas 5 a 13 (dependiendo de su tipo), mientras que un elemento de tipo «*barrier*» (barrera) se pintará en la capa 15, por lo que siempre se verá «encima» de la carretera.

El criterio seguido para la ordenación de las capas ha sido en parte inspirado por el utilizado en el proyecto *JOSM*[11] pero modificado para conseguir una visualización acorde a las necesidades del juego (ya que el proyecto JOSM está pensado para ser pintado como mapa). El ancho de cada tipo vía también ha sido realizado inspirado por las relaciones utilizadas en los mapas del proyecto *JOSM*[12].

Como los datos descargados de OpenStreetMap representan un área finita, se han de establecer unos límites al escenario de juego. Estos límites son atravesables, pero más allá de ellos el mapa terminará bruscamente ya que fuera del área descargada sólo se continúan los caminos iniciados en el interior, pero no se crean nuevos.
Por ello, se representan visualmente como unas líneas rojas discontinuas y al atravesarlos aparece un aviso en la pantalla en el que se advierte de que te encuentras fuera del área mapeada y este aviso permanece hasta que te reincorporas al interior de la zona delimitada del escenario.

C.3.1. Nodos

Un nodo tiene la siguiente estructura:
Tiene un identificador (el mismo que el del elemento «node» de OSM al que representan), un listado con las etiquetas también obtenidas de OSM y una posición expresada en pixeles que es el resultado de la conversión de las coordenadas WGS84 a UTM y éstas a su vez a las del sistema del juego, donde 10 pixeles equivalen a un metro y se toma como referencia (0,0) la esquina superior izquierda de la caja delimitadora del área descargada.
También tiene dos listados con los nodos con los que está directamente conectado, uno con los nodos que son accesibles con las reglas de tránsito de los vehículos enemigos y otro con las de los vehículos del tráfico. Además también se incluye un listado que contiene la distancia a otros nodos no directamente conectados, que se va rellenando dinámicamente durante la ejecución y sirve para evitar la repetición de ciertos cálculos (ver sección C.5.4 y Figura C.3).
Otro dato que incluye, y es muy importante, es un listado con los identificadores de los caminos en los que está incluido este nodo. Gracias a este dato se pueden calcular los nodos interconectados.
Por último, también se incluye a que «sector» pertenece.

[11] `http://josm.openstreetmap.de`

[12] `http://josm.openstreetmap.de/browser/josm/trunk/src/org/openstreetmap/josm/data/osm/visitor/paint/MapPainter.java?rev=4628`

Los «sectores» es un concepto introducido para asegurar que todos los vehículos que aparezcan en nodos pertenecientes al mismo sector podrán llegar a encontrarse. El concepto es el siguiente: se dividen todos los nodos en diversos sectores, de forma que un nodo estará en el mismo sector que todos los demás nodos a los que sea posible acceder desde él en el grafo de nodos interconectados. Es decir, si dos nodos están en diferente sector significa que para realizar el recorrido entre uno y otro en algún momento será necesario salirse de los caminos y circular campo a través.

Una vez calculados todos los sectores, se elige como sector «bueno» el que tiene un mayor número de nodos, y este sector será el único en el que podrán aparecer los actores. De esta forma se garantiza que un vehículo enemigo que te persiga siempre podrá llegar a alcanzarte.

Un *lugar de interés* es un nodo que tiene dos datos adicionales: una lista con los tres aparcamientos más cercanos y el valor de la distancia existente desde el nodo hasta el camino más cercano transitable por los jugadores. Es un concepto introducido para permitir los objetivos de tipo *tarea* en los modos de juego de resolver tareas y de supervivencia.

Los nodos elegidos para ser *lugar de interés* son los que representan bancos, tiendas, restaurantes, hoteles, y otros puntos de interés.

C.3.2. Caminos

Los caminos pueden ser de dos tipos: áreas, cuyos nodos inicial y final coinciden y forman un camino cerrado, o verdaderos caminos, con un inicio y final diferenciados. La diferencia es únicamente estética ya que se pintan de diferente forma pero mantienen en común el resto de características.

La estructura es la siguiente:
Un camino cuenta con el identificador y las etiquetas obtenidas del elemento «way» de OSM al que hace referencia. También cuenta con un listado de los nodos que componen el camino y una lista con los segmentos rectos entre los nodos, que se utilizan para pintar, calcular si los vehículos están sobre el camino y para crear los puntos intermedios donde se crearán las plazas de aparcamiento y los puestos de comida.
Estos mismos segmentos rectos que forman el camino también están representados, aunque con distinto formato, en una lista que usa el *steering behavior* de *path following* para mantenerse sobre la calzada.

Otros datos que también están presentes son el ancho, el color o la capa a la que pertenece, así como otros muchos parámetros con propiedades para la circulación

(por ejemplo indicando si el camino es transitable por los vehículos del tráfico o si se trata de una calle de sentido único).

Un valor muy importante son las propiedades del terreno, es decir, el «efecto» que produce sobre los vehículos: infranqueable, ralentiza la velocidad del vehículo, causa daño a los vehículos, etc.

Al igual que los nodos, también debido a la inclusión de los modos de juegos donde aparecen tareas como objetivos, se ha incluido una lista con los tres aparcamientos más cercanos, en este caso al punto medio del primer segmento del camino, ya que es este el que se utiliza como objetivo de las misiones de tareas.

C.3.3. Multipolígonos

Los multipolígonos son las estructuras con las que se representan las áreas complejas, bien por el gran número de caminos que la componen o por necesitar definir áreas en el interior de otras áreas (por ejemplo un patio de luces en el interior de un edificio).
Es la única de las relaciones *OSM* representadas en el juego.

Para crear esta estructura en el juego, debido a su complejidad, se copió la implementación utilizada en el proyecto *JOSM*[13], aunque se simplificó ligeramente.

Además de esta implementación, también se desarrolló otra alternativa más simple (sin permitir roles en los componentes) que pudiera ser utilizada en los casos en los que la otra implementación no da buenos resultados (por ejemplo en los ríos). Las dos implementaciones comparten la estructura utilizada y el valor de una variable es la que determina que implementación es la usada.

La estructura general de un multipolígono, independientemente de cuál de las dos implementaciones se use, es la siguiente:
Al igual que los otros elementos, cuenta con el identificador y las etiquetas obtenidas del elemento de OSM al que hace referencia, y como los caminos, también cuenta con la capa en la que se debe pintar y con las propiedades del terreno al que representa.

Si la implementación utilizada es la de *JOSM*, cuenta con una lista de elementos «*PolyData*», los cuales tienen una estructura que permite que cada camino formante del multipolígono tenga un rol definido, que puede ser anillo interior o exterior del área. Sin embargo si la implementación es la simple, se cuenta con una variable que contiene el polígono representando el área, la cual no puede tener huecos ni contar con varios anillos interiores o exteriores.

[13] `http://josm.openstreetmap.de/browser/josm/trunk/src/org/openstreetmap/josm/data/osm/visitor/paint/relations/Multipolygon.java?rev=4628`

C.4. Física y colisiones

Como se ha explicado en el capítulo 2.6, el sistema encargado de las físicas del juego puede dividirse en cuatro algoritmos: detección de colisiones (con el terreno o con los actores) y aplicación de fuerzas resultantes de la colisión (también de forma diferenciada para colisiones con el terreno y colisiones con los actores).

A continuación se explican estos algoritmos en detalle.

C.4.1. Detección de colisión con elementos del terreno

Es necesario conocer sobre qué elemento del terreno se encuentra un vehículo, ya que dependiendo de sus propiedades el vehículo deberá reaccionar de una manera u otra. Para averiguarlo se utiliza el siguiente algoritmo:

Para cada elemento del terreno, se llama a una función (diferente si se trata de un camino o de un multipolígono) en la que, conociendo el centro de la posición del vehículo, el radio de la menor circunferencia capaz de contenerlo y los cuatro vértices del menor rectángulo rotado que lo contiene, se realiza primero un primer cálculo en el que se comprueba si el menor rectángulo capaz de contener al elemento del terreno («*bounding box*») colisiona con la menor circunferencia que contiene al vehículo.
Este cálculo es capaz de determinar con total seguridad si no existe colisión, ahorrándonos realizar cálculos más complejos. Sin embargo, si el resultado del cálculo es que sí que colisiona, se debe realizar un segundo cálculo, éste más costoso, en el que se comprueba si alguno de los cuatro vértices del rectángulo rotado que representa al vehículo está contenido dentro del elemento del terreno.
Con este segundo cálculo ya se puede determinar definitivamente si existe o no la colisión.

Este proceso se realiza con todos los elementos del terreno, no sólo con la primera colisión coincidente ya que puede que exista otra colisión con un terreno con propiedades más restrictivas (como se explica más adelante en esta misma sección). Sin embargo, en cuanto se han registrado que el vehículo va a sufrir las más restrictivas de las propiedades («velocidad: sin problemas», «infranqueable: sí» y «obras: sí») ya se puede dejar de continuar con la búsqueda ya que nunca se encontrará una colisión cuyas propiedades sobrescriban estas.

Otra optimización realizada es omitir todos los elementos de la capa que contiene los edificios, ya que se decidió que se pintaran por debajo de las capas que contienen los caminos transitables y que no fueran colisionables. Esta decisión se tomó ya que la idea inicial de que fueran colisionables presentaba un gran problema, y era que en muchos casos estaban situados invadiendo la calzada por lo que existían caminos que se estrechaban o se interrumpían por los edificios complicando exageradamente el algoritmo de path-finding necesario.

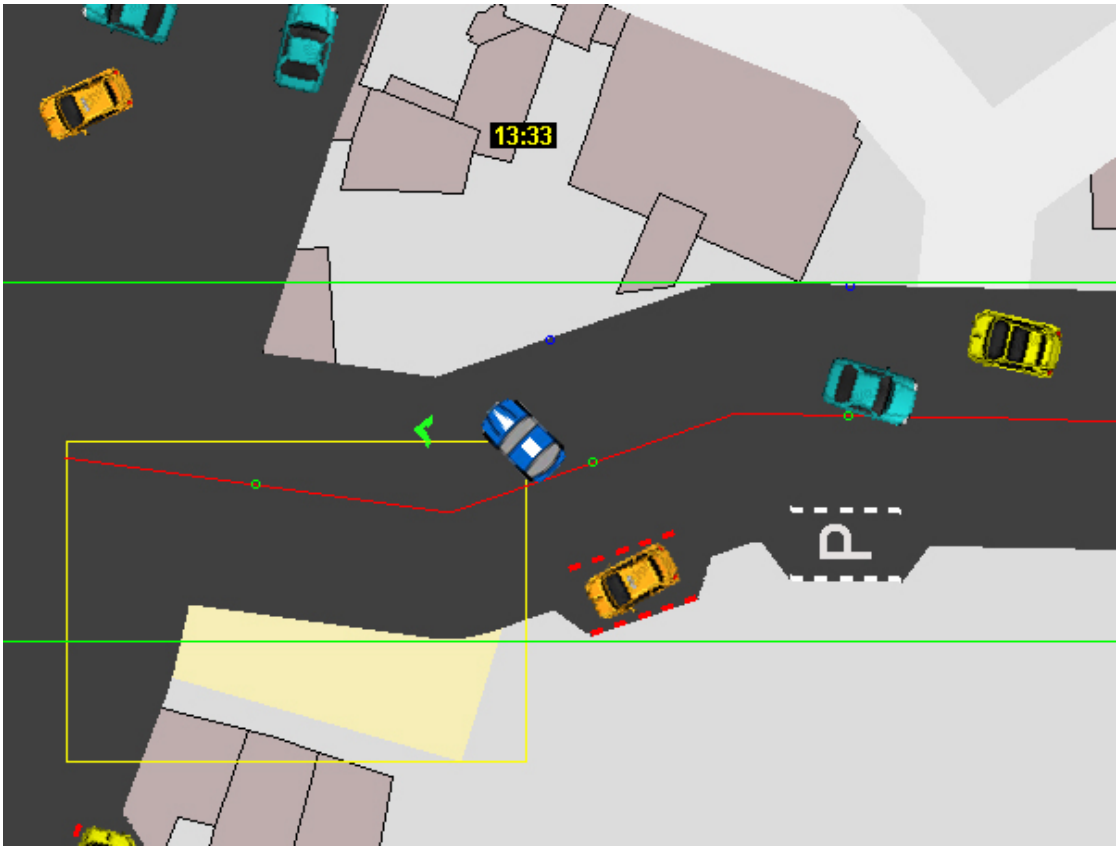


Figura C.5: Detección de colisiones con el terreno. En esta captura los bordes de los «*bounding box*» están pintados de diferente color según si realmente existe colisión (verde), existe colisión con el *boundig box* (amarillo) o no existe colisión (no hay línea). Además en el caso de existir colisión se pinta de rojo los segmentos del camino.

C.4.2. Detección de colisión con otros actores

La detección de colisiones con otros actores sigue un esquema similar al anterior en cuanto a que primero se realiza una comprobación poco costosa que es capaz de descartar la colisión, y en caso de que no la descarte, se realiza una segunda comprobación en detalle que sí que es capaz de determinar con exactitud la existencia de colisión.

La primera comprobación consiste en comprobar si las menores circunferencias capaces de comprender a los vehículos se intersectan.

La segunda comprobación consiste en aplicar el *Separating Axis theorem*, el cual afirma que si dos polígonos conversos no colisionan, existe un eje perpendicular a una arista de uno de los polígonos en el cual la proyección de los objetos no se superpone.[14] Ver Figura C.6.

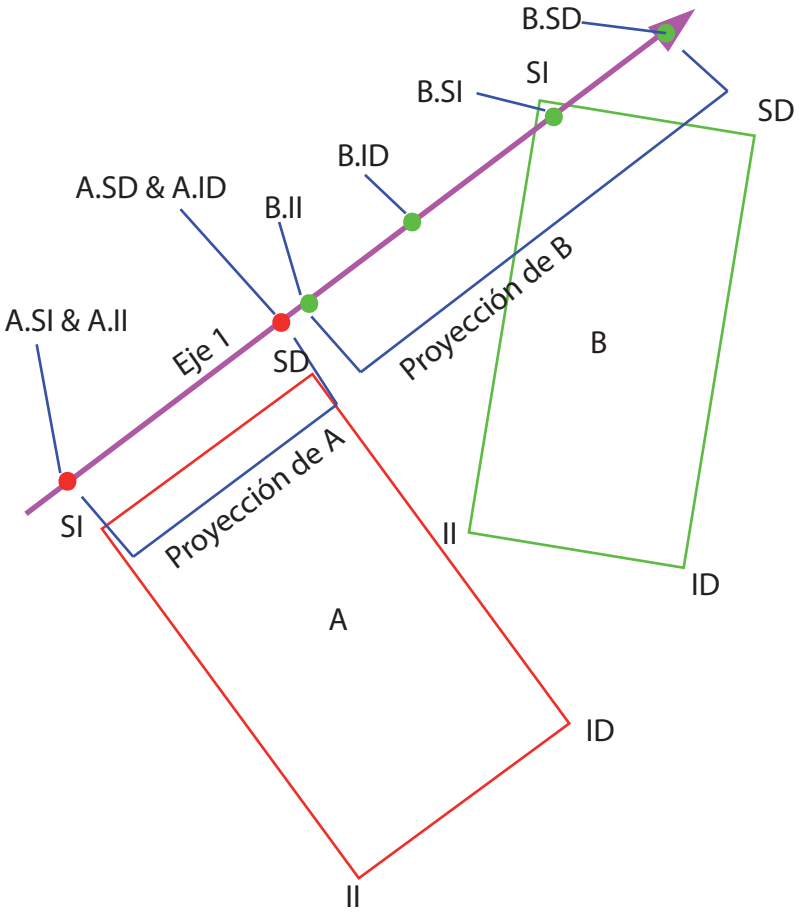


Figura C.6: *Separating Axis theorem*

[14] http://www.codezealot.org/archives/55#sat-algo

El algoritmo utilizado se ha basado en el propuesto en el articulo «A Verlet based approach for 2D game physics» de Benedikt Bitterli [3], simplificándolo ya que en nuestro caso los polígonos son siempre rectángulos.

Por norma general las circunferencias y rectángulos rotados utilizados para la detección de colisiones están situados de forma que abarquen el actor deseado por completo. Sin embargo, en el caso de las falsas plazas de aparcamiento («falsas» ya que siempre están ocupadas), es deseable que la colisión se realice con el vehículo aparcado (que en realidad no existe como elemento independiente sino que forma parte del aparcamiento). Por esta razón se modifican el rectángulo y la circunferencia para que coincidan con la imagen del vehículo aparcado, como se puede ver en la Figura C.7.

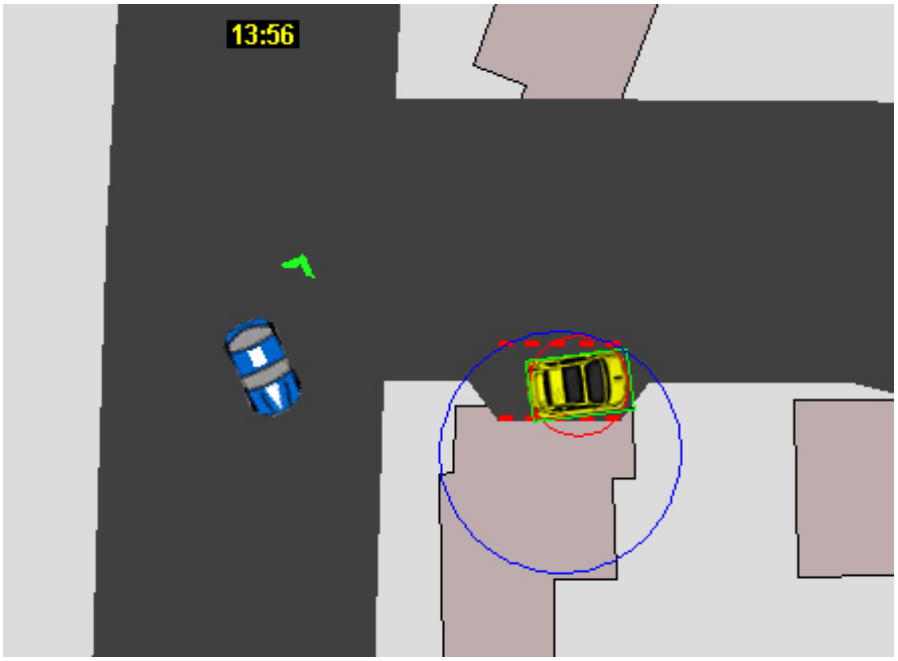


Figura C.7: Captura en la que se observa el rectángulo rotado (verde) y la circunferencia (rojo) que contienen la zona colisionable del parking. También se aprecia la circunferencia que marca el obstáculo que debe evitar el *steering behavior* de *obstacle avoidance* (azul)

C.4.3. Cálculo de la fuerza resultado de una colisión con otros actores

El algoritmo que calcula la fuerza que se debe aplicar en una colisión con un actor es el siguiente:

1. Calculas la fuerza resultante de las fuerzas de los dos vehículos implicados.
2. Si el otro vehículo está en estado «inmovil» te restas la fuerza calculada en el paso 1. En caso contrario le sumas esa fuerza al otro vehículo.
3. Estableces la velocidad del vehículo a cero.

El cálculo de la fuerza resultante descrito en el paso 1 se calcula de la siguiente manera:

1. Obtenemos los vectores velocidad de los dos implicados. Ver Figura C.8.a
2. Realizamos una rotación de forma que la línea imaginaria entre los dos implicados quede en el eje Y. Ver Figura C.8.b
3. El vector resultado (provisional) es la componente Y del vector velocidad de mi vehículo, salvo que se trate de un choque «por alcance» en cuyo caso se le debe restar la componente Y del vector del otro vehículo. Ver Figura C.8.c
4. Para asegurar que no se genere una fuerza de atracción cuando el vector está en sentido opuesto al otro implicado (sentido negativo de la Y), se debe asegurar que el valor resultado nunca será mayor a -1.
   (Es -1 y no 0 porque así nos aseguramos de que siempre exista fuerza de repulsión entre los coches).
5. Se vuelve a rotar el vector resultado invirtiendo la rotación realizada en el paso 2. Ver Figura C.8.d

C.4.4. Aplicación del resultado de la colisión con el terreno

La resolución de la colisión con un elemento del terreno es diferente, y mucho más simple, que la realizada en las colisiones con otros actores.

El algoritmo que analiza si existe colisión con algún elemento del terreno (es decir, circulas sobre él), en el caso de que se produzca colisión, devuelve una estructura que contiene las propiedades del terreno: si causa daño, velocidad permitida, si es infranqueable y si se trata de un tramo en obras.

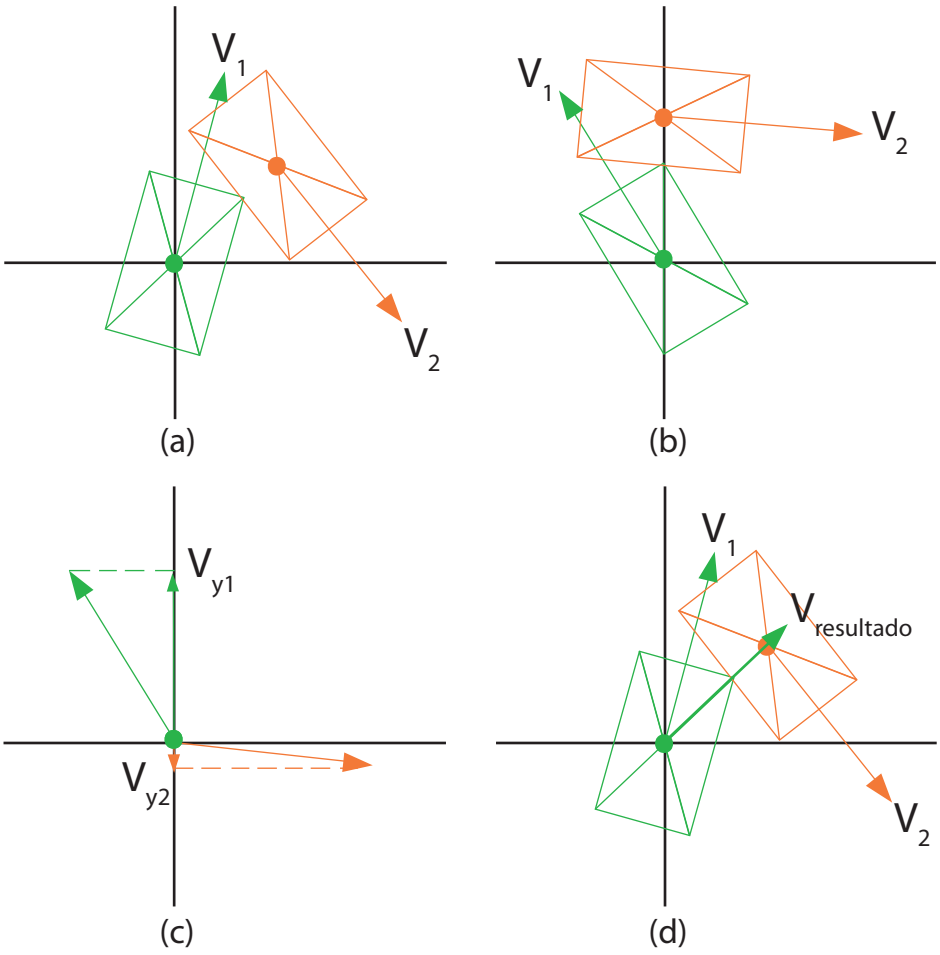


Figura C.8: Cálculo del vector fuerza resultante de una colisión

Como el vehículo puede colisionar con varios elementos del terreno simultáneamente, las propiedades que devuelve el algoritmo son los máximos de cada valor perjudicial (daño, infranqueable y obras) y el menor valor velocidad de todos los elementos sobre los que está circulando. Por ejemplo, si el vehículo circula sobre una carretera residencial en obras que atraviesa un curso de agua, el valor de la velocidad se obtendrá de la carretera ya que tendrá un valor «camino sin problemas» (el mejor valor posible).

Con este valor del terreno que devuelve el algoritmo, se realizan las siguientes comprobaciones:

- Si el vehículo colisiona con un terreno para el que no es apto (salvo que se trate de una plaza de aparcamiento), se marca al vehículo como inmóvil, se le establece velocidad nula, y se mueve a la posición del ciclo anterior (la justo anterior a la colisión). Al marcar el vehículo como inmóvil no se volverá a efectuar las comprobaciones de colisión con el terreno hasta que no vuelva a moverse (concretamente se mueva más de 1,25 metros).
- Si el valor «velocidad» de terreno es «ralentizar», se le marca al vehículo una variable para que en el siguiente ciclo su movimiento se vea reducido a la mitad.

En la Tabla C.2 se muestra una relación de los posibles valores del terreno.

«Velocidad»	detener, ralentizar, sin problemas
«Obras»	sí, no
«Infranqueable»	sí, no
«Daño»	extremo, ligero, no

Tabla C.2: Posibles valores del terreno

C.5. Inteligencia Artificial

En esta sección se ofrece una visión detallada de la inteligencia artificial desarrollada para el juego.

C.5.1. Steering behaviors

Como se ha explicado en el capítulo 2.7.2, se han utilizado los llamados *Steering behaviors*, comportamientos básicos de movimiento de los actores. Todos ellos se han obtenido de [17].

Los comportamientos implementados son los siguientes:

Seek (buscar) es el comportamiento básico en función del cual se pueden crear los demás. Consiste en dirigir el vehículo hacía el objetivo. Esto lo logra ajustando la dirección de forma que la velocidad está radialmente alineada con el objetivo. La velocidad deseada es un vector en dirección del vehículo al objetivo, cuya longitud puede ser la velocidad actual del vehículo o la velocidad máxima (en `Vanet-X` se ha usado la velocidad máxima). Nótese que si se aplicase esta velocidad deseada, el vehículo empezaría a orbitar en torno al objetivo, ya que es una fuerza de atracción. El vector dirección deseado (marcado como *dirección seek*) se obtiene como la diferencia de la velocidad deseada y la velocidad actual del vehículo. Ver Figura C.9.

Flee (huir) es el comportamiento inverso a *seek*. En lugar de dirigir el vehículo hacia el objetivo lo dirige en el sentido contrario, de forma que se aleje lo máximo posible del objteivo. Ver Figura C.9.

Pursuit (persecución) es similar a *seek* pero el objetivo es móvil en lugar de fijo. Consiste en aplicar el comportamiento *seek* con la posición futura predicha del objetivo. Esta predicción se realiza suponiendo que el objetivo no varíe su trayectoria, y se calcula su posición futura calculando la distancia recorrida en un tiempo T y añadiéndosela a la actual. Es muy importante establecer un T apropiado, $T = D \times c$ es considerado un valor apropiado, siendo D la distancia entre los dos vehículos y c un parámetro de giro. Ver Figura C.10.

Evasion (evasión) es el comportamiento inverso a pursuit, es decir, usando *flee* en lugar de *seek*. Ver Figura C.10.

Arrival (llegada) tiene el mismo comportamiento que *seek* mientras estás lejos del objetivo. La diferencia radica en que disminuye la velocidad conforme te acercas al objetivo hasta llegar a detenerse sobre él.

Obstacle avoidance (evitación de obstáculos) dota al vehículo de la habilidad para esquivar obstáculos de su entorno. Consiste en mantener una zona delante del vehículo en la que compruebas si hay algún obstáculo y en caso de que lo haya se aplica un cambio de trayectoria en sentido contrario. De esta forma, a diferencia del resultado si se hubiera aplicado *flee*, solo se cambia la trayectoria en caso de que alguno de los obstáculos efectivamente esté en la trayectoria del vehículo. Para representar los obstáculos se usan esferas que los contengan, y siempre tiene prioridad la más cercana al jugador (si hay más de una que colisiona con la trayectoria solo se tiene en cuenta la primera). Ver Figura C.11.

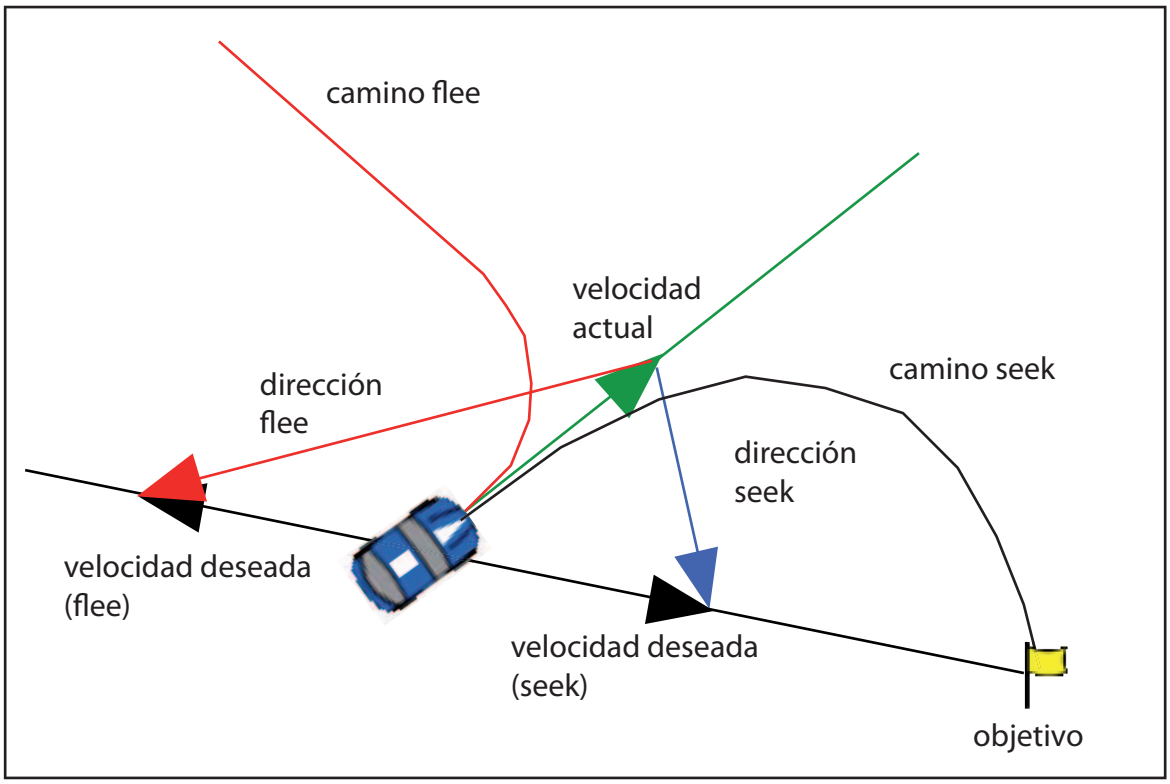


Figura C.9: Comportamientos *seek* y *flee*

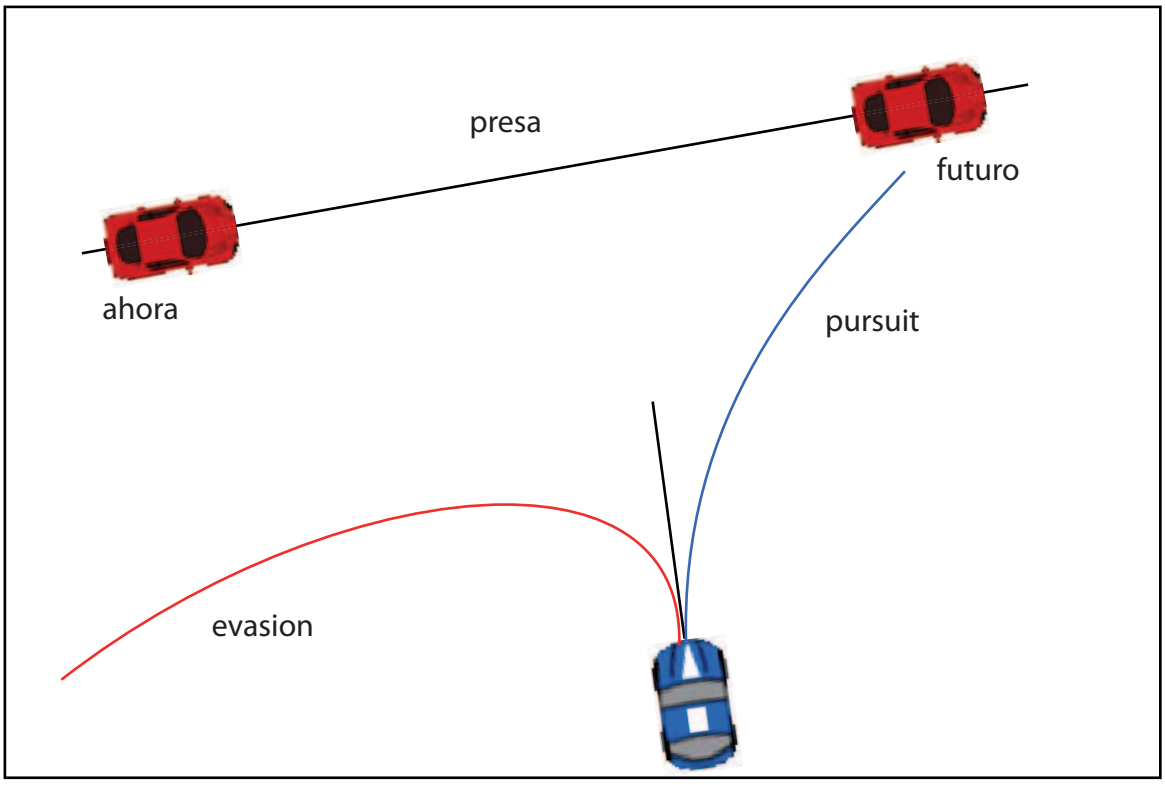


Figura C.10: Comportamientos *pursuit* y *evasion*

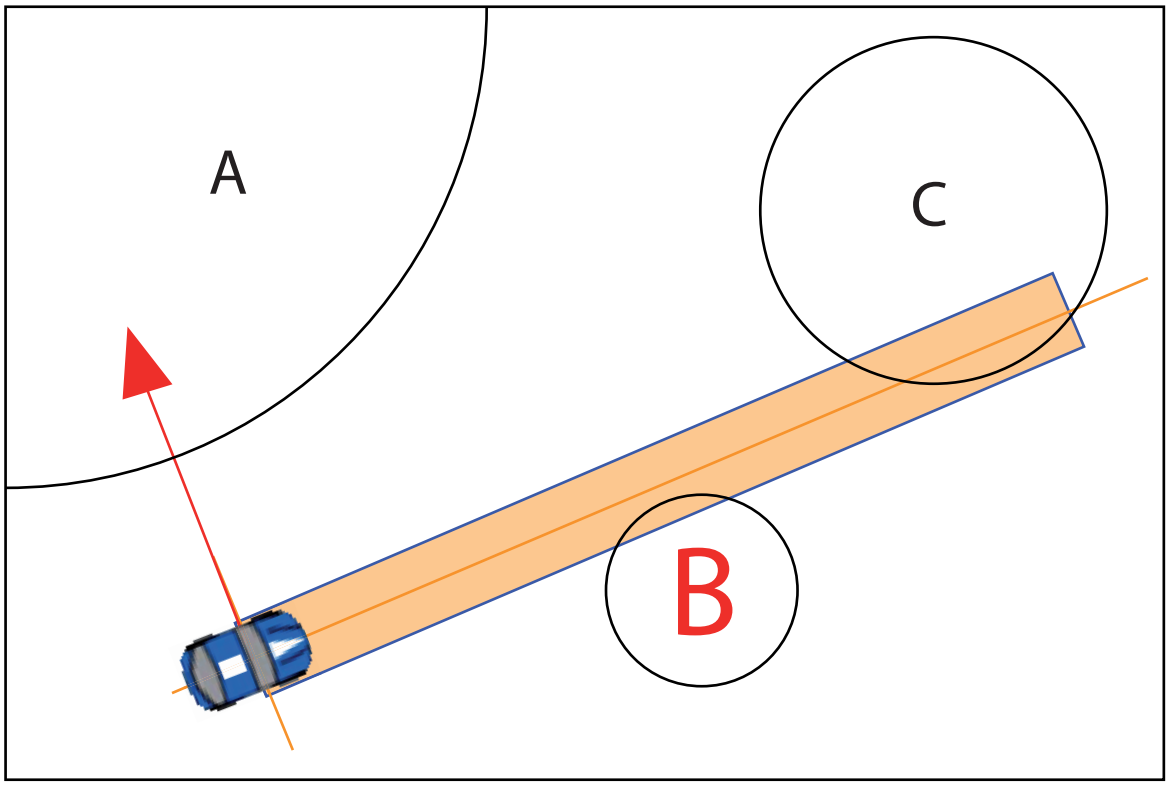


Figura C.11: Comportamiento *obstacle avoidance*: el obstáculo B es el primero en la trayectoria por lo que se aplica una fuerza para evitarlo.

Wander (deambular) genera una trayectoria aleatoria. En lugar de generar una fuerza aleatoria en cada ciclo, lo cual generaría una trayectoria demasiado nerviosa, se mantiene «memoria» de cuál ha sido la fuerza anterior para generar unas transiciones suaves. Esto se realiza de la siguiente manera: se genera una esfera ligeramente avanzada respecto al vehículo, y la fuerza que se genere será el vector desde el vehículo hasta un punto del perímetro de la esfera. Para producir la fuerza del siguiente ciclo se añade un desplazamiento aleatorio al valor anterior, y la suma se constriñe de nuevo al perímetro de la esfera. En la Figura C.12 se puede observar que la fuerza máxima de giro viene dada por la esfera grande y la cantidad del desplazamiento aleatorio (esfera pequeña) determina la velocidad con la que cambia.

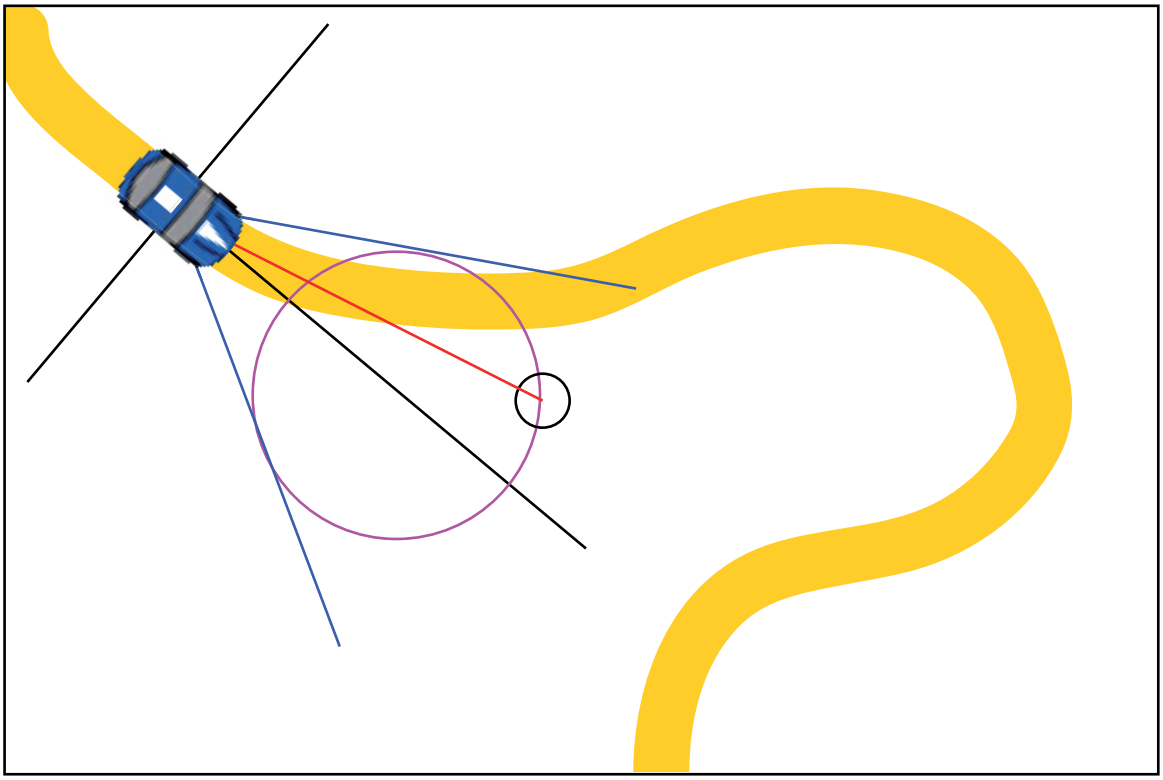


Figura C.12: Comportamiento *wander*

Path following (seguimiento de ruta) permite que un vehículo circule por un camino sin salirse. No hay que confundirlo con obligar a un vehículo a ir exactamente por el camino, como si fuera sobre un raíl, sino que este comportamiento produce un resultado más natural, ya que el vehículo puede seguir la trayectoria que quiera siempre que no se salga del camino.
En la implementación utilizada se representa el camino como una polilínea

con un radio, como si de un «tubo» se tratara. El objetivo de este comportamiento es mover el vehículo por dentro de este camino sin salirse del radio del «tubo». Si el vehículo no está inicialmente dentro, la primera acción es acercarse al camino, y después seguirlo.
El procedimiento para calcular la fuerza de dirección a aplicar es el siguiente: se calcula una predicción de la posición futura del vehículo, como la realizada en el comportamiento *obstacle avoidance*, y se proyecta sobre el punto más cercano de la polilínea. Si la distancia de la proyección a la polilínea es menor que el radio, significa que no te vas a salir del camino y no se necesita corregir la trayectoria. En caso contrario, se debe usar el comportamiento *seek* teniendo como objetivo el punto de la polilínea sobre el que se ha proyectado la posición futura. Ver Figura C.13

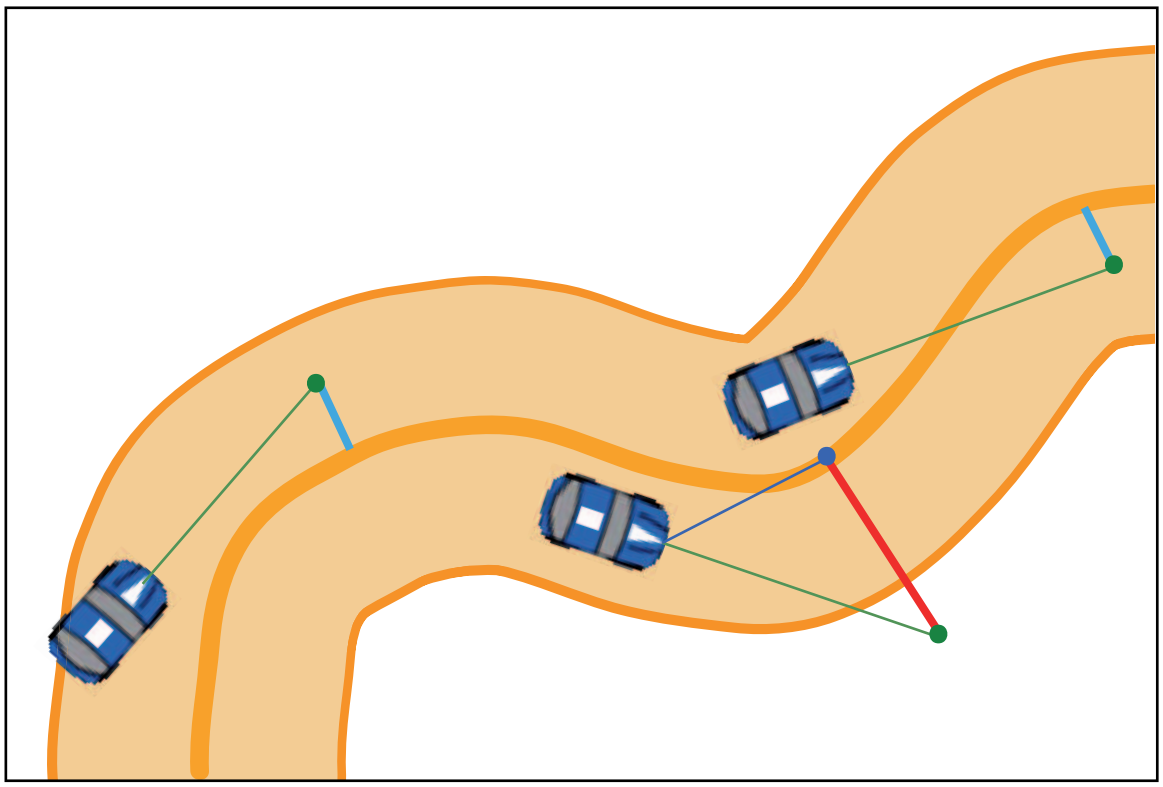


Figura C.13: Comportamiento *path following*

Unaligned collision avoidance (esquiva de obstáculos no alineados) es un comportamiento que trata de evitar que colisionen vehículos moviéndose en diferentes direcciones. El funcionamiento consiste en que cada vehículo calcula cual va a ser su mayor aproximación a cada uno de los demás vehículos. Si la mayor aproximación a un vehículo es en el futuro, y la distancia que

tendrán es menor a una establecida (círculos de la Figura C.14) implica que hay un riesgo de colisión. Si se ha calculado que habrá alguna posible colisión, el vehículo tratará de evitar la más cercana de todas aplicando una fuerza que le aleje de la colisión.

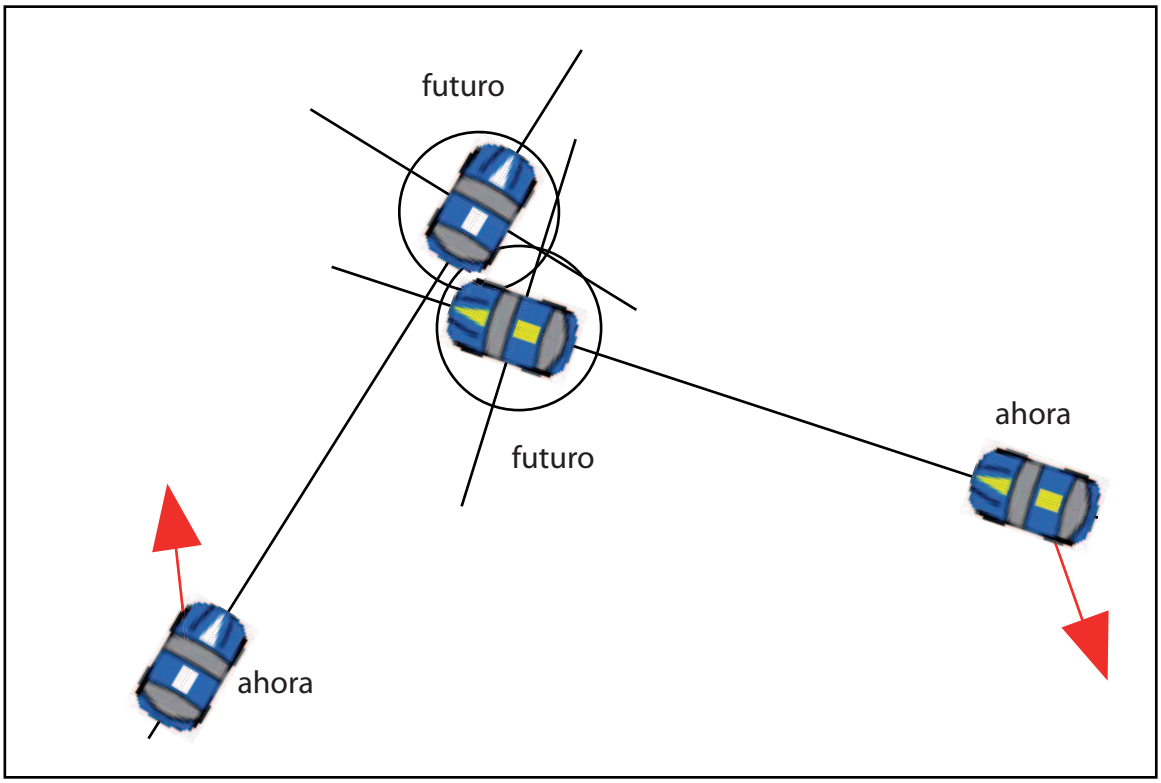


Figura C.14: Comportamiento *unaligned collision avoidance*

Estos comportamientos se pueden combinar para formar comportamientos más complejos. Esta combinación puede realizarse de múltiples formas, la escogida es la siguiente: se ordenan los *steerings* por prioridad, y se analizan en orden de forma que se aplicará el primero que de como resultado una variación de la trayectoria. De esta forma si en un comportamiento hemos dado prioridad a no salirse de la pista (*path following*) antes que a esquivar los otros vehículos (*unaligned collision avoidance*), es posible que nos choquemos contra uno de ellos a costa de asegurarnos mantenernos dentro de la calzada.
Otra forma posible sería sumando los resultados obtenidos con los distintos *steerings* (pudiendo tener unos más peso en la suma que otros).

Todos los comportamientos complejos (explicados en la siguiente sección) tienen la misma composición de prioridades de *steerings*, variando únicamente en los

vehículos que se contabilizan en la *unaligned collision avoidance* (Sólo tráfico y enemigos o también jugadores) y en el comportamiento menos prioritario (*arrival*, *pursuit*, *evasion*...) y su forma de ejecución (*steering*, *steering suave*...). Éste diseño se puede ver en la Figura C.15.

Las formas de ejecución anteriormente citadas y su funcionamiento son los siguientes:

- *steering*: modo normal, maneja los controles del vehículo según el vector recibido.
- *steering suave*: gira mas o menos suavemente dependiendo del cambio de ángulo necesario para encararnos al objetivo.
- *steering always up*: sin frenar, siempre al máximo de velocidad.
- *steering reverse gear*: en marcha atrás.

C.5.2. Comportamientos complejos

Todos los actores dotados de inteligencia artificial tienen un comportamiento en común, y es el de pasar de estado normal a marcha atrás y viceversa. Este comportamiento se ha realizado dado que la inteligencia artificial no es perfecta y hay ocasiones en las que los vehículos acaban saliéndose de los límites de la carretera, siendo necesario un procedimiento para que den marcha atrás y se reincorporen a la circulación. El diagrama de este comportamiento se puede ver en la Figura C.16. El estado «normal» es diferente para cada tipo de vehículo y se detalla a continuación diferenciado según el tipo de vehículo.

Enemigos: su comportamiento depende de si el vehículo tiene un rol de perseguidor o de perseguido. Si es el perseguidor, sigue el camino dictado por el *path-finding* usando un comportamiento de llegada con los nodos del camino para que frene al llegar a cada nodo y así tome los cruces entre calles a una menor velocidad (*steering_PF_ARRIVAL*, ver Figura C.15). En el momento en que se encuentra en la misma calle del objetivo, ya no hace falta usar el camino proporcionado y se aplica un comportamiento de persecución (*steering_PF_PURSUIT*) sobre el vehículo objetivo (siempre se eligirá como objetivo al jugador más cercano). El comportamiento es de persecución y no de búsqueda para que se anticipe a los movimientos de la presa.
En caso de que se trate del perseguido, el comportamiento es el siguiente: si el vehículo que le persigue está cerca (a menos de 100 metros), aplica un comportamiento de huida (*steering_PF_FLEE*) sobre el perseguidor, de

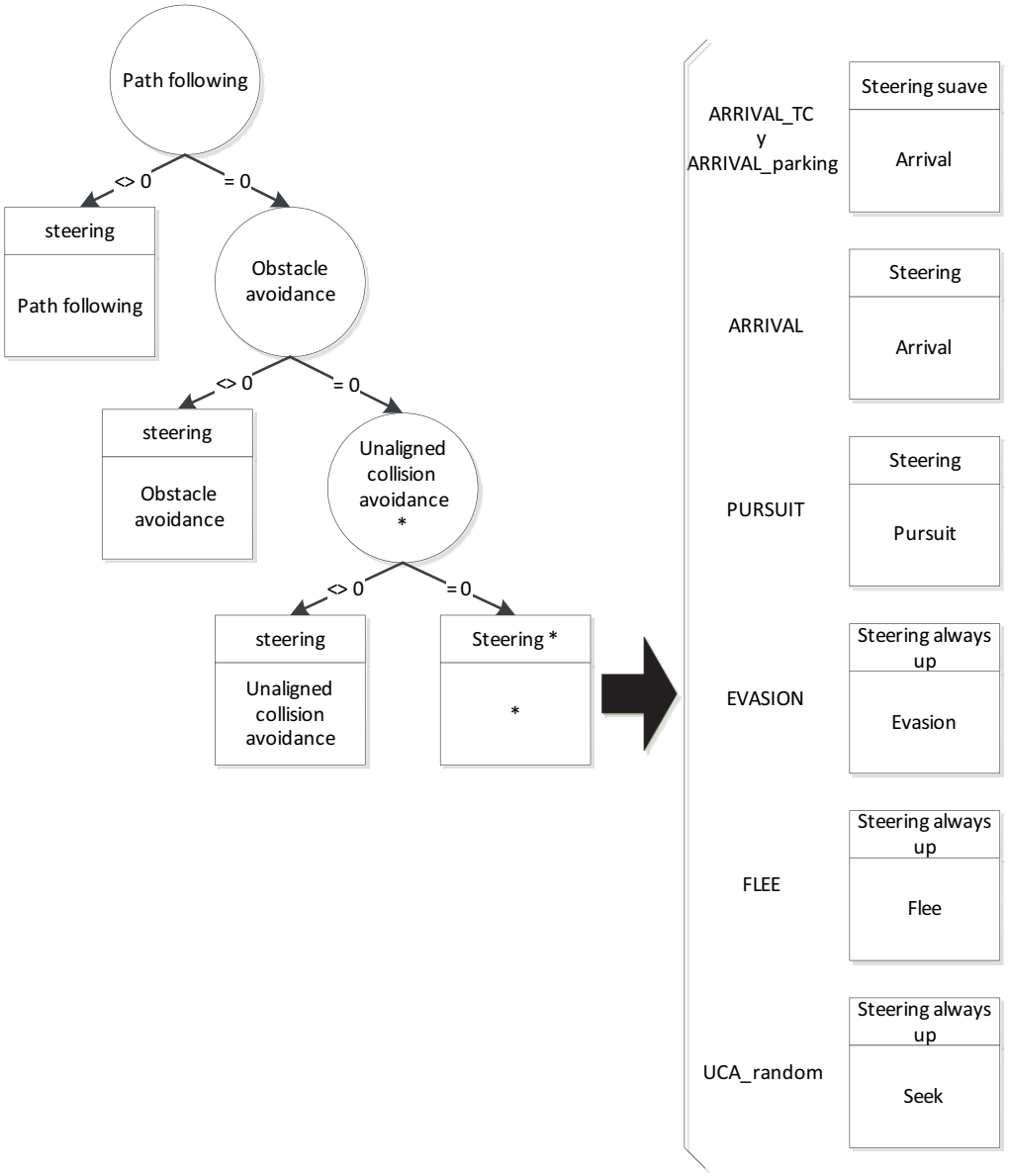


Figura C.15: Prioridades en la composición de comportamientos a base de *steerings*

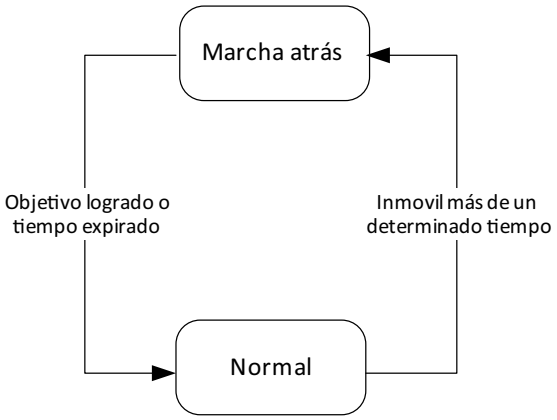


Figura C.16: Diagrama de estados IA: visión general

forma que tratará de alejarse lo máximo posible de su posición actual. Sin embargo, si el perseguidor está lejos, se usará un comportamiento de evasión (*steering_PF_EVASION*,) ya que de usar el anterior comportamiento, el vehículo siempre huiría hacia el extremo del escenario más alejado del perseguidor, hasta salirse del mapa y quedándose allí atascado. Ver Figura C.17.

Ambulancias: su comportamiento es muy simple: elige un nodo al azar y se dirige a él. Cuando llega hasta él, elige otro y asi sucesivamente. Existe un tiempo límite para alcanzar el nodo, si tarda más elige un nuevo nodo objetivo.

Tráfico: tiene el comportamiento más complejo de los tres. Como se muestra en la Figura C.18, dentro de su estado normal tiene cuatro posibles subestados:

- *Circular*: consiste en elegir un nodo objetivo y alcanzarlo. Este estado se repite hasta que al vehículo le llega una señal que indique que hay menos coches buscando parking de los establecidos. Ver Figura C.19.
- *Buscar aparcamiento*: es el mismo comportamiento que el anterior pero ahora está atento a aparcamientos vacíos. De esta forma si visualiza un aparcamiento libre pasará al siguiente estado (*Aparcar*), mientras que si en un tiempo establecido no ha logrado visualizar ninguno vuelve al estado anterior (*Circular*). Además, si recibe un evento VESPA de parking libre se dirige a la posición del evento (aunque por el camino puede encontrar otro aparcamiento libre más cercano y aparcar en él).
- *Aparcar*: consiste en realizar la maniobra de aparcamiento sobre el aparcamiento libre objetivo. Se realiza mediante un comportamiento de llegada para que el vehículo frene al aparcar. Si se logra aparcar se avanza

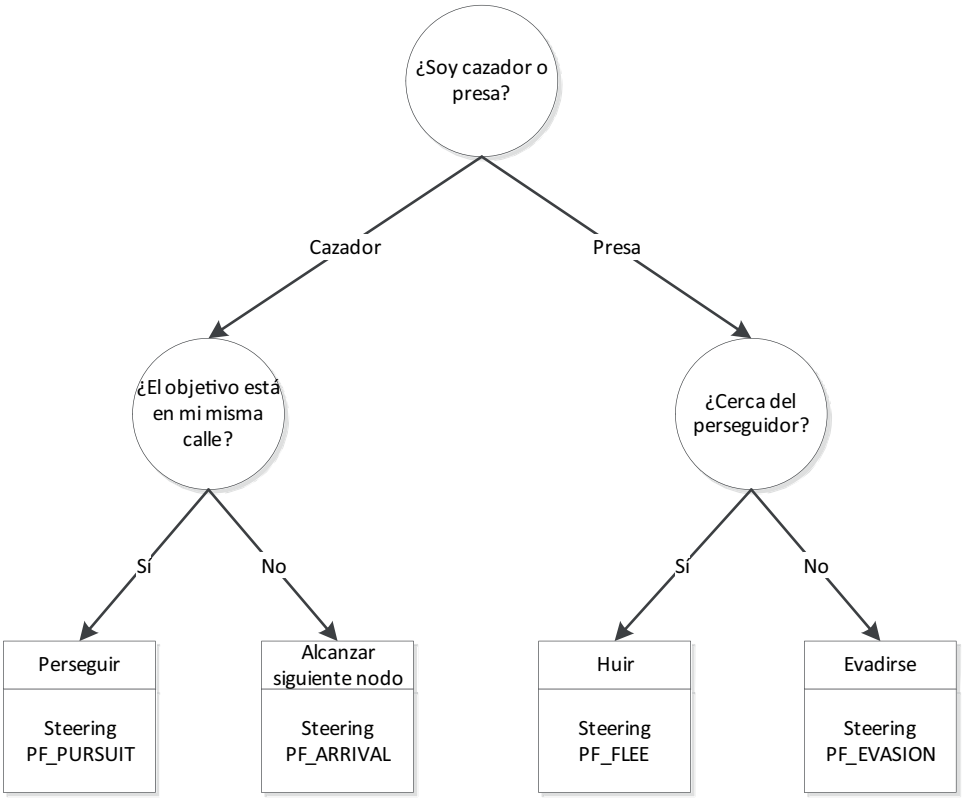


Figura C.17: Detalle del estado *Normal* de la inteligencia de los vehículos enemigos

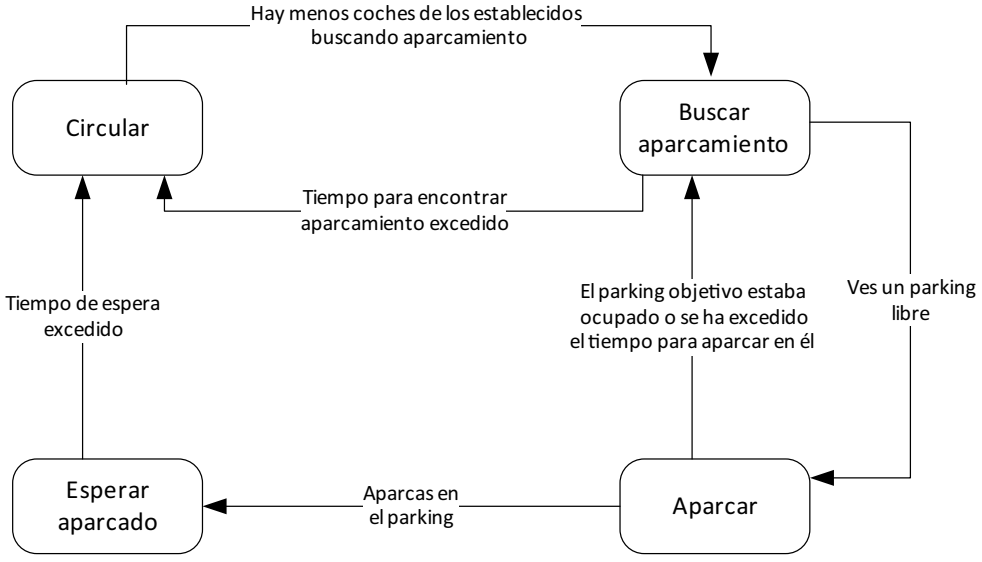


Figura C.18: Detalle del estado *Normal* de la inteligencia de los vehículos del tráfico

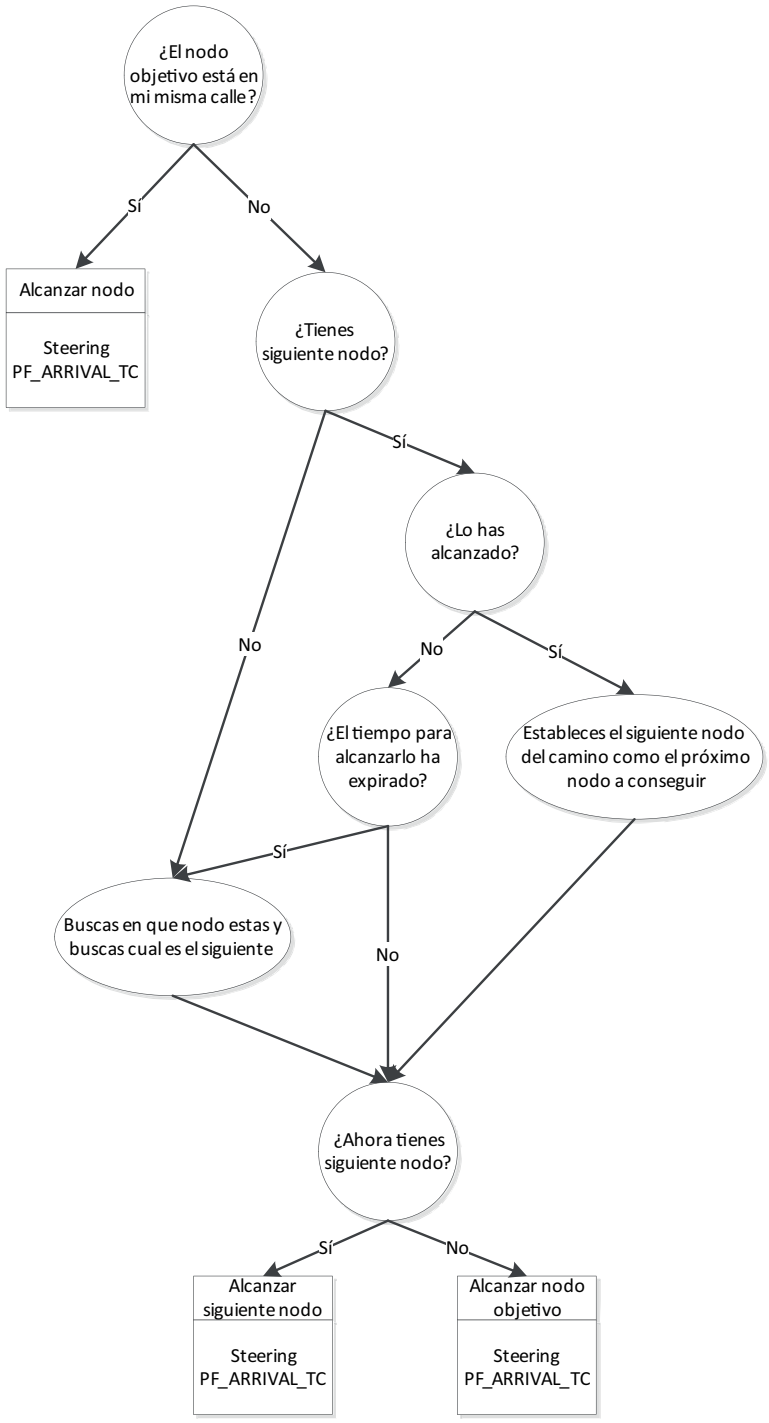


Figura C.19: Detalle del estado *Circular* y *Buscar aparcamiento* de la inteligencia de los vehículos del tráfico

al siguiente estado (*Esperar aparcado*) mientras que si no se ha logrado se retrocede al estado anterior.

- *Esperar aparcado*: consiste en esperar quieto dentro de la plaza de aparcamiento durante un tiempo determinado. Una vez completado ese tiempo se vuelve al estado inicial (*Circular*).

C.5.3. Soluciones a las carencias de la IA

A pesar de los esfuerzos realizados por conseguir una inteligencia artificial solvente, hay ocasiones en el juego en el que es necesario aplicar ciertas «trampas» para corregir los errores de la inteligencia. Estas situaciones solucionadas son dos y ambas están relacionadas con los vehículos del tráfico: solucionar que a veces se quedan atascados al salirse de la carretera y no son capaces de reincorporarse a la calzada, y evitar un molesto efecto de «nerviosismo» consistente en que giran de una forma poco gradual y están continuamente corrigiendo su trayectoria.

Desatascador de vehículos

Como norma general, cuando los vehículos controlados por la IA se salen de la calzada, al cabo de unos segundos dan marcha atrás y se reincorporan satisfactoriamente. Sin embargo, hay ocasiones que por la topología del terreno la inteligencia desarrollada no es capaz de reincorporarse y se queda el vehículo atascado. Para estas ocasiones se ha realizado un método que determina si en efecto el vehículo está bloqueado y le aplica una rotación de 90º para ver si así en esa nueva posición es capaz de reincorporarse.

El funcionamiento detallado es el siguiente:
Para cada vehículo se tiene un registro con sus últimas cinco posiciones (las cuales se recogen cada dos segundos) y se comprueba si ha existido movimiento significativo en total o en alguno de esos cuatro intervalos (para evitar que si te mueves pero acabas en la posición inicial no cuente como movimiento). Si en esos diez segundos el vehículo ha recorrido menos de 6,25 metros, se considera atascado, y en el siguiente ciclo del juego se le realiza una rotación de 90º en el sentido de las agujas del reloj. De esta forma el vehículo seguirá rotando cada diez segundos mientras siga atascado, hasta que la IA consiga reincorporarlo a la calzada.

Corrector de trayectoria *nerviosa*

Los vehículos del tráfico tenían un movimiento espasmódico en lo concerniente al ángulo, variaban de ángulo en cada ciclo y resultaba un efecto muy feo a la vista.

El motivo de este movimiento espasmódico es que la velocidad de giro de los vehículos del tráfico es mayor que la de los demás vehículos, en parte por circular a menor velocidad y también porque se comprobó que aumentando esta velocidad se obtenían mejores resultados en la esquiva de los obstáculos y en el seguimiento de la calzada.

Como, por estos motivos, cambiar esta velocidad de giro no era una opción, se desarrolló la siguiente solución: lograr que la velocidad de giro de los vehículos fuera variable, rápida en los casos en los que se necesita y lenta en el resto para evitar estos «temblores» indeseados.
Esto se consiguió calculando la diferencia entre el vector velocidad del vehículo y el vector que indica la dirección hacia la que se encuentra su objetivo. Si la diferencia entre estos vectores es mayor de 90^o, se considera que se necesita hacer un giro pronunciado, y se aplica la velocidad de rotación normal, sin embargo si es menor, se trata de un giro pequeño y se aplica una velocidad de rotación menor para que el giro resultante sea más reducido.
De esta forma, gracias a esta solución, los vehículos podían realizar giros más suaves por lo que no tenían que estar constantemente corrigiendo su trayectoria. Sin embargo, aunque supuso una gran mejora, no se consideró suficiente, y se planteó una segunda solución adicional: que aunque estos temblores tengan lugar realmente, «ocultarlos» al usuario. Esta solución se aplica no solo a los vehículos del tráfico sino a todos los controlados por la inteligencia artificial.

La idea de esta solución es la siguiente: en lugar de pintar el vehículo con su verdadero ángulo, se guardan los últimos ángulos y se calcula su media, la cuál será la que se utilice para pintarlo, consiguiendo así una mayor suavidad en los cambios de ángulo.

Esto se ha implementado de la siguiente forma:
Para cada vehículo se guarda una lista con sus últimos cinco ángulos. Cuando vas a pintar el vehículo, antes de hacerlo, eliminas el ángulo más antiguo de la lista y añades el actual. Si la velocidad es negativa se realiza esto ya que siempre es necesario incorporar los nuevos ángulos a la lista, pero se pinta usando el ángulo normal, ya que marcha atrás no se producen estos «temblores» en el movimiento. Sin embargo, si la velocidad es positiva se aplica la fórmula

$$\text{angulo} = \frac{(a_0 + i_4) + (a_0 + i_3) + (a_0 + i_2) + (a_0 + i_1) + a_0}{5} \tag{C.1}$$

siendo a_0 el ángulo más antiguo de la lista y i_n el incremento del ángulo desde el ángulo a_0 hasta a_n.

C.5.4. Path-finding

Para realizar la búsqueda de caminos (*path-finding*) se hace uso del algoritmo de búsqueda A^*[15], el cual necesita conocer la estructura de las interconexiones entre nodos y sus distancias. Como se hace uso de este algoritmo muy a menudo (cada vez que tenemos un nuevo objetivo o cuando el objetivo ha cambiado de calle), esta relación de interconexiones y sus distancias se calcula una única vez al iniciarse el servidor, antes de que dé comienzo la partida, por lo que después el acceso a estos datos se realiza con un bajo coste.

El algoritmo del *path-finding* no es el único que accede a esta estructura de nodos, sino qué también se utiliza para calcular la distancia entre dos nodos cualesquiera. Esto es necesario ya que cuando se calcula el camino a seguir se busca el nodo más cercano a tu posición y se forma el camino a partir de él, existiendo la posibilidad de que ese primer nodo esté en sentido contrario al objetivo (ver Figura C.20). Para evitar que esto suceda, se calcula si el primer nodo está más alejado del objetivo que el vehículo, y en ese caso se coge como primer nodo el siguiente, que ya estará bien encaminado.

Cómo por este motivo se requiere con mucha frecuencia conocer la distancia entre

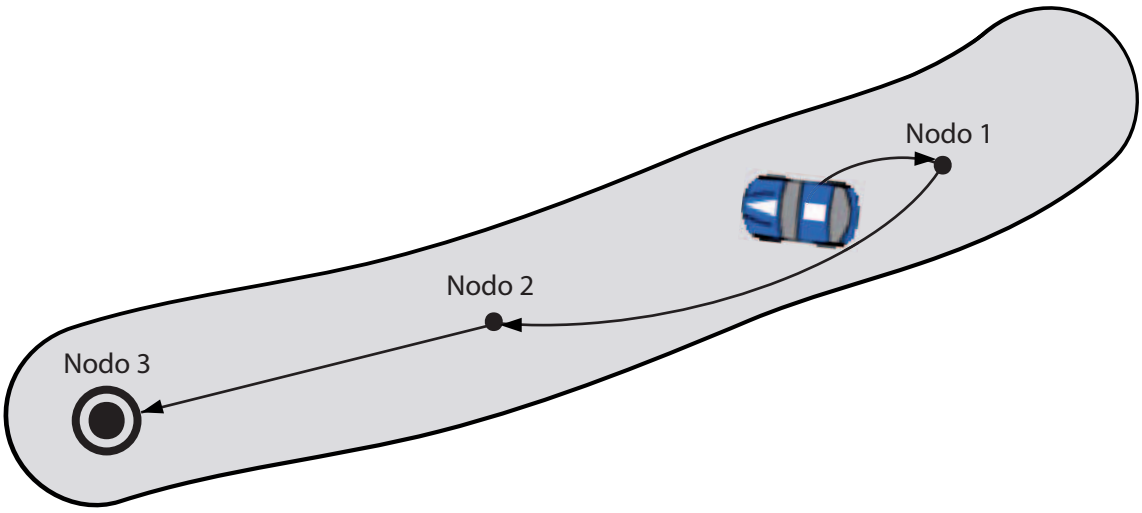


Figura C.20: Camino con primer nodo en sentido opuesto

dos nodos, y sólo es conocida la distancia entre nodos que estén interconectados, es habitual tener que realizar cálculos de la distancia vectorial entre nodos no interconectados, y muchas veces se repiten los cálculos. Por ello, se decidió que después de calcular la distancia, almacenarla de forma que la próxima vez que

[15] obtenido de `http://code.google.com/p/jianwikis/wiki/AStarAlgorithmForPathPlanning`

se requiriera sólo hiciera falta consultarla, y después de realizar varias pruebas se comprobó que en efecto esto suponía una reducción considerable del número de cálculos necesarios. Los resultados de estas pruebas se pueden ver en la Tabla C.3.

Nodos interconectados		18k
Nodos no interconectados	se repiten	11k
	no se repiten	6k

Tabla C.3: Número de usos del cálculo de la distancia entre dos nodos

Otro aspecto importante del *path-finding* a tener en cuenta es que cómo los vehículos del tráfico y los enemigos tienen distintas zonas por las que les es permitido circular, se ha desarrollado el algoritmo $A*$ y la estructura de interconexiones de nodos de forma duplicada, una para cada tipo de vehículo. Además, en el caso del algoritmo para el tráfico, se tienen en cuenta los sentidos de las calles de una única dirección, para evitar que tomen caminos que en la vida real no son correctos.

C.5.5. Normas de circulación

Como se ha mencionado en la sección anterior, el tráfico respeta el sentido de la circulación en las calles de un solo sentido, de forma que no obtienen caminos que atraviesen calles en contradirección. Sin embargo, en el resto de calles no respetan la norma de circulación de circular por el carril derecho de la vía.

Esta característica no se implementó ya que resultó muy complicado idear un método sencillo para que los coches circulasen únicamente por el lado derecho de los caminos, y además se encontró el inconveniente de que, como los vehículos pasarían de ocupar toda la calzada a únicamente los carriles de su sentido, tendrían menos espacio para maniobrar y por lo tanto empeoraría sustancialmente la efectividad del manejo de los vehículos por parte de la inteligencia artificial.

Por el esfuerzo requerido para lograr esta característica, se consideró que realizar una simulación detallada de las normas de circulación hasta este nivel de precisión estaba fuera del alcance del Proyecto Fin de Carrera, y se decidió garantizar únicamente el sentido de circulación de las calles de un único sentido.

C.6. Funcionamiento en red

En esta sección se tratarán aspectos relacionados con el modelo de red utilizado en `Vanet-X`, incluyendo desde el funcionamiento básico hasta las técnicas de mejora y optimizaciones realizadas.

C.6.1. Funcionamiento básico

El esquema del funcionamiento es el siguiente:

|Cliente| crea y envía *InputSnapshot* (contiene los eventos de teclado) al servidor.

|Servidor| elimina los clientes que lleven varios ciclos sin enviar nada.

|S| recibe los *Snapshots* enviados por los clientes.

|S| para cada *InputSnapshot* recibido, resetea sus acumuladores si es necesario (ver sección C.6.5), establece cual es la secuencia del último *ack* de dicho cliente y almacena sus datos (eventos de teclado) para que los use posteriormente. El actor *Player* es asíncrono y pedirá usar estos datos de forma asíncrona, siendo entonces cuando se establezca la secuencia del último estado recibido por dicho cliente.

|S| crea un *Snapshot* personalizado para cada cliente (con solo lo que está dentro de una determinada distancia de su jugador) y lo envía.

|C| recibe todas las Snapshots enviadas desde la última recepción y se queda con la más actual, guardándola en un buffer y marcando si es *buena* o no (será *buena* si hace *ack* al último estado). En el caso de que sea *buena* se almacena también como último estado recibido (para que el siguiente *InputSnapshot* que generemos utilice la secuencia de dicho estado como nueva secuencia de *ack*, y así se evite el problema de añadir latencia artificialmente a la conexión al realizar interpolación).

|C| coge el primer *Snapshot* del buffer (o no coge nada si aun no hay suficientes elementos en dicho buffer).

|C| si dicho *Snapshot* es bueno, aplica con el anterior *Snapshot* procesado la descompresión delta para con esos datos actualizar el estado de juego del cliente.

Otros aspectos importantes del funcionamiento de red son los siguientes:

- El servidor se crea con direccion de red *wildcar* para que acepte conexiones de todas las interfaces, y el cliente se unirá siempre a la dirección IP privada. De esta forma se evitan problemas con ciertas configuraciones de red con router + switch, que hacen que no sea posible conectarse a tu propia IP pública desde dentro de dicha red.

- El servidor espera un tiempo establecido a que se conecte el cliente y si pasado ese tiempo el cliente no se ha unido, el servidor considerará que ha habido un error en la inicialización del cliente y se concluirá. Este comportamiento no se da al crear un servidor dedicado (ver sección D.2.1).
- Cuando el cliente trate de unirse a una partida, el servidor comprobará la cantidad de jugadores conectados a la partida, y si está llena mostrará indicándotelo.
- Si se pierde la conexión el cliente muestra un error sale al menú principal. En el caso de que el error haya sido solo por parte del cliente, el servidor seguirá funcionando si quedan más jugadores y el cliente desconectado no es el jugador 1.

Uso de Sockets (TCP y UDP combinado)

Una característica de los videojuegos de tiempo real (un juego de coches es uno de ellos) es que los mensajes tienen que llegar lo más rápido posible, ya que si llegan más tarde de lo previsto generalmente no sirven de nada ya que el estado del "mundo de juego"habrá cambiado. Es por esta razón por la que la fiabilidad no es una prestación interesante ya que si se pierde un paquete, conseguirás entregarlo tarde, por lo que no servirá de nada, y además se habrá sobrecargado la red.
Para disimular este efecto (llamado latencia o *lag*) que se obtiene al perder paquetes de datos, se introducen técnicas como la predicción, que consiste en que el cliente también tiene una copia local del «mundo» y de los métodos que va a ejecutar el servidor, y ejecuta todo igual que el servidor con la esperanza de llegar a los mismos resultados.
Estos resultados el cliente los da temporalmente por buenos y los utiliza para pintar por pantalla y que así, aunque tarde en responder el servidor, el cliente pueda jugar fluido y sin parones.
Posteriormente, si cuando al cliente le llega el paquete que ha enviado el servidor con sus resultados (que son los validos) se comprueba que el cliente se había equivocado en su predicción, se suelen utilizar otras técnicas para corregir el estado local del cliente sin que el jugador lo note. Éstas últimas técnicas no se han aplicado por considerarse fuera de los objetivos del Proyecto Fin de Carrera.

La razón de utilizar *sockets* en lugar de *RMI* o alguna otra tecnología de más alto nivel es que, como se acaba de explicar, para este modelo de red es esencial el rendimiento del protocolo, ya que se van a transmitir paquetes de datos un mínimo de 25 veces por segundo, sin necesidad de fiabilidad, y todos los protocolos de más alto nivel introducen muchas mejoras pero generalmente a costa del rendimiento [21].

El uso de los siguientes protocolos es el siguiente: se usa TCP para enviar el estado inicial del servidor antes de que comience el juego y UDP a partir de ese momento, ya que una vez comenzado el juego no importa que se pierdan mensajes, pero es necesario que el estado inicial llegue correctamente a todos los jugadores, y utilizar TCP evita programar dichas características de fiabilidad. Como desventaja de este uso de TCP, esto supone otro puerto extra que necesita abrirse en los NAT y firewalls.

Apertura de puertos

Para el correcto funcionamiento de este modelo de red, como ocurre en muchos juegos, es necesario abrir puertos si estas detrás de un NAT.

Fueron estudiadas diferentes posibilidades de evitar esto. Una era utilizar el protocolo *UPnP*, como algunos clientes de P2P, pero esta solución estaba muy desaconsejada en varios foros dedicados al tema de desarrollo de videojuegos ya que existen grandes problemas de seguridad en el protocolo.
Otra posibilidad estudiada es el *NAT Punch-through*. Este método se basa en que exista un servidor que no esté detrás de un NAT y a través de él se pongan en contacto los clientes y se averigüe el puerto que usa cada uno. Una implementación de este método es Raknet[16].
El problema de este método radica en la necesidad de tener siempre un servidor maestro en funcionamiento.

Finalmente, como los jugadores habituales están acostumbrados a este requerimiento de abrir puertos, y dado que aplicar estas soluciones está fuera del alcance del Proyecto Fin de Carrera, se decidió seguir con el requerimiento de abrir los puertos necesarios para jugar.

Control del flujo de mensajes

El manejo de las comunicaciones UDP no se realiza en un hilo separado sino en el hilo principal de ejecución.
Esto es porque la principal ventaja de tener las comunicaciones en un hilo separado es poder recibir los paquetes en cuanto estén, sin tener que esperar a que el hilo principal termine de hacer los cálculos de movimiento, colisiones y pintar la pantalla.
Sin embargo, aunque tuviésemos el paquete en el mismo instante en que se recibe, no vamos a usarlo hasta que el hilo principal no llame a la función de actualizar los datos, con lo que lo único que estaríamos haciendo es meter el paquete en un buffer y esperar a que el hilo principal lo consuma, que es lo mismo que se consigue

[16] `http://www.jenkinssoftware.com/raknet/manual/natpunchthrough.html`

con el comportamiento normal de los sockets (conforme van llegando los paquetes se almacenan en un buffer hasta que tú los pides) [13].

C.6.2. Interpolación-extrapolación

La interpolación y la extrapolación son métodos aplicados en el cliente para conseguir una representación fluida del resto de actores cuando la conexión con el servidor no es lo suficientemente buena.

El problema de un modelo de red básico, sin la aplicación de estos métodos, es el siguiente. En cada ciclo, el servidor envía al cliente un estado (*Snapshot*) con la actualización del mundo de juego. Para enviar el mínimo de datos necesarios, dicho estado se realiza calculando la diferencia delta respecto al anterior estado que se sabe que ha procesado el cliente.
Este dato es conocido en el servidor ya que el cliente, en cada InputSnapshot que envía al servidor, le informa de cuál es el último estado procesado.
El problema radica en que, debido a la latencia de la conexión, la información de que el cliente ha procesado un nuevo estado tarda en llegar al servidor, por lo que éste le puede seguir enviando nuevos estados creados respecto al que se cree que es el estado del cliente pero que en realidad ya no lo es. Cuando estos estados lleguen al cliente, no podrán ser descomprimidos ya que se han creado en base a un estado anterior al actual. Son lo que vamos a llamar un Snapshot *no bueno*. Y si en un ciclo no se recibe un estado *bueno* (se recibe uno *no bueno* o no se recibe ninguno), no se puede actualizar el estado del mundo de juego, permaneciendo sin cambios. Y no que no se reciban estados buenos de forma constante implica que los actores avancen «a trompicones» por el escenario.

Para evitar este efecto visual indeseado, se aplica la siguiente idea: tener un buffer de estados recibidos más nuevos que el que se va a procesar, de forma que si uno no es bueno, se pueda interpolar la posición del actor con los datos de uno de los estados almacenados «futuros» (Ver Figura C.21).
Esto tiene el inconveniente de añadir más latencia artificialmente, ya que la única forma de tener ese buffer de estados futuros es procesar los estados no al recibirlos sino con un cierto retraso establecido (Ver Figura C.22).

Cuando los estados malos recibidos de forma consecutiva superan la capacidad del buffer de interpolación (dicha capacidad es modificable desde el fichero de texto *ParamConfig.txt*), se procede a aplicar otro método diferente: la extrapolación.

Ésta simplemente consiste en tomar como base los últimos datos válidos recibidos de cada actor y extrapolarlos al estado que se requiere representar.

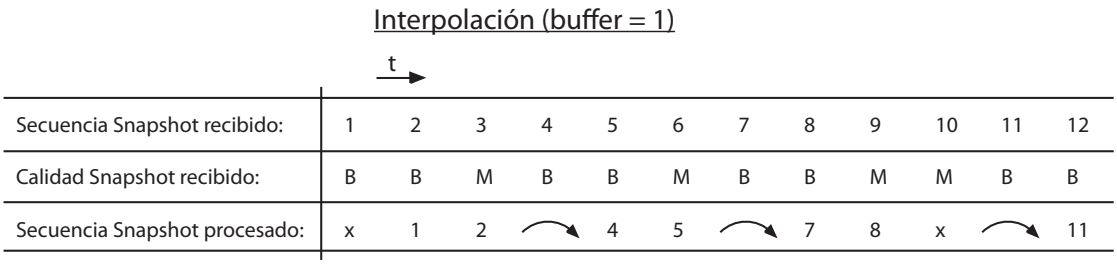


Figura C.21: Interpolación con buffer de 1 estado. La calidad de un Snapshot recibido puede ser buena (B) o mala (M). Una X en la secuencia del Snapshot procesado indica que no se ha procesado ninguno, ya que no había ocupación suficiente del buffer, mientras que una flecha indica que se realiza interpolación.

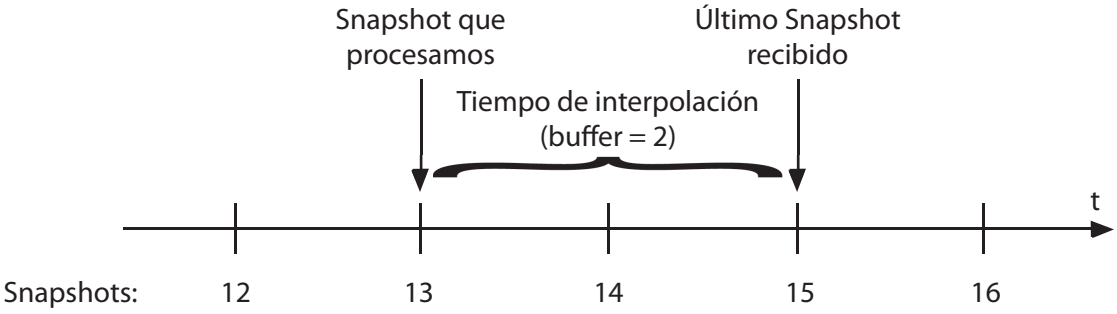


Figura C.22: Diferencia entre el último estado recibido y el estado que se pinta en pantalla, debido al buffer de interpolación

Un ejemplo simplificado es el siguiente: el coche A tiene posición (13,45) y una velocidad por ciclo de (2,1), por lo tanto se puede aventurar que hay muchas probabilidades de que en el siguiente estado su posición sea (15,46).

El problema de la extrapolación es que la precisión de los resultados decrece rápidamente conforme más estados se deban extrapolar. Es decir, estando en el estado *X* es fácil acertar la posición que tendrá en el estado *X+1*, pero si quieres calcular la posición *X+10* es probable que el margen de error sea demasiado grande. Esto es así ya que en la posición de un vehículo intervienen más elementos, como por ejemplo colisiones contra otros vehículos o contra elementos del terreno. Por tanto, se ha optado por realizar la extrapolación solo hasta una cantidad definida en el fichero de texto *ParamConfig.txt*, no actuando más allá de ese valor por considerarse que el error puede ser demasiado grande.

Hay que anotar que, aunque en el motor *Source* [22] se realiza una extrapolación al uso, en `Vanet-X` se ha considerado conveniente intentar mejorar el margen de error usando una simplificación de la predicción (que se explicará a continuación), es decir, en lugar de simplemente actualizar la posición teniendo en cuenta el vector velocidad, se aplica también las mismas reglas que en el servidor si se salen de la calzada o están sobre un terreno que disminuya la velocidad. De esta forma, con muy poco procesamiento adicional, se garantiza una mayor precisión del resultado que obtengamos.

C.6.3. Predicción

Otro de los problemas causados por la latencia de la conexión es que, al igual que el resto de actores, el vehículo del jugador no tiene un movimiento fluido y además, debido a que los eventos de teclado deben enviarse al servidor y éste devolver un estado actualizado, las teclas pulsadas se ven reflejadas con retraso, siendo muy incómodo para el jugador.
Este problema de que las acciones del jugador se vean representadas con retraso viene dado por el esquema cliente-servidor tradicional, con servidor autoritativo, en el que se debe esperar a la respuesta del servidor a tus acciones antes de pintarlas. Ver Figura C.23.

La predicción consiste en lo siguiente: el cliente trata de predecir con el menor error posible el resultado que devolverá el servidor, y lo aplica de forma que en el cliente se ven reflejados los cambios del jugador instantáneamente. Posteriormente, cuando recibimos el resultado del servidor, se aplica, y sobre esta nueva posición se vuelven a realizar las predicciones restantes (para lograr el movimiento instantáneo del jugador se realiza la predicción de cada uno de los estados enviados al servidor

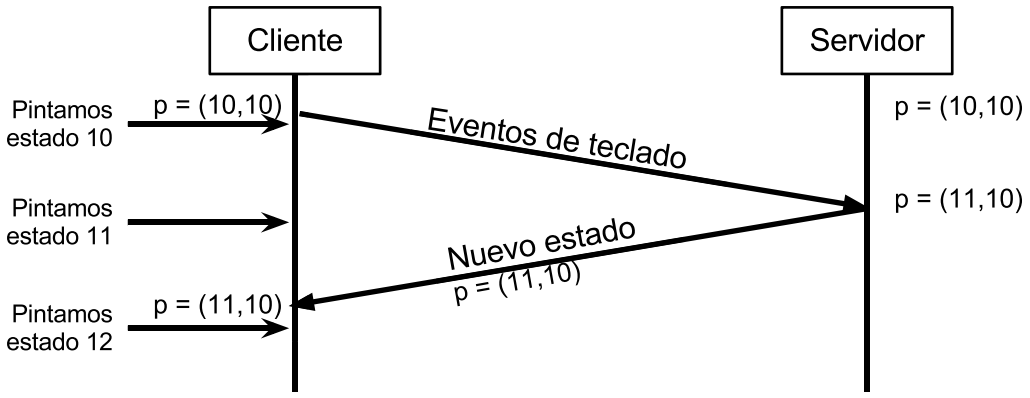


Figura C.23: Efecto de la latencia de red

pero aún no contestados). De esta forma se corrigen los posibles errores cometidos en la predicción, aunque de forma algo brusca.

Para evitar esta brusquedad en la corrección se podría aplicar un suavizado para que esa corrección se realice a lo largo de varios ciclos en lugar de instantáneamente (como se realiza en el motor *Source* [22]), pero se ha considerado que no era un aspecto imprescindible para los objetivos de este Proyecto Fin de Carrera y por lo tanto se ha dejado para mejoras futuras.

Para realizar la predicción se siguen los siguientes pasos: se crea un buffer que contiene los eventos de teclado de cada estado. Cuando se envía un InputSnapshot al servidor se almacenan junto con su número de secuencia, y al recibir un Snapshot del servidor se eliminan los eventos de teclado cuyo estado ya haya sido superado.

Posteriormente, tanto si se procesa un Snapshot bueno como si no, se aplica la predicción sobre el vehículo del jugador de la siguiente forma: para cada estado que se lleve de retraso (la diferencia entre el último que se haya recibido del servidor y el actual del cliente) se cogen los eventos de teclado del primer estado almacenado en el buffer y se simulan los cálculos que se realizan en el servidor y se calculan las colisiones (desactivables en *ParamConfig.txt*). Es importante tener en cuenta que los cálculos de colisiones son una mera aproximación ya que se comprueba la colisión del vehículo del jugador en el estado que se está prediciendo con el resto de actores en el último estado recibido del servidor.
Este proceso se repite para cada estado retrasado partiendo como base del resultado de la predicción del estado anterior.

C.6.4. Compresión delta

Para reducir la cantidad de datos enviados, se aplica el método conocido como *compresión delta*[17], que consiste en no enviar toda la información sino solo la diferencia respecto al último envío.

Esto puede realizarse de dos maneras. La primera es diferencia a nivel byte, que consistiría en que si en el estado anterior se ha enviado *posición=(12,15)* y ahora se quiere enviar *posición=(14,14)*, se envíe *posición=(+2,-1)*, de forma que en valores muy grandes se pueda reducir el tamaño requerido para enviarlos (ejemplo: en lugar de enviar un valor de tipo *Long*, enviar un *Short* con la diferencia).
La otra manera de aplicarlo, es la diferencia a nivel de aplicación, que consistiría en enviar únicamente los elementos de la estructura que hayan tenido cambios. Por ejemplo si de un vehículo se mantiene la velocidad constante y solo cambia la posición, enviar únicamente esta última, ahorrando así enviar la velocidad.
También existe la posibilidad de combinar ambas formas, aplicando la diferencia respecto a la estructura, y en los elementos que quedan para enviar, aplicar la diferencia respecto a byte, pero ha sido el segundo método (diferencia a nivel de aplicación) la forma elegida en este Proyecto Fin de Carrera.

Para poder realizar la compresión delta, es necesario que el servidor almacene cual es la secuencia del último estado aplicado por cada cliente, dado que no todos los clientes tendrán el mismo estado, y también almacene todos los datos de los últimos estados, desde el más antiguo de los aplicados en los clientes hasta el último.
Con estos datos, el servidor le envía a cada cliente los datos resultantes de realizar la compresión delta entre el estado actual del servidor y el estado actual (conocido) del cliente.
Al realizar la compresión delta se coge el Snapshot que se toma como base para la compresión delta y para cada actor se comprueba si cada valor de la estructura (posición, velocidad, etc.) ha sido modificado y se anota, para que solo se envíen los elementos modificados.

En el lado del cliente únicamente se necesita tener almacenado el último estado procesado, para descomprimir el Snapshot que se reciba.

C.6.5. Envío de solo actores cercanos

Inicialmente se enviaban los datos de todos los actores (aparte de la compresión delta) sin importar si estaban cerca del jugador y por lo tanto eran de interés o estaban lejanos y daban igual sus datos (ya que para el cliente no es necesario

[17] `https://en.wikipedia.org/wiki/Delta_encoding`

saber el ángulo o la velocidad de un coche que está en la otra punta del mapa). Para reducir el tamaño de los paquetes de red que se han de enviar, se ha optado por enviar todos los actores, pero si están lejanos solo enviar su identificador -necesario siempre- y un booleano que indique lejanía.

Esto introduce un problema, y es que para la compresión delta ya no hay que comprobar si una variable de un actor ha tenido modificaciones respecto al estado anterior sino respecto al último estado que se sabe que ha recibido el cliente en el que los datos del actor hayan sido enviados (porque el actor estaba cercano).

Un ejemplo de este problema se muestra en la Tabla C.4.

Estado enviado	10	11	12	13	14
Lejano	NO	SÍ	SÍ	NO	NO
Valor enviado	25	45	45	45	45
Ha cambiado (en servidor)	?	SÍ	NO	NO	NO
Valor recibido	25	-	-	**45**	45
Aplicar cambios (en cliente)	?	-	-	**NO**	NO

Tabla C.4: Problema de no enviar actores lejanos

Para solucionar este problema, se introduce un «acumulador» donde se acumulen los cambios realizados a cada variable de un actor entre diferentes estados (Ver Tabla C.5).

Estado enviado	10	11	12	13	14
Lejano	NO	SÍ	SÍ	NO	NO
Valor enviado	25	45	45	45	45
Ha cambiado (en servidor)	?	SÍ	NO	NO	NO
Valor recibido	25	-	-	**45**	45
Acumulador	?	SÍ	SÍ	SÍ	NO
Aplicar cambios (en cliente)	?	-	-	**SÍ**	NO

Tabla C.5: Se crean los acumuladores como solución para poder enviar solo actores cercanos

El funcionamiento de los acumuladores es el siguiente. Cuando el servidor calcula para cada cliente que valores han cambiado (y por lo tanto deben enviarse), si el cliente está lejano, se almacena en su acumulador el resultado de realizar un *OR* entre el valor booleano que se acaba de calcular (que indica si ha habido cambio respecto del último estado), y el valor almacenado previamente.

Si el cliente vuelve a estar cercano, para determinar si debe aplicarse el cambio, se realiza un *OR* entre el valor booleano del acumulador y el que se calcula respecto del último estado, restableciendo posteriormente los valores del acumulador.

Hay que anotar que este restablecimiento de los valores del acumulador no se realiza instantáneamente sino cuando se recibe la confirmación (*ack*) de que el cliente ha recibido el Snapshot en el que se le enviaban los cambios.

También hay que tener en cuenta que no se puede acumular todo en el mismo actor ya que lo que se acumula esta personalizado para cada cliente. Es decir, es necesario un acumulador por cada pareja actor-cliente.

Para el funcionamiento de las flechas que indican la dirección en la que se encuentran las banderas y los enemigos, es necesario conocer siempre la posición de estos tipos de actores, aunque estén lejos. Por esa razón se hace uso de una estructura de tipo lista que contiene las posiciones de los actores de este tipo que se han enviado como lejanos.

Una solución análoga a la explicada para el envío de únicamente actores cercanos hay que aplicarla a cualquier variable que indique cambio de un valor que sea establecida a cierta o falso manualmente.
Esto se aplica al envío de los eventos del radar asociado a cada jugador y a los objetivos de la ronda.

En la Tabla C.6 se observa una traza en la que se aprecia el funcionamiento de este método para el envío de los objetivos de la ronda.

C.6.6. Optimizaciones

Además de las técnicas arriba nombradas, se han realizado otras optimizaciones que se describen a continuación.

Uso de *Externalizable* en lugar de *Serializable*

El envío de los datos se realiza mediante el uso de *Externalization* en lugar de *Serialization* debido a que ofrece un mayor control y de esta forma se logra un menor tamaño de los paquetes enviados.[18]

Aún así, el uso de *ObjectOutputStream* introduce cabeceras nada despreciables (de 40 bytes o más), ya que se incluyen los descriptores de clase.
Para lograr reducir estas cabeceras, se hace uso de *obj.writeExternal(stream)* en

[18] `http://thejavacodemonkey.blogspot.com.es/2010/08/java-serialization-using-serializable.html`

PRUEBA CON LAG DE 1 ESTADO Y BUFFER PARA INTERPOLACIÓN DE 2

...
nueva.changed_lugaresObjetivos false acumulador.changed_lugaresObjetivos false
acumulador.changed_lugaresObjetivos <<- false
game:ronda.changed_objetivos //<-- *aqui se cambian los objetivos en el servidor*
nueva.changed_lugaresObjetivos **true** acumulador.changed_lugaresObjetivos false
acumulador.changed_lugaresObjetivos <<- **true**
resetearas en 74 //<-- *se marca que se resetee cuando llegue el ack del estado 74*
nueva.changed_lugaresObjetivos false acumulador.changed_lugaresObjetivos **true**
acumulador.changed_lugaresObjetivos <<- **true**
nueva.changed_lugaresObjetivos false acumulador.changed_lugaresObjetivos **true**
acumulador.changed_lugaresObjetivos <<- **true**
nueva.changed_lugaresObjetivos false acumulador.changed_lugaresObjetivos **true**
acumulador.changed_lugaresObjetivos <<- **true**
nueva.changed_lugaresObjetivos false acumulador.changed_lugaresObjetivos **true**
acumulador.changed_lugaresObjetivos <<- **true**
cl_processSnapshot:changed_lugaresObjetivos //<-- *se procesa en el cliente*
//<-- *llega el ack del estado 74, por lo que se resetea el acumulador a falso*
nueva.changed_lugaresObjetivos false acumulador.changed_lugaresObjetivos false
acumulador.changed_lugaresObjetivos <<- false
...

Tabla C.6: Traza de funcionamiento del acumulador

lugar de *stream.writeObject(obj)*[19], aunque no siempre, ya que es necesario que el paquete comience con los descriptores de clase de la estructura que contiene a las demás (es decir, el Snapshot).

Es importante tener en cuenta que si no se envía la información del descriptor de clase, en ocasiones es necesario enviar un byte extra para saber qué tipo de clase es y así poder hacer la creación del tipo oportuno.

Compactación de booleanos

Debido a que un valor del tipo booleano se puede representar con un único bit pero sin embargo se serializa ocupando un byte, se ha realizado una «compactación» de los diversos valores booleanos presentes en las estructuras, agrupándolos en valores de tipo *Byte*.

En las figuras C.24, C.25, C.26, C.27 y C.28 se muestran varios ejemplos de cómo se han realizado estas agrupaciones.

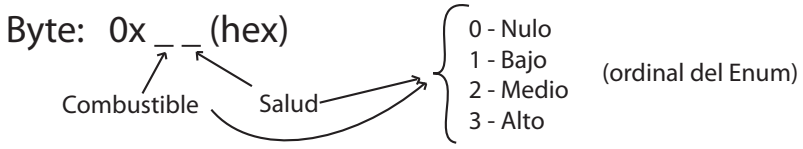


Figura C.24: Agrupación de booleanos de la clase *Opciones*

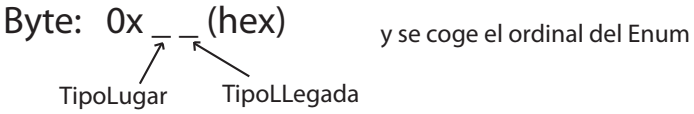


Figura C.25: Agrupación de booleanos de la clase *Tarea*

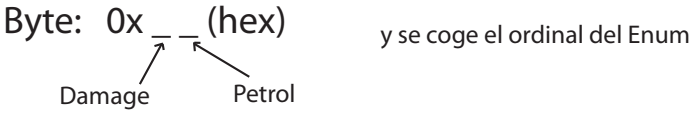


Figura C.26: Agrupación de booleanos de la clase *WaitingRoom*

[19] `http://docs.oracle.com/cd/E14571_01/web.1111/e13727/design_best_practices.htm`

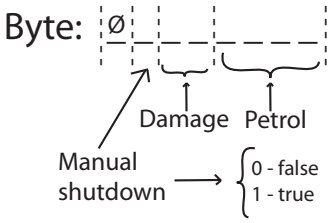


Figura C.27: Agrupación de booleanos de la clase *GameOver*

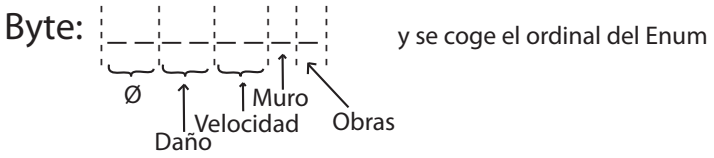


Figura C.28: Agrupación de booleanos de la clase *DanyoVelocidadYCalle*

C.6.7. Unión de jugadores a la partida

A pesar de que inicialmente se había planteado que la conexión de los jugadores a la partida se realizará de forma simultánea al comienzo de la misma, como suele suceder en la gran mayoría de los juegos de coches, más tarde se vio la necesidad de cambiar el modo de conexión ya que, al no tratarse de un juego de coches al uso, en el que tiene lugar una carrera y no tiene sentido unirte una vez empezada, en `Vanet-X` la mecánica de juego es muy diferente y tenía sentido dotar al juego de este tipo de conexión, que es la habitual en los juegos de acción.

Con este nuevo tipo de conexión que se decidió, la partida comienza cuando se ha conectado el primer jugador, y cualquier otro usuario que lo desee puede unirse sobre la marcha.

En la Figura C.29 se puede ver la secuencia temporal del proceso de conexión del primer jugador a la partida. Para los siguientes jugadores el proceso es el mismo, con la única diferencia de que la partida ya habrá comenzado anteriormente.

El proceso es el siguiente:
El servidor, después de crear todos los datos necesarios para comenzar la partida (terreno, elementos, etc.) crea una instancia de su *gestor de conexiones*, el cual es el encargado de todas las peticiones de los clientes que se realicen de forma asíncrona.

Las peticiones asíncronas son las siguientes: unión a la partida, información de la partida (para la sala de espera) y puntuaciones al acabar la partida. Todas ellas se realizan por TCP. El resto de transmisiones de red se realizan de forma síncrona: el envío del estado del mundo inicial al unirse el jugador a la partida

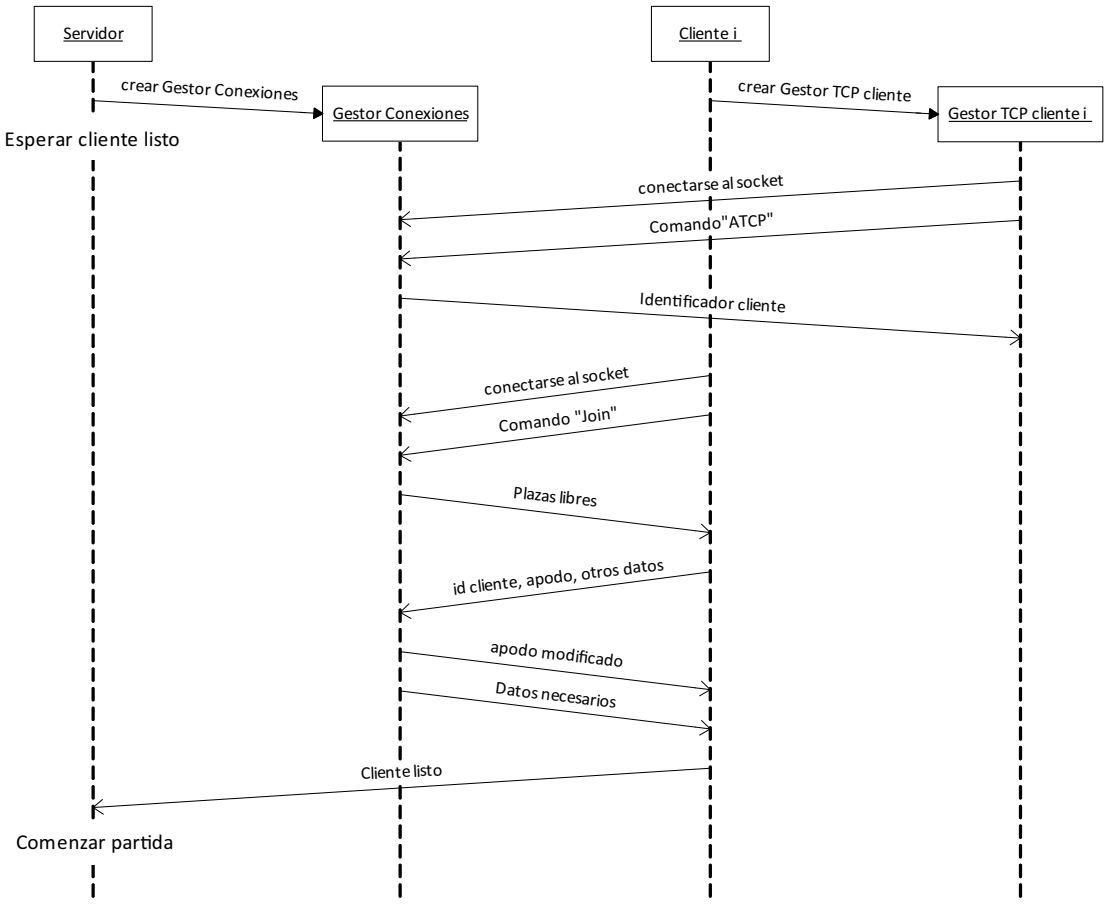


Figura C.29: Secuencia temporal de la unión de jugadores a la partida

(por TCP) y los envíos a los clientes del estado del mundo en cada ciclo del juego.

El cliente, al iniciarse, también crea su propio gestor de conexiones (llamado *gestor TCP del cliente*), ya que también existen transmisiones asíncronas del servidor al cliente.

La primera conexión es una petición del gestor del cliente al del servidor para obtener su identificador de cliente único, y también se realiza para almacenar dicha conexión para las futuras comunicaciones asíncronas con el cliente como destino.
Acto seguido, es el cliente el que se conecta con el gestor del servidor, con el objetivo de primero comprobar si quedan plazas libres en la partida (en caso contrario el cliente acaba y se vuelve al menú principal), y después el cliente envía su identificador (el cual había sido previamente recibido del servidor, y el objetivo de reenviarlo es asociar las conexiones síncrona y asíncrona), su apodo deseado y otros datos.
El servidor, al recibir el apodo, comprueba que no haya existido desde que se inició la partida otro jugador con el mismo apodo, y en caso contrario lo modifica (añadiéndole un número incremental entre paréntesis), reenviándoselo acto seguido al jugador. Finalmente, también envía el estado inicial del mundo y el estado correspondiente al ciclo actual, junto con la estructura que contiene la explicación de la ronda que se muestra al jugador y las puntuaciones actuales.
Tras esto, el cliente envía una señal al servidor indicando que está listo, para que éste le añada al juego y lo comience en caso de ser el primer jugador en unirse.

C.7. Modos de juego y gestión de rondas y objetivos

En esta sección se explicaran detalles de la implementación del modo «tareas» (que también está presente en los modos «supervivencia» y «aparcar» ya que parten de la misma base) y de los elementos introducidos por resultar necesarios para dichos modos (plazas de aparcamiento y capacidad de salir del coche).

C.7.1. Modo tareas

Los modos diferentes a «capturar las banderas» y «capturar los vehículos enemigos» (modos llamados «rally» por su semejanza al clásico videojuego Rally-X original), necesitan de la creación de nuevas estructuras que soporten los cambios de la mecánica de juego que suponen.

En este apartado se profundizará en los elementos para el modo de juego «tareas» ya que tanto «supervivencia» como «aparcar» tienen las mismas estructuras

con ligeras variaciones, y dado esta similitud no se ha considerado necesario explicarlas de nuevo por separado.

En este modo de juego, los objetivos ya no son actores (banderas o vehículos) sino elementos del terreno, que pueden ser lugares de interés (nodos que se han considerado relevantes por representar tiendas, hospitales, etc.) o direcciones. Y para estos objetivos ya no sólo existe la opción de lograrlos en coche sino que también se incluye la posibilidad de que se requiera lograrlo aparcando en una plaza de aparcamiento considerada cercana o lograrlo saliendo del coche (después de haberlo aparcado) y llegando a pie hasta el objetivo.

Por esta razón se ha creado una nueva estructura llamada «Tarea», que incluye todos los datos requeridos de un objetivo: tipo de lugar, tipo de llegada y distancia al primer y tercer aparcamiento más cercano. Los datos referidos a las distancias a aparcamientos son necesarios ya que cuando el tipo de llegada requerido es aparcando, la puntuación obtenida por el jugador no se calcula de la misma forma que habitualmente (más puntos contra menos tiempo se tarde en lograr el objetivo) sino que viene dada por una función (C.2) que bonifica la cercanía al objetivo. Además, solo cuentan como aparcamientos cercanos, y por lo tanto válidos, los tres más cercanos (salvo en el modo de juego «aparcar» ideado para la explotación, ver capítulo 3.5).

$$\text{puntos} = \frac{100 * \text{distancia del objetivo al párking}}{\text{distancia del objetivo al jugador}} \tag{C.2}$$

El tipo de llegada de la tarea viene dado por una función de probabilidad (ver Tabla C.7). En una misma ronda pueden existir hasta tres objetivos simultáneos, en cuyo caso ambos serán del mismo tipo (dirección o lugar de interés) y contarán con el mismo tipo de llegada.

Probabilidad	tipo
60 %	en coche
30 %	aparcar cerca
10 %	a pie

Tabla C.7: Elección de «tipo de llegada» en una tarea

Es importante anotar que cuando el objetivo generado es de tipo dirección, el objetivo en realidad representa solo un punto de la calle, y en los modos de llegada «en coche» y «a pie» se da por completado al acercarte a menos de 200 pixeles (25 metros) de distancia. El punto elegido siempre es el punto medio del primer segmento del camino.
Los objetivos de tipo dirección por defecto sólo se generan en calles con nombre, ya

que así si conoces la zona, sabes hacia donde encaminarte ya durante el tiempo de intermisión, sin tener que esperar a que comience la ronda y aparezca el objetivo en el mini-mapa. Sin embargo, si en el área descargada como escenario no existen calles con nombres, y tampoco existen lugares de interés, entonces sí que se generarán objetivos en calles sin nombres, para así poder jugar en el escenario.
Salvo que suceda este caso extraordinario en el que no existen calles con nombre, el algoritmo de decisión de nuevos objetivos tiene una probabilidad de 50 % de generar un objetivo de tipo dirección y el otro 50 % de que sea de tipo lugar de interés.

C.7.2. Plazas de aparcamiento

El primero de los elementos introducidos para soportar la mecánica del juego deseada en los modos «tareas» y «supervivencia» son las plazas de aparcamiento.

Las plazas de aparcamiento se deben pintar en los laterales de la calzada y alineadas a ella. Para conseguirlo, se introdujo el concepto de los «puntos de parking» de las calles: se calcula cual es el punto intermedio de cada segmento que forma la calle, y se proyecta en los ejes laterales del camino, creando dos «puntos de parking» uno a cada lado de la calzada. Estos puntos sólo se crearan si están a más de 30 pixeles (el valor del radio de una plaza de aparcamiento) de otras calles, para asegurar que los aparcamientos no sobresaldrían por la otra calle. Estos puntos calculados son los posibles puntos de aparición de las plazas de aparcamiento. En la Figura C.30 se puede observar los puntos intermedios de cada segmento (circunferencia verde) y sus correspondientes puntos de parking (circunferencia azul, remarcada por una exterior de color rojo para aumentar su visibilidad). Como se puede observar, en este caso no todos los puntos intermedios han generado puntos de parking, ya que en el caso de dos de ellos no se han generado por resultar demasiado cercanos a un cruce de caminos.

Cómo las plazas de aparcamiento sobresalen ligeramente en el interior de la calzada, se debe adaptar la inteligencia que controla los vehículos para que las esquive. Esto se ha logrado creando un obstáculo esférico en su posición e introduciéndolo en la lista de obstáculos que tiene en cuenta el *steering behavior* de *obstacle avoidance*. Evidentemente cuando un vehículo de tráfico se encuentra en estado «aparcar», se desactivará de su *steering behavior* el obstáculo esférico del parking objetivo para permitir atravesarlo y así estacionar en él.

Hubo un problema al realizar esta implementación y es que se descubrió que había ocasiones en las que los vehículos no esquivaban correctamente las plazas de aparcamiento. El motivo era que los obstáculos esféricos tenían su centro en el borde del camino, y si el vehículo se acercaba pegado al borde de la calzada, su *steering behavior* le indicaba que para esquivarlo podía torcer en las dos direcciones en lugar de sólo hacia el centro del camino. Este problema radica en el hecho de

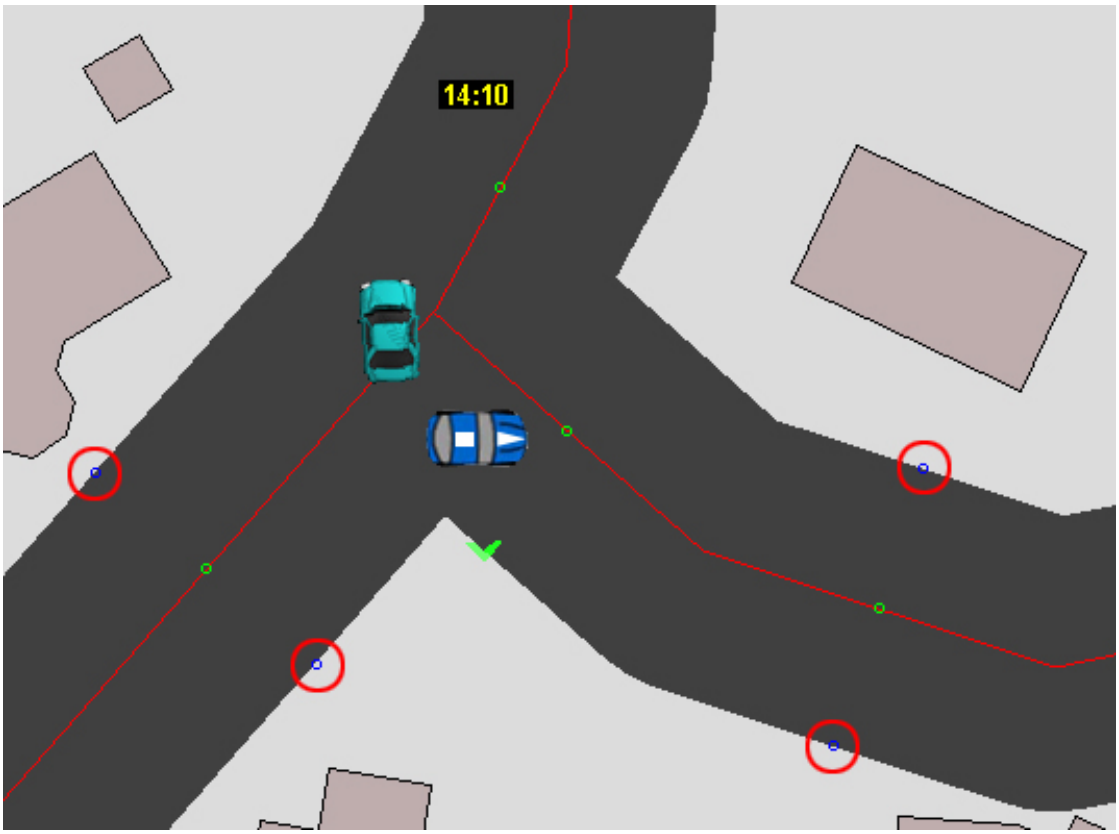


Figura C.30: Captura mostrando los puntos de parking (azul rodeado de una circunferencia roja) ylos puntos intermedios (verde)

haber implementado la unión de *steering behaviors* con un sistema de prioridades en lugar de uno aditivo (ver capítulo C.5.1).
Para solucionarlo, en lugar de optar por cambiar la implementación de la unión de *steering behaviors*, se tuvo la idea de incrementar el radio del obstáculo y colocar su centro más alejado del borde del camino, de forma que así el comportamiento de esquiva de obstáculos sólo devolviese la opción de girar hacia el interior de la calzada.
La contrapartida de éste método es que si este obstáculo esférico sobresale por otra calle debido a que se encuentra muy cercana, la inteligencia evitará esa zona que sobresale a pesar de que realmente no exista ninguna posible colisión.
En la Figura C.7 (pág. 161) se observa el obstáculo esférico de las plazas de aparcamiento (circunferencia azul).

Cuando el tipo de llegada es «aparcar», para que el jugador sepa cuáles son los aparcamientos válidos para lograr el objetivo, se pintan unas líneas desde dichos aparcamientos hasta el objetivo, indicando sobre ellas la puntuación que otorga cada aparcamiento.
Estas líneas no aparecen constantemente, ya que entonces encontrar una plaza de aparcamiento sería tan sencillo como acercarte al objetivo y luego seguir una de las líneas hasta el aparcamiento, sino que solo aparecen cuando estás a menos de una determinada distancia del aparcamiento.

C.7.3. Capacidad de salir del vehículo

El otro elemento introducido se trata de la capacidad del jugador para abandonar el vehículo y llegar a los objetivos caminando.

Para realizarlo, se ha decidido que el actor «Player» siempre sea el elemento bajo control del jugador, tanto cuando vaya en coche como a pie, y que en el caso de ir a pie se cambien sus constantes de velocidad y la imagen que se mostrará. Como además de estos parámetros hay otros muchos que también varían, se ha creado una estructura dentro de la clase «Player» en la que se guardan esos valores mientras el jugador va a pie y se recuperan cuando vuelve a montarse en el vehículo.

También se ha creado una nueva clase actor: «DummyPlayer», que tiene la apariencia de nuestro vehículo y colisiona con los mismos actores que cualquier otro vehículo, excepto con los jugadores, a los que permite que le atraviesen para que así el jugador pueda volver a montar en el vehículo.

C.7.4. Modo supervivencia

El modo supervivencia tiene una mecánica diferente al modo «tareas» pero se basa en los mismos elementos, únicamente modificando ciertos aspectos y funcio-

nes.

Las únicas diferencias destacables de la implementación son la modificación del algoritmo que decide qué objetivos crear y la modificación del finalizador de la partida, que ahora ya no es no completar los objetivos sino empezar la ronda con una cantidad negativa de puntos.

La decisión de qué objetivos crear en la ronda se ve complicada por la aparición de las rondas especiales. Éstas son rondas que aparecen cada mucho tiempo y que suponen que los jugadores deban completar el objetivo generado si no quieren perder una cantidad considerable de puntos.
Estos objetivos son lugares de interés donde se puede comer (restaurante, cafetería, etc.) o, en el caso de no existir lugares de interés de ese tipo, se crean dos «puestos de comida», que se crean con las mismas reglas que las plazas de aparcamiento y sirven para permitir que aparezcan estas rondas especiales en cualquier escenario independientemente de los tipos de lugares que incluya.

La decisión del tipo de objetivo sigue el siguiente esquema (inspirado en las rondas especiales del *modo zombi* del *Call of Duty: World at War*):

- No puede tocar realizar una ronda especial antes de la ronda tercera.
- A partir de esa ronda, la probabilidad aumenta un 10 % en cada ronda.
- Cuando se realiza una ronda especial, la probabilidad par la próxima ronda se establece en cero y no volverá a incrementarse hasta dos rondas más tarde.
- Si la ronda especial no tiene lugar, el objetivo tendrá una probabilidad del 50 % de ser de tipo dirección y otra tanta de ser de tipo lugar de interés, al igual que ocurría en el modo «tareas».

De esta forma garantizamos que este tipo de rondas no tengan lugar demasiado a menudo para que al jugador le dé tiempo a reunir los puntos suficientes para sobrevivir en caso de no lograr el objetivo de dicha ronda.

Anexo D

Sobre VESPA y la explotación

En este anexo se tratarán en detalle todos aquellos aspectos sobre VESPA y la explotación del juego como método de evaluación que no han podido ser tratados en el capítulo 3 o han sido tratados de forma resumida.

D.1. Vespa

En esta sección se realizará una breve introducción al sistema VESPA, se explicarán las interfaces desarrolladas así como la implementación desarrollada, también se explicará la implementación realizada para simular en el juego la existencia de atascos y por último se verán diversos problemas y posibles mejoras encontradas durante la implementación de VESPA.

D.1.1. Breve introducción a VESPA

VESPA (Vehicular Event Sharing with a mobile P2P Architecture) es un sistema diseñado para vehículos para compartir información en redes ad-hoc entre vehículos. La originalidad de VESPA es que soporta cualquier tipo de evento en la red (p.ej. plazas de aparcamiento libres, accidentes, frenados de emergencia, obstáculos en la calzada, información del tráfico en tiempo real, información relativa a la coordinación de vehículos en situaciones de emergencia, etc.).
VESPA se basa en el cálculo de una probabilidad de encuentro (EP) para determinar si un vehículo se encontrará con un determinado evento, en cuyo caso el sistema decidirá avisar al conductor. La probabilidad de encuentro también se usa para realizar de forma eficiente la diseminación de información entre vehículos.

En la Figura D.1 se muestra una visión general de los módulos de los que se compone el sistema VESPA (existe un módulo adicional llamado agregador de datos pero no se muestra en la figura ya que no ha sido implementado en la

implementación de VESPA elaborada para el juego).

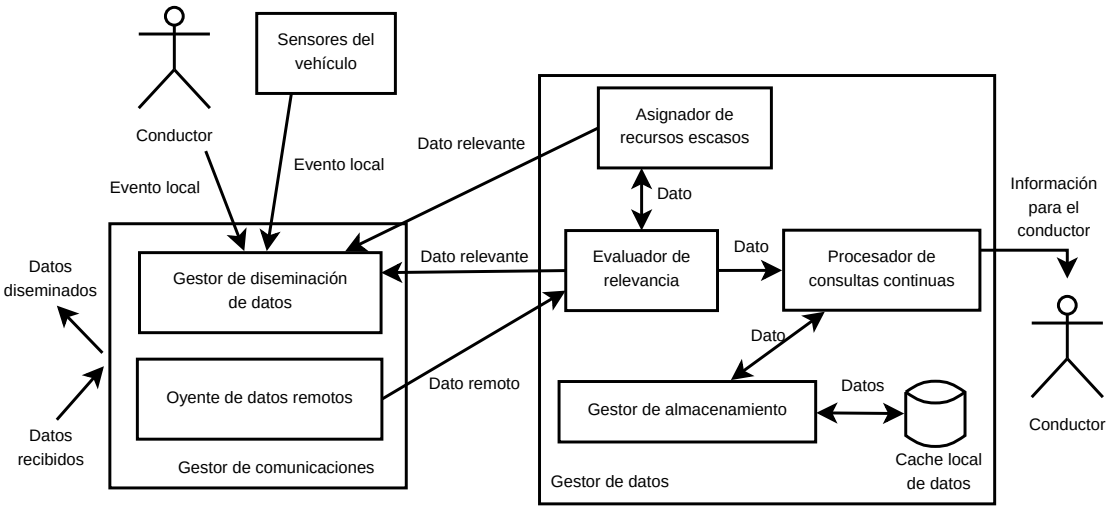


Figura D.1: Módulos del sistema VESPA

D.1.2. Interfaces desarrolladas

La implementación del sistema VESPA se ha realizado de forma que se permita una fácil sustitución por una implementación diferente de VESPA o incluso por otro *DMS (Data Management Strategy)*. Para ello se han definido las interfaces y clases abstractas básicas que se necesitan para poder interaccionar desde el juego con VESPA. Posteriormente se ha realizado una implementación «recortada» del sistema VESPA como instanciación de dichos interfaces.

Se han desarrollado dos tipos de interfaces diferentes:

- Una interfaz *IDataManagementStrategy*, que declara los métodos que deben ser implementados por una *DMS* para permitir su integración con el videojuego. En esta interfaz se han definido métodos para definir los tipos de eventos interesantes para el conductor, un método para generar un evento, etc.

- Otras cuatro interfaces que permiten la interacción del *DMS* con las entidades que implementan dichas interfaces (p.ej. obtener una referencia al gestor de datos del vehículo, obtener información sobre el historial de posiciones registradas en el dispositivo GPS, etc.). Estas cuatro interfaces son las siguientes: *IVisible*, que debe ser implementada por todas aquellas entidades que puedan ser «observadas» por el *DMS*, *IVehicle*, que debe ser implementada por todos los vehículos que equipen un sistema *DMS*, para permitir que

al DMS interactuar directamente con él, e *IPlayerVehicle* e *ITrafficVehicle*, que han de ser implementados por todos los vehículos equipados con sistema *DMS* de jugadores y del tráfico respectivamente.

Además de estas interfaces principales, también se han desarrollado las siguientes:

- Una interfaz *IEvent*, que representa a la estructura de un evento del sistema *DMS*. Esta interfaz no contiene ningún método ya que es una estructura que no se utiliza dentro del juego, únicamente de forma interna a la implementación del *DMS* y por ello se deja total libertad al desarrollador que implemente un nuevo *DMS*.

- Además, también se ha creado una clase *GenericEvent*, que implementa a *IEvent* y que ha sido dotada de la estructura y métodos que se han considerado básicos para cualquier *DMS*, de forma que pueda extenderse o utilizarse «tal cual» por un desarrollador externo encargado de implementar un *DMS* diferente.

- Otra interfaz *IDmsOptions*, que debe ser implementado de forma que contenga las opciones/parámetros necesarios para la inicialización del *DMS*.

- Por último una clase *FactoryDMS* que es una factoría para crear instancias de diferentes *DMS*. Esta clase contiene un único método *createDMS(...)*, que debe ser extendido para reconocer un valor específico de la propiedad Java *dataStrategy* (o un valor en un fichero de texto de configuración) que identifique la nueva *DMS* y cree la instancia de dicha *DMS* cuando se requiera.

Además también hay otras clases, ya implementadas, que deben ser tenidas en cuenta ya que se utilizan en los métodos de las interfaces arriba mencionadas:

- *ContextOfEvent*: Representa el contexto del evento (posición, tiempo, etc.).

- *GpsHistory*: Tiene el registro histórico de posiciones del vehículo y los métodos necesarios para calcular los vectores de dirección y movilidad.

- *Position2D*: Representa un vector bidimensional y tiene los métodos necesarios para su manipulación. Es el tipo de vector de posición usado en el juego.

- *Position4D*: Representa una posición espacio-temporal. Es el vector usado en *GenericEvent* y en la implementación de VESPA desarrollada.

- *RadarRepresentationOfEvent*: Representa un evento con el mínimo de información necesaria para ser representado en el radar.
- *WrapperParkingEvents*: Contiene una lista con los identificadores de las plazas de aparcamiento mostradas actualmente en el radar, junto con un objeto que sirve para realizar la sincronización de dicha lista.
- *WrapperRadarPoints*: Contiene una lista con la información mínima de cada evento de radar que debe ser representado, junto con un objeto que sirve para realizar la sincronización de dicha lista.

En la Figura D.2 se puede apreciar la arquitectura de la conexión entre el juego y el *DMS*.

D.1.3. Implementación desarrollada

A partir de las interfaces previamente creadas, se ha desarrollado una implementación del sistema VESPA que cumpla las características básicas del sistema. Únicamente no se han implementado las características del cálculo de la probabilidad de encuentro (*EP*) con mapas de carreteras digitales (en su lugar se realiza usando vectores geográficos) y la agregación de datos.

Para el desarrollo de esta implementación se ha tratado de dividir el trabajo en los módulos básicos que componen el sistema VESPA (ver Figura D.1), de forma que se pueda sustituir dicha implementación de VESPA por otra no solo de forma completa sino también solo los módulos precisos.
Para realizar esta implementación modular se ha hecho uso del patrón de diseño *Factory method* para cada uno de los módulos necesarios.

En la Figura D.3 se observa la estructura de la arquitectura VESPA implementada y en la Figura D.4 se muestra como se comunican los diferentes módulos entre sí junto con las principales funciones de los interfaces que los representan.

A continuación se detallan las ideas generales de la implementación de VESPA desarrollada, así como los algoritmos principales.

Existen cuatro hilos de ejecución: el del módulo sensor, encargado de supervisar el entorno y crear los eventos locales, el del módulo gestor de almacenamiento, encargado de comprobar periódicamente los eventos almacenados y eliminar los que correspondan, el del módulo procesador de consultas continuas, encargado de revisar los eventos almacenados y mostrar al conductor los que correspondan, y el del módulo de entrada de radio, encargado de la escucha de los eventos remotos enviados a través de mensajes de radio.

En las figuras D.5, D.6, D.7 y D.8 se muestran las tareas realizadas en cada hilo de ejecución.

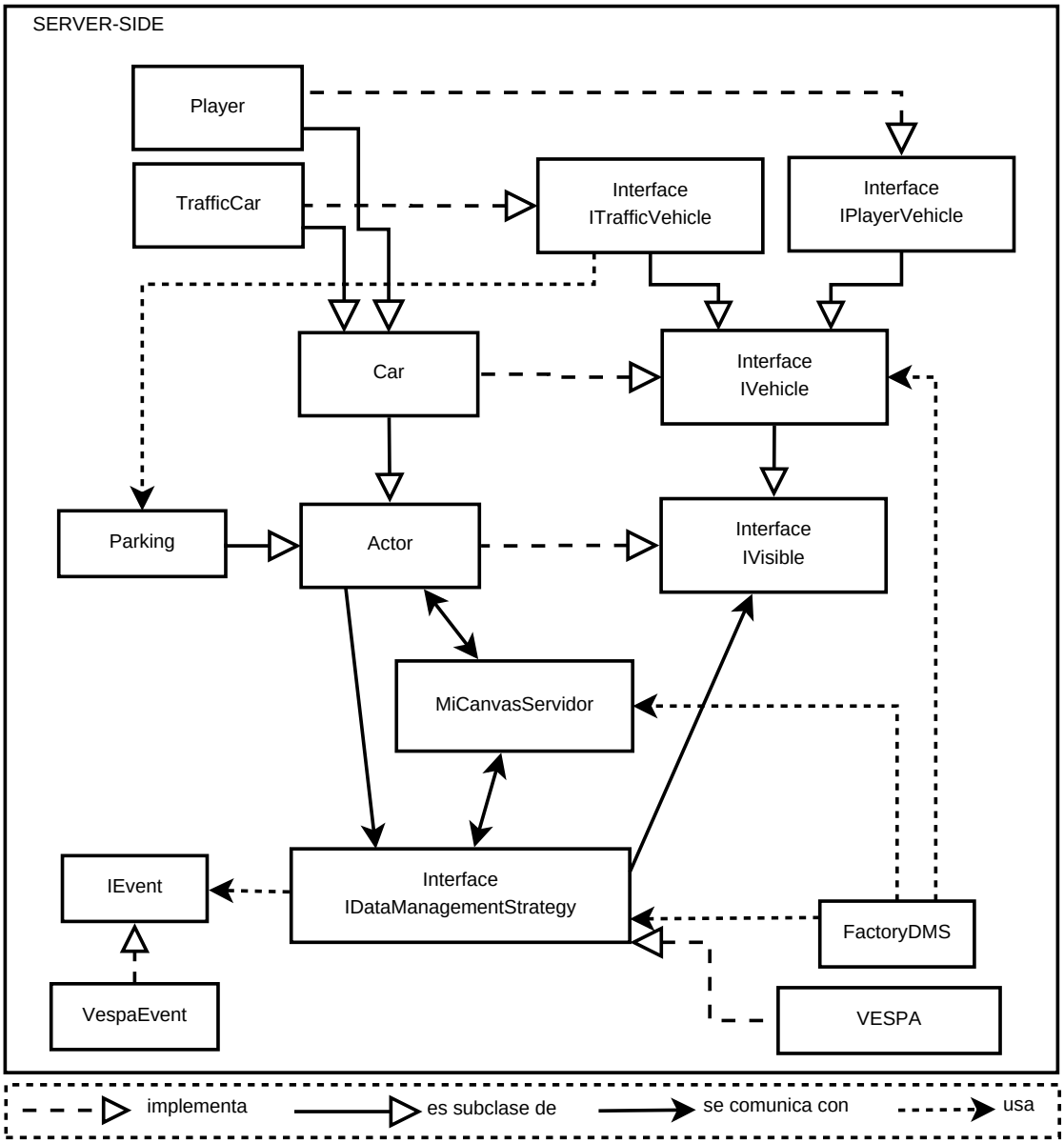


Figura D.2: Arquitectura de la la conexión entre el juego y el DMS

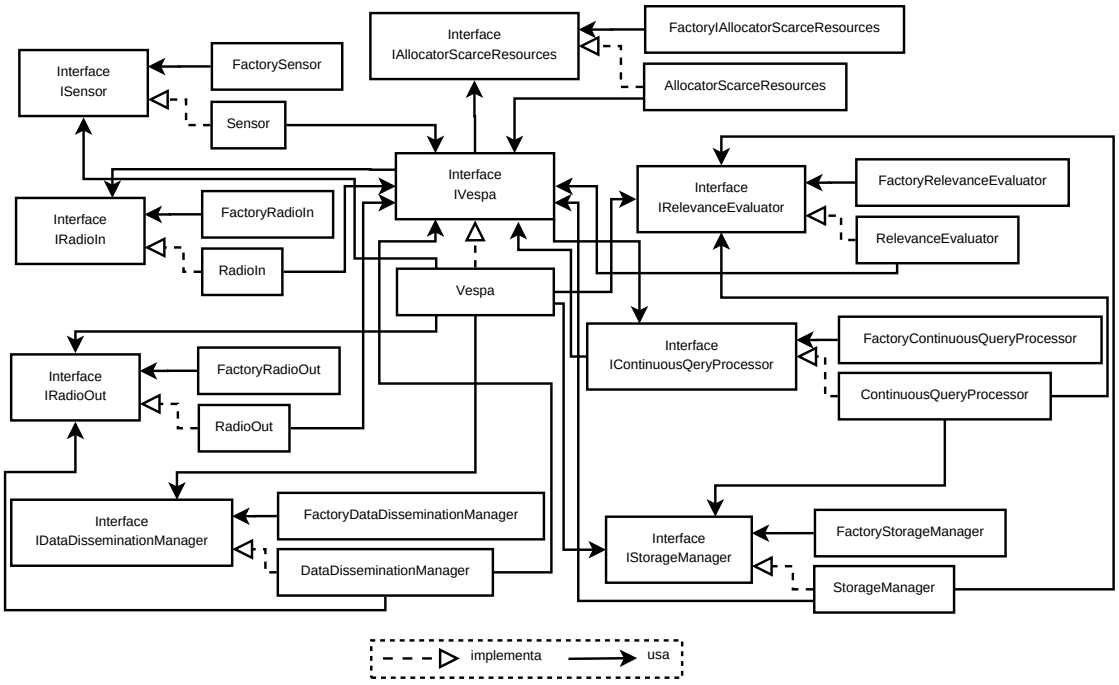


Figura D.3: Estructura de la arquitectura VESPA implementada

La estructura de un evento VESPA es la mostrada en la Tabla D.1.

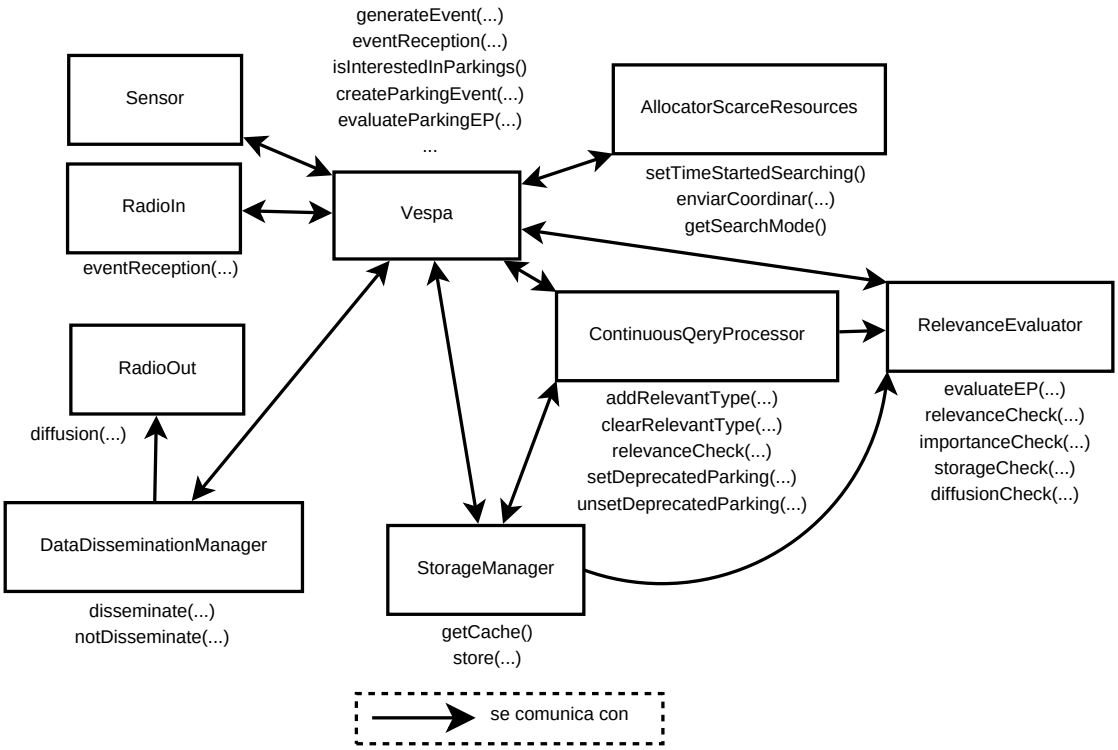


Figura D.4: Comunicación entre los diferentes módulos implementados

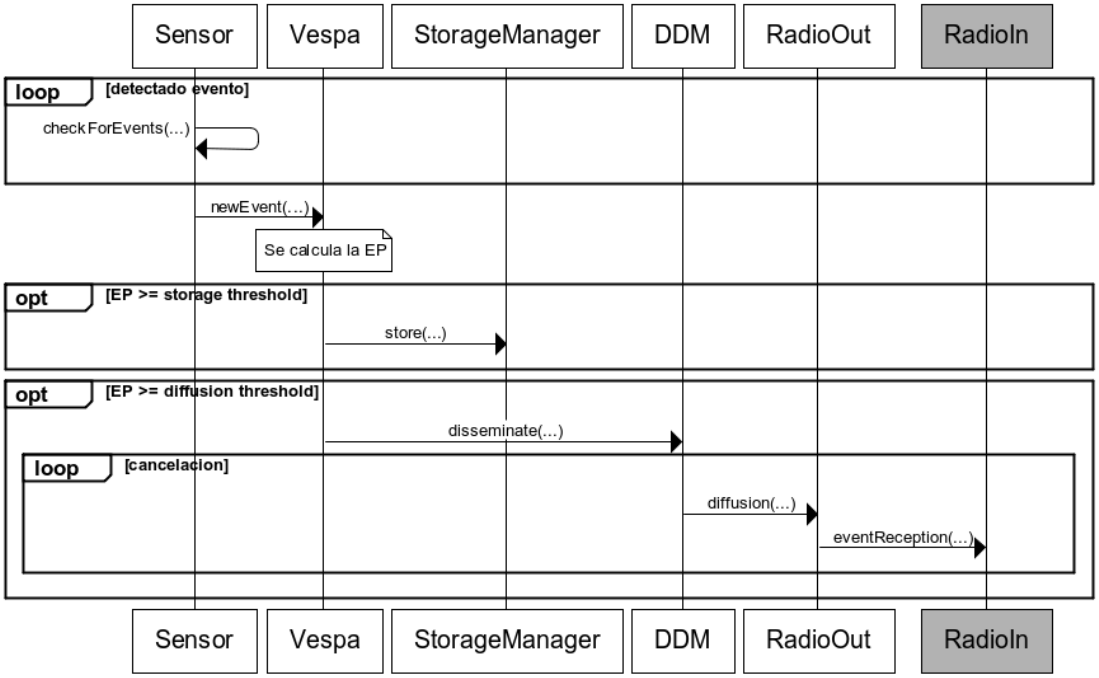


Figura D.5: Hilo de ejecución de la detección de un evento (el módulo sombreado es del vehículo receptor)

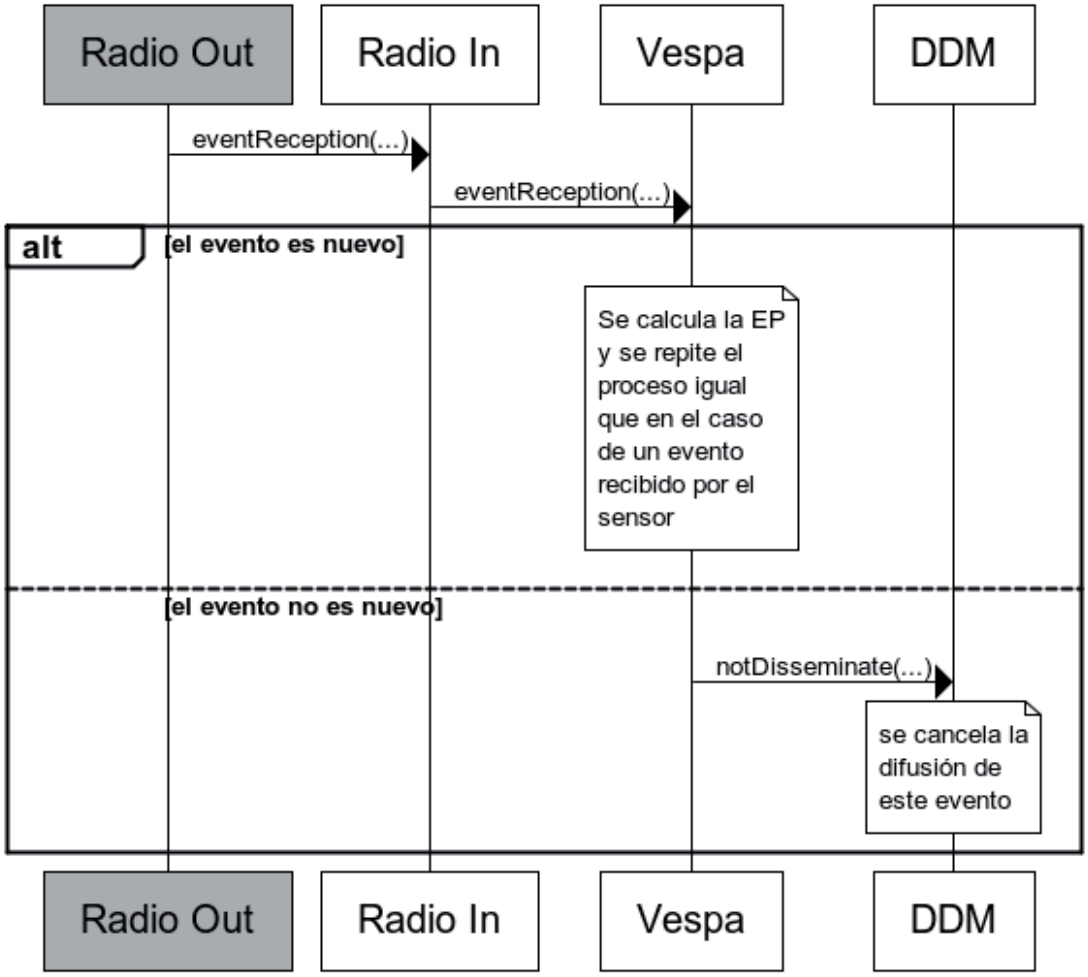


Figura D.6: Hilo de ejecución de la recepción de un evento (el módulo sombreado es del vehículo emisor)

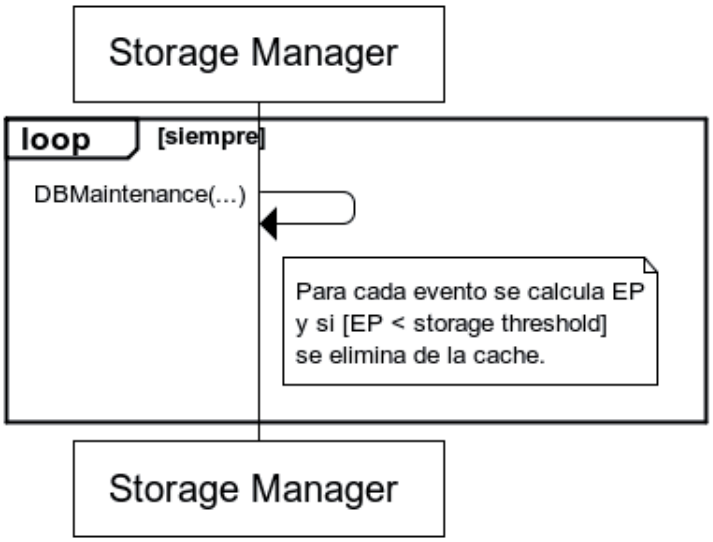


Figura D.7: Hilo de ejecución del gestor de almacenamiento

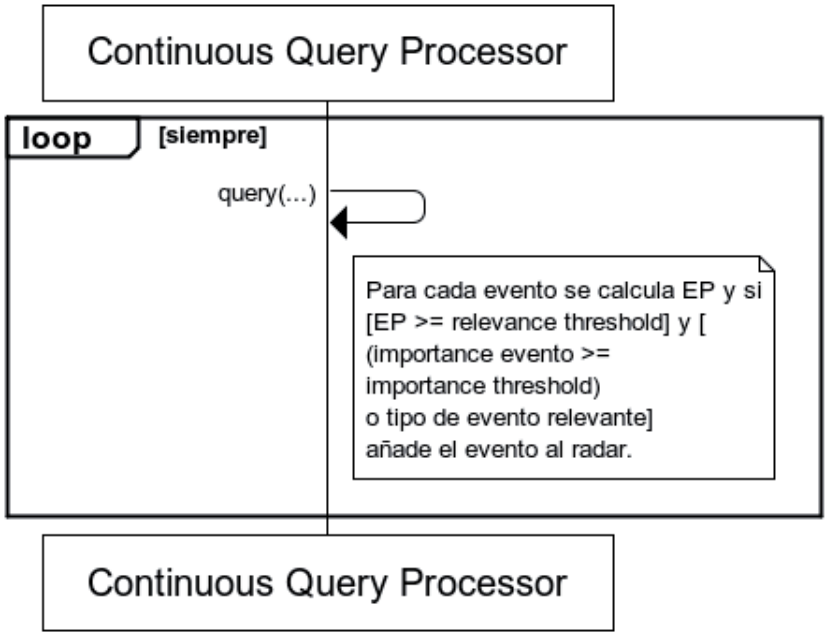


Figura D.8: Hilo de ejecución del procesador de consultas continuas

String	ukey	clave única, generada concatenando el identificador único del vehículo con el identificador localmente único del evento.
int	version	para distinguir entre diferentes actualizaciones del mismo evento.
float	importance	ayuda a determinar cuándo se debe informar de la información al conductor. Contiene un valor entre 0 y 1.
Position4D	currentPosition	tiempo y posición correspondiente a la generación del evento.
Position4D	directionRefPosition	tiempo y posición de una referencia anterior para permitir a cada vehículo evaluar la dirección del evento, lo cual es necesario para estimar su relevancia. Contiene el valor *NULO* si se trata de un evento no dependiente de la dirección.
Position4D	mobilityRefPosition	tiempo y posición de una referencia anterior para permitir a cada vehículo evaluar la movilidad del evento, lo cual es necesario para estimar su relevancia. Contiene el mismo valor que *currentPosition* en el caso de que sea un evento estacionario.
Position4D	lastDiffuserPosition	posición del último vehículo que retransmitió el mensaje. Es usado por el protocolo de diseminación.
int	hopNumber	indica el número de redifusiones del mensaje.
String	description	describe de forma precisa el evento representado.
String	type	añadido de mi implementación para agilizar ciertos cálculos. Puede tener los siguientes valores: *{DirectedMobile, DirectedNotMobile, NotDirectedMobile, NotDirectedNotMobile}*.
int	actorId	añadido de mi implementación, necesario para saber a qué plaza de aparcamiento hace referencia un evento de aparcamiento libre, ya que si la veo ocupado tengo que hacer caso omiso de dicha plaza hasta que me llegue en otro evento.

Tabla D.1: Estructura de evento VESPA implementada

A continuación se detallan los algoritmos más importantes de la implementación.

Algoritmo D.1: Detección de un evento

```
### Hilo de ejecución del módulo Sensor ###

void Sensor.run()
{
  while (continueRunning)
    Sensor.checkForEvents();
    sleep
}

void Sensor.checkForEvents()
{
  for all flags
    if (distance(flag,yo)<= SIGHT_RANGE)
      vespa.generateEvent("Flag",flag.id);
}

void Vespa.generateEvent(String description, int id)
{
  currentPosition = obtener ultimo registro del gps

  if (evento es dirigido)
    directionReferencePosition = obtener vector direccion del gps
  else
    directionReferencePosition = null;

  if (evento es movil)
    mobilityReferencePosition = obtener vector movilidad del gps
  else
    mobilityReferencePosition = currentPosition;

  if (listOfEventsById.containsKey(id)) //será una actualización
    oldEventData = listOfEventsById.get(id);
    event = crear nuevo evento VESPA con version = oldEventData.version + 1;
  else //será un nuevo evento
    event = crear nuevo evento VESPA

  put event on listOfEventsById list //para diferenciar nuevos eventos de actualizaciones
  put event on listOfEventsByKey list //para no procesar retransmisiones de tus eventos
  Vespa.DMProcess(event);
```

```
}

void Vespa.DMProcess(VespaEvent event)
{
  EP = calculo de la EP
  if (EP >= umbral de difusion)
    DataDisseminationManager.disseminate(event);

  if (EP >= umbral de almacenamiento)
    StorageManager.store(event);
}

void DataDisseminationManager.disseminate(VespaEvent event)
{
  distance = distancia entre event.lastDiffuserPosition y mi posicion
  waitingTime = D*(1-(distance/RADIO_RANGE))*1000; //cálculo de T en segundos
  if (waitingTime<0)
    waitingTime = 0;

  añadir evento a la lista listOfEventsPendigToBeSent
  programar un temporizador en waitingTime segundos para ejecutar ↩
    DataDisseminationManager.MyTimerTask.run(event);
}

void DataDisseminationManager.MyTimerTask.run()
{
  if (listOfEventsPendigToBeSent contiene este evento+version)
    DataDisseminationManager.disseminateNow(event);
}

void DataDisseminationManager.disseminateNow(VespaEvent event)
{
  event.hopNumber++
  event.lastDiffuserPosition = mi posicion
  RadioOut.diffusion(event);

  /* Si en D' segundos no hemos recibido el mismo evento, debemos reenviarlo ya que
    significará que nadie lo ha recibido. Así que ponemos de nuevo el evento en la
    lista de eventos pendientes de enviar y programamos un envío. Cuando recibamos
    el evento, eliminaremos el evento de la lista de forma que cuando el temporizador
    finalice ya no estará en la lista de pendientes y no se enviará.*/
  añadir evento a la lista listOfEventsPendigToBeSent
  programar un temporizador en D' segundos para ejecutar ↩
```

```
    DataDisseminationManager.MyTimerTask.run(event);
}

void RadioOut.diffusion(VespaEvent event)
{
  for all {players, redCars, trafficCars, ambulances} diferentes de mi
    if (distancia entre el otro actor y yo es menos que RADIO_RANGE y el otro ↩
      actor tiene VESPA habilitado)
        elOtroActor.RadioIn.eventReception(event);
}

*** En el sistema VESPA del otro vehículo ***

void RadioIn.eventReception(VespaEvent event)
{
  listaDeEventos.addDato(event);
}
```

Algoritmo D.2: Recepción de un evento

```
### Hilo de ejecución del módulo Radio In ###

void RadioIn.run()
{
  while (continueRunning)
  {
    event = listaDeEventos.getDato();
    Vespa.eventReception(event);
  }
}

void Vespa.eventReception(VespaEvent event)
{
  esNuevo = true;
  if (listOfEventsByKey contiene el evento)
    oldEventData = listOfEventsByKey.get(event.ukey);
    if (event.version <= oldEventData.version)
      esNuevo = false;

  if (esNuevo)
    se añade el evento a la lista listOfEventsByKey
    DMProcess(event);
  else
    DataDisseminationManager.notDisseminate(event);
```

```
}

void DataDisseminationManager.notDisseminate(VespaEvent event)
{
  if (listOfEventsPendigToBeSent contiene el evento con la misma version y el ↩
    hopNumber del recibido es mayor que el almacenado)
      eliminamos el evento de la lista listOfEventsPendigToBeSent
}
```

Algoritmo D.3: Hilo de ejecución del gestor de almacenamiento

```
### Hilo de ejecución del módulo Storage Manager ###

void StorageManager.run()
{
  while (continueRunning)
    StorageManager.DBMaintenance();
    sleep
}

void StorageManager.DBMaintenance()
{
  for all eventos almacenados en cache
    EP = calculo de la EP
    if (EP < umbral de almacenamiento)
      se elimina el evento de la cache
}
```

Algoritmo D.4: Hilo de ejecución del procesador de consultas continuas

```
### Hilo de ejecución del módulo Continuous Query Processor ###

void ContinuousQueryProcessor.run()
{
  while (continueRunning)
    ContinuousQueryProcessor.query(game current sequence);
    sleep
}

void ContinuousQueryProcessor.query(int currentSequence)
{
  se vacia la lista listOfRadarPoints del vehiculo
  for all eventos de la cache
    EP = calculo de la EP
```

```
    if ((EP >= umbral de relevancia) && (event.importance >= umbral de importancia
      || listOfRelevantTypes contiene event.description)
        añadimos la representacion del evento a la lista listOfRadarPoints del vehiculo
}
```

El cálculo de la probabilidad de encuentro (EP) ha sido desarrollado adaptando (y arreglando, ya que la versión usada no era la final y contenía errores) el algoritmo del simulador, que aplica la siguiente fórmula [6]:

$$EP = \frac{100}{\alpha \times \Delta d + \beta \times \Delta t + \gamma \times \Delta g + \zeta \times c + 1} \tag{D.1}$$

siendo Δd la mínima distancia al evento a lo largo del tiempo, Δt el tiempo hasta lograr el mayor acercamiento al evento, Δg la diferencia entre el tiempo en que se ha generado el evento y el momento en que el vehículo estará más cercano al evento, c el ángulo entre el vehículo y el evento y α, β, γ y ζ coeficientes de penalización.

D.1.4. Protocolo de reserva

El funcionamiento teórico del protocolo de reserva es el siguiente [5]: El vehículo que abandona la plaza de aparcamiento se convierte en el coordinador de la misma. Éste envía un mensaje para informar a todos los vehículos a su alcance que dicha plaza está disponible y permanece un tiempo T a la escucha de posibles respuestas.
Cada vehículo que esté interesado le responderá al coordinador aportando su identificador y la información necesaria para que el coordinador pueda elegir a qué vehículo asignar la plaza.
Cuando el coordinador elija a un vehículo, le enviará un mensaje notificándoselo, debiendo éste responder al coordinador confirmando la recepción del mensaje y que tomará la plaza de aparcamiento.
Si el coordinador no ha sido capaz de encontrar un vehículo interesado, se cambia de coordinador. Este proceso lo inicia el actual coordinador, que enviará un mensaje a los vehículos cercanos. Los vehículos que reciban el mensaje (y no sean ya coordinadores de otra plaza) responderán indicando su estimación de cuántos vehículos cercanos buscan aparcamiento. El coordinador ordenará las respuestas según las estimaciones y contactará con los vehículos de la lista en orden hasta que uno confirme la recepción y pase a ser el nuevo coordinador. En el caso de no poder realizar este cambio de coordinador, el actual coordinador mantendrá este rol y después de un periodo de tiempo difundirá de nuevo el mensaje sobre la plaza disponible.

En la implementación que desarrollada se ha simplificado un poco este procedimiento, pero manteniendo las ideas fundamentales para que el resultado sea, si bien no idéntico sí muy similar.
Las diferencias radican en que, como gracias a las estructuras del mundo de juego, el vehículo coordinador puede conocer qué vehículos a su alcance están interesados sin necesidad del intercambio de mensajes, éste proceso se realiza de esta forma «simulada», enviándose únicamente el mensaje final, que en este caso contiene la situación de la plaza de aparcamiento libre.
El cambio de coordinador también se realiza sin envío de mensajes y además se ha modificado el parámetro de ordenación, que ya no es el número de vehículos cercanos buscando aparcamiento sino que ahora se realiza una ordenación según la lejanía al coordinador actual.

Algoritmo D.5: Métodos del protocolo de reserva

```
enviarCoordinar(datos parking)
{
  enviarEvento(datos parking)
  si devuelve falso:
    cambiarCoordinador(datos parking)
    si devuelve falso:
      stablecer un temporizador y cuando finalice se ejecutará esta misma función.
}

booleano enviarEvento(datos parking)
{
  Cuenta VESPAs (en alcance & buscando parking) (*)
  Si 0:
    devuelve falso
  si 1:
    Le envía evento Parking (por VESPA)
    devuelve cierto
  si 2+:
    Busca al que lleve más tiempo buscando/más cercano/mayor EP (*)
    Le envía evento Parking (por VESPA)
    devuelve cierto
}

booleano cambiarCoordinador(datos parking)
{
  Busca el VESPA (en alcance) más lejano (*)
  si existe:
    establecer un temporizador en ese coche y cuando finalice se ejecutará ↩
```

```
      enviarCoordinar(datos parking) (*)
    devuelve cierto
  sino:
    devuelve falso
}
```

Nota importante: cuando pone **(*)** significa que es una función «trucada» que se hace con datos que no conoce VESPA sino que se obtienen directamente del juego.

Para determinar a qué vehículo asignar la plaza disponible de entre todos los candidatos, se han elaborado tres estrategias diferentes: aquél que lleva más tiempo buscando una plaza, aquél más cercano o aquel con una mayor *EP*.
Ésta elección de estrategia viene determinada por una variable del fichero de texto de configuración (*ParamConfig.txt*) y se aplica a todos los vehículos con VESPA.

D.1.5. Necesidades de la implementación

En el desarrollo de la implementación de VESPA para `Vanet-X` han aparecido diversos problemas debido a la necesidad de poder generar eventos «por observación» (el conductor será el que los cree al ver un cierto elemento) en lugar de por generación propia que sería lo habitual (p.ej. eventos de servicios de emergencia, accidentes, obstáculos, etc.).
Esta necesidad de generar eventos en base a observaciones viene dada por los eventos que informan de la presencia de una bandera objetivo y por los que avisan de un vehículo enemigo.
Respecto a los eventos de vehículos enemigos otra opción sería asumir que el vehículo enemigo es robado y tiene el sistema VESPA instalado y activo debido al sistema antirrobo. En la implementación realizada se han tenido en cuenta las dos posibilidades y la elección del método a utilizar es una variable configurable en el menú de opciones de VESPA.

Para permitir la generación de eventos mediante observación se ha incorporado un método de calcular los vectores de movilidad y dirección de forma externa y también un método para diferenciar si el evento que se ha de crear es un evento inédito o es una actualización de otro previamente creado (por el mismo sistema).

El método para distinguir entre nuevos eventos y actualizaciones consiste en que cada vez que se genere un nuevo evento hay que almacenar sus datos en una estructura indexada por el identificador del objeto al que representa (una bandera, otro vehículo o a sí mismo). De esta forma, si se trata de un evento «de posición» («jugador», «servicio de emergencia» o «enemigo», ya que son los únicos que pueden actualizar su posición), antes de la generación del evento se ha de revisar esta estructura para comprobar si ya existe una entrada para el objeto en cuestión,

en cuyo caso deberemos crear una actualización.

El método para calcular el vector de dirección (mostrado en el Algoritmo D.6) consiste en acceder a los datos guardados de la última actualización del evento (almacenado en la estructura indexada creada como solución del problema anterior) y usar estos datos para calcular el valor de *directionReferencePosition*. Esto se realiza calculando la distancia entre el valor almacenado de dicha variable y la posición actual, de forma que si es menor que una distancia determinada se devuelve dicho valor antiguo pero si es mayor se devuelve la posición actual.
De esta forma se consigue que se vaya actualizando el vector cada cierta distancia.
El método para el cálculo del vector de movilidad es análogo a éste.

Algoritmo D.6: Obtención del vector dirección

```
Position Vespa.getMyDirectionVector(Position currentPosition,
                                    Position directionReferencePosition)
{
  if (directionReferencePosition == null ||
    distancia(currentPosition,directionReferencePosition)
    >= TRAVELHISTORYDIRECTIONMINDIST)
    return currentPosition;
  else
    return directionReferencePosition; //el valor antiguo
}
```

D.1.6. Atascos (elaborados para el aprovechamiento de VESPA)

Se ha incluido este apartado dentro de la sección dedicada al sistema VESPA ya que la inclusión de los atascos en el mundo de juego está completamente ligada a este sistema.

Se ha realizado una implementación realista desde el punto de vista del jugador pero que no lo es tanto desde el punto de vista de los vehículos del tráfico controlados por el computador, ya que tienen acceso a una lista común de segmentos de calle en situación de atasco en lugar de tener listas separadas conforme reciben los eventos.

A continuación se explican las ideas generales seguidas en la implementación realizada.
La notificación de un atasco se produce cuando un vehículo del tráfico lleva más de 10 segundos sin moverse (sin estar aparcado), por lo que se presupone que puede formar un embotellamiento y se avisa de ello.

Dicho aviso se realiza mediante la creación y emisión de un evento VESPA, de tipo «accidente», que se mostrará en el radar de los vehículos de los jugadores.
Los vehículos del tráfico evitan los atascos al calcular su ruta mediante el algoritmo de *path-finding*. Para realizar esto los datos del evento VESPA no son suficientes sino que se necesita el segmento exacto donde tiene lugar el atasco. Por este motivo se utiliza una lista global donde se almacenan los atascos que existen actualmente, de donde los vehículos del tráfico pueden obtener los datos necesarios para evitar dicho atasco correctamente. A pesar de que en ese aspecto no dependen de los eventos recibidos por VESPA, en realidad sí que los dependen ya que es solo cuando reciben un nuevo evento de atasco cuando recalculan la ruta actual.

Con esta implementación relativamente sencilla y parcialmente realista se logra incluir los atascos en el mundo de juego de forma que pueden ser utilizados para mostrar las ventajas que supone contar con un sistema como VESPA.

En la Figura D.9 se observa la ejecución de una versión de desarrollo de `Vanet-X` en la que en el radar se muestran en rojo los segmentos de calles notificadas como atasco.

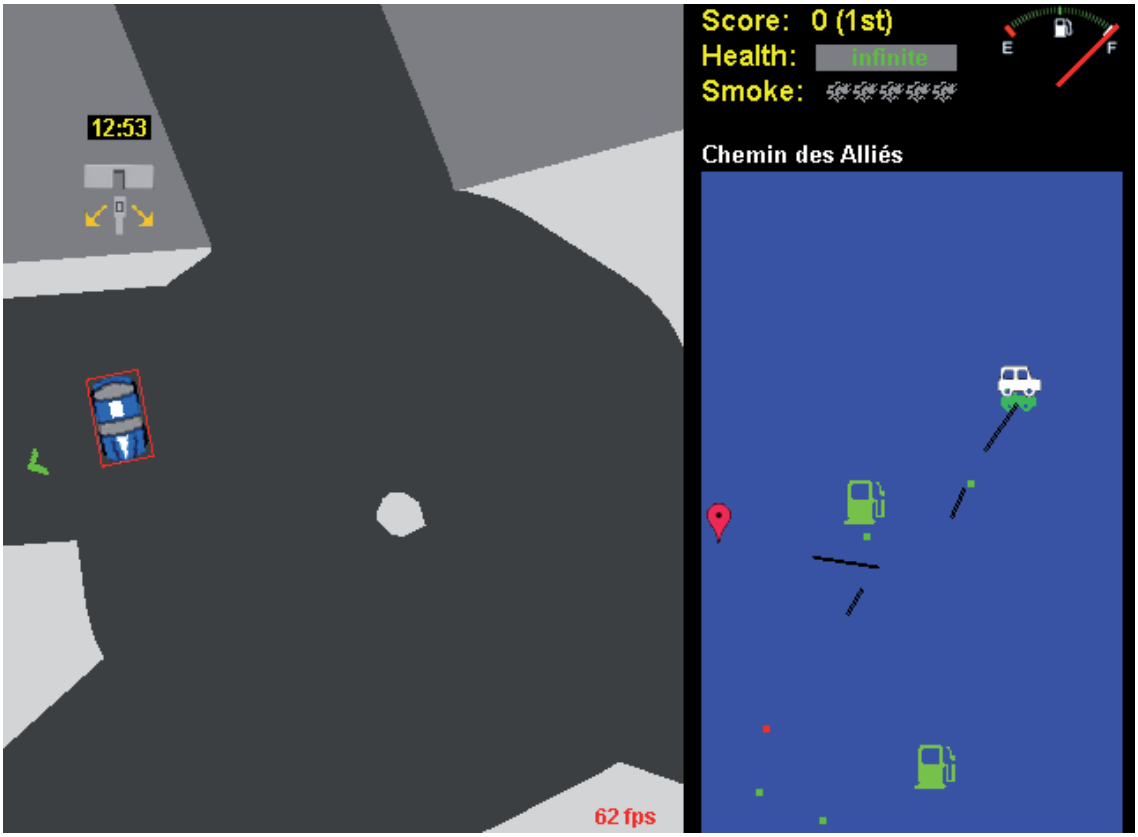


Figura D.9: Atascos representados en el radar (en negro)

D.2. Añadidos para la explotación

En esta sección se explicará en detalle diferentes aspectos relacionados con la explotación del juego como método de evaluación de estrategias de gestión de información. Estos aspectos son: el servidor dedicado, el servidor de recogida de estadísticas y el sistema de generación de estadísticas.

D.2.1. Servidor dedicado

Como en la mayoría de los juegos en línea, en `Vanet-X` se ha implementado la idea de contar con un servidor dedicado que pueda dejarse en permanente funcionamiento en un computador (generalmente con mejor potencia y ancho de banda de lo habitual en un ordenador doméstico), de forma que los jugadores puedan unirse a dicho servidor.

El funcionamiento es el siguiente: existe un proceso (*Master server*) que está siempre en funcionamiento y que acepta las peticiones de conexión de los clientes. Con cada petición, este proceso comprueba si ya esta creada una instancia del *lado servidor* de `Vanet-X`, creándola si no existía. Posteriormente devuelve al cliente el puerto a través del cual se podrá unir a la partida.

Dicha instancia del servidor finalizará cuando se acabe la partida (por lograr o fallar los objetivos) o se hayan desconectado todos los jugadores. De esta forma no hace falta tener permanentemente una partida en marcha con el consumo de CPU que conlleva sino que se iniciará y finalizará cuando sea necesaria.

Además, el proceso *Master server* está diseñado a prueba de fallos[1] de forma que aunque tenga lugar un error crítico en el servidor, el proceso se reinicie automáticamente y vuelva a estar a la espera de nuevas peticiones.

Al ejecutar el servidor dedicado, se requerirá que el usuario seleccione la carpeta de juego, donde se almacenarán las estadísticas y de donde se obtendrán los escenarios disponibles y la configuración y parámetros que se usarán al crear el servidor. Posteriormente, también se requererirá que se seleccione cual será el escenario que se usará para la partida.

Salvo la situación del directorio de juego, que es fija para toda la ejecución del servidor dedicado, los demás parámetros pueden ser modificados mediante dos métodos:

1. El primer método para cambiar esta configuración y parámetros es obtener acceso al computador en el que funciona el servidor dedicado y sustituir los

[1] Salvo fallos de la Máquina Virtual Java

archivos correspondientes del directorio de juego.
El fichero *paramConfig.txt* puede ser editado con un editor de texto, pero para realizar cambios en la configuración (fichero *config*) el método consiste en iniciar un servidor normal (como si fueras a jugar), en los menús realizar los cambios deseados, dándole siempre a guardar cambios, y -muy importante si se cambia el modo de juego- iniciar la partida. Una vez iniciada se puede cerrar la aplicación cuando se desee, pero este paso es básico ya que los cambios realizados en el modo de juego solo se guardan al darle al botón de iniciar la partida.

2. Para las situaciones en las que no se puede tener acceso directo al directorio de juego del servidor dedicado, se ha creado un *terminal* que se usa para comunicarse mediante comandos con el servidor dedicado mediante el protocolo TCP/IP y que tiene las siguientes opciones:

   - cambiar el escenario que se usará al crear el servidor (no se cambia el escenario de la partida en curso): el terminal recibe el listado de los escenarios disponibles y envía tu elección al servidor dedicado.
   - modificar la configuración del juego (fichero *config*): se pide al usuario que introduzca uno a uno los nuevos valores deseados, mostrándole previamente los valores actuales.
   - modificar la configuración de VESPA (fichero *config*):: de igual forma que el anterior.
   - modificar los parámetros (fichero *paramConfig.txt*):: de igual forma que el anterior.
   - recuperar los ficheros de estadísticas generados, decidiendo el usuario si desea conservar los ficheros en el servidor o eliminarlos.

   Para poder realizar estas acciones es necesario que el usuario del terminal introduzca una contraseña establecida al crear el servidor dedicado.

El servidor dedicado está implementado de forma conjunta con el resto del juego, siendo necesario ejecutar el juego con los parámetros `-dedicated puerto_maestro puerto_partida contraseña`, donde el primer argumento indica en que puerto estará a la escucha de nuevas conexiones de los clientes (este es el puerto que deben conocer los clientes), el segundo indica el puerto en el que se creará la partida y el tercero la contraseña que se pedirá cuando se intente acceder desde el *terminal*.

Asimismo, el terminal, que también esta implementado de forma conjunta, puede ser accedido ejecutando el juego con los parámetros `-terminal IP puerto`, debiendo coincidir los dos argumentos con la dirección y puerto en los que funciona el proceso servidor dedicado. La contraseña de acceso es requerida posteriormente.

A continuación se muestran la estructura del código usado para el servidor dedicado (Algoritmo D.7) y las interacciones esquematizadas entre el cliente, el servidor y el servidor dedicado (figuras D.10, D.11 y D.12).

D.2.2. Servidor de recogida de estadísticas

Uno de los objetivos de este Proyecto Fin de Carrera es recoger estadísticas acerca del uso y funcionamiento del sistema VESPA. Por ello, hay que desarrollar un método por el cual se puedan obtener dichas estadísticas aún cuando la partida no se desarrolle en los servidores dedicados sino en partidas creadas por los usuarios.

La forma de realizar esto es que el servidor, al finalizar la partida, después de generar los ficheros de estadísticas los envíe a un servidor externo que se encargará de recogerlos y almacenarlos. A dicho servidor externo se le llamará *Statistics server*

La estructura del código del servidor de recogida de estadísticas se muestra en Algoritmo D.8 y Algoritmo D.9.

Cada servidor tiene definido un puerto y una dirección DNS dinámica, modificable en *ParamConfig.txt*, al que se conectará para enviar los ficheros de estadísticas después de crearlos.
El *Statistics server*, cada vez que se inicie, actualizará la dirección IP a la que apunta dicha dirección DNS dinámica, haciendo uso de de la API asociada al servicio.

El almacenamiento de los ficheros en cada servidor local se realiza almacenando todos los ficheros creados durante la partida en un directorio cuyo nombre es la fecha y la hora de inicio de dicha partida en formato «*aaaa_mm_dd hh_mm_ss*» (ver sección D.2.3 para más detalles). Sin embargo, esta solución no es válida para el *Statistics server* ya que puede recibir los datos de varias partidas con fecha y hora coincidentes.
Por esta razón se ha decidido que cada cliente envíe al *Statistics server* un identificador único (haciendo uso de la clase *java.util.UUID*) que se tome como nombre del directorio. Este identificador cambia cada vez que se inicia una nueva partida y no coincidirá con los de otros clientes por lo que es una elección óptima para asegurar que cada directorio contenga únicamente los ficheros enviados por el cliente que le corresponde.

Al igual que el servidor dedicado, el servidor de recogida de estadísticas está implementado de forma conjunta con el resto del juego, siendo necesario para su uso ejecutar el juego con el parámetro `-statserver`.

Algoritmo D.7: Estructura del servidor dedicado

```
inicializar directorio base
elegir escenario
cargar parámetros y configuración
while (continuar)
{
  inicializar servidor TCP
  while (continuar)
  {
    aceptar cliente
    if (peticion terminal)
    {
      Ejecución paralela (en nuevo hilo):
      {
        recibir contraseña
        if (contraseña correcta)
        {
          recibir tipo de petición
          procesar petición
        }
      }
    }
    else
    {
      if (no hay partida creada)
      {
        crear una partida nueva
      }
      enviar puerto del juego a cliente
    }
  }
  cerrar servidor TCP
}
```

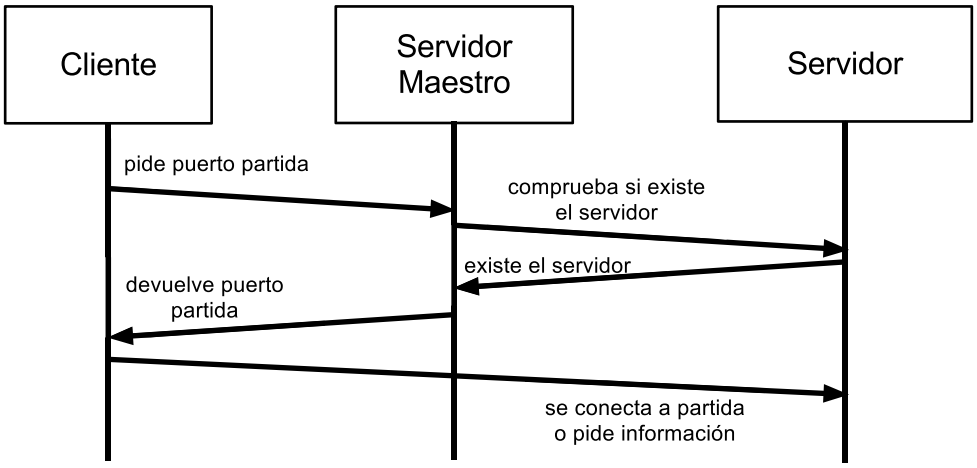


Figura D.10: Esquema conexión con servidor dedicado (hay una partida en curso)

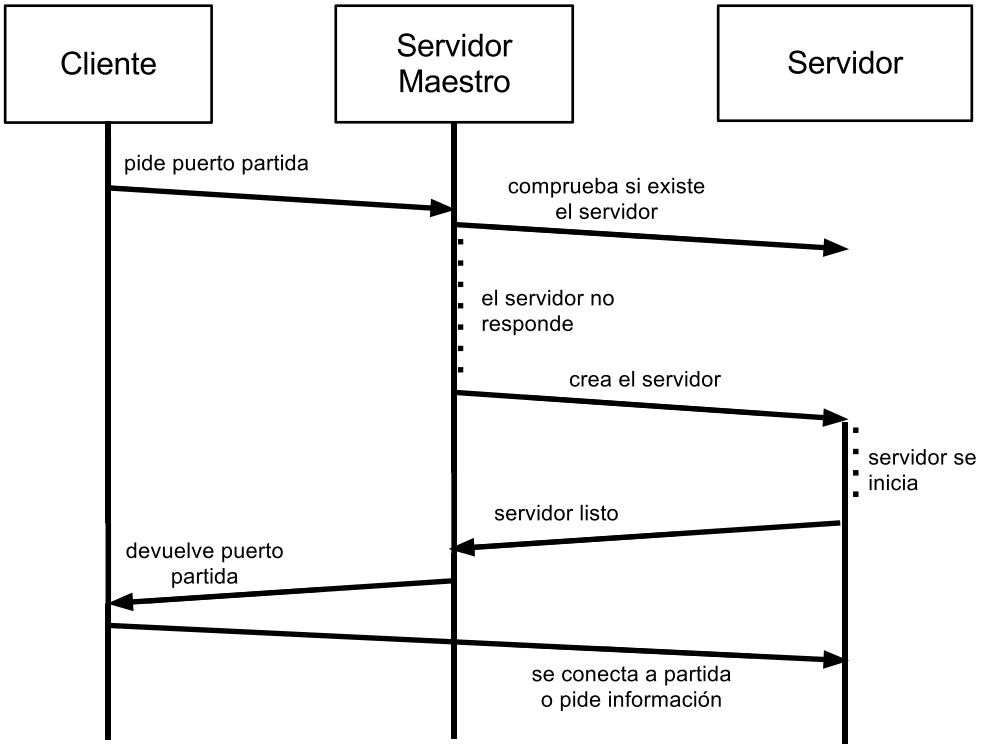


Figura D.11: Esquema conexión con servidor dedicado (no hay ninguna partida en curso)

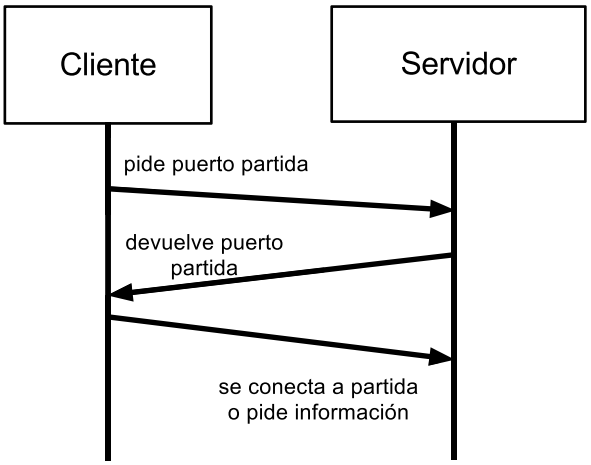


Figura D.12: Esquema conexión sin servidor dedicado

Algoritmo D.8: Estructura del servidor de recogida de estadísticas

```
Crear carpeta destino
while (continuar)
{
  inicializar servidor TCP
  while (continuar)
  {
    aceptar conexion
    crear hilo para el manejo de esa conexion
  }
  cerrar servidor TCP
}
```

Algoritmo D.9: Estructura del hilo que maneja la conexión (Statistics server)

```
recibe el identificador del cliente
recibe el tipo de estadistica que se enviará
recibe la estadistica del tipo especificado
escribe la estadistica en fichero
cierra conexión
```

D.2.3. Estadísticas

Uno de los objetivos de este Proyecto Fin de Carrera es poder recoger estadísticas de VESPA y otros aspectos para poder analizarlos posteriormente. Para ello, se obtienen diferentes tipos de estadísticas: del juego, de VESPA y de las tareas de aparcamiento. Todas estas estadísticas se generan en diversos ficheros de texto, almacenados en una misma carpeta con un nombre único e identificativo, junto a los cuales también se genera un fichero de texto adicional que incluye los parámetros de configuración actuales, para así poder replicar la prueba en un futuro.

La estructura de almacenamiento de dichos ficheros se puede observar en la Figura D.13.

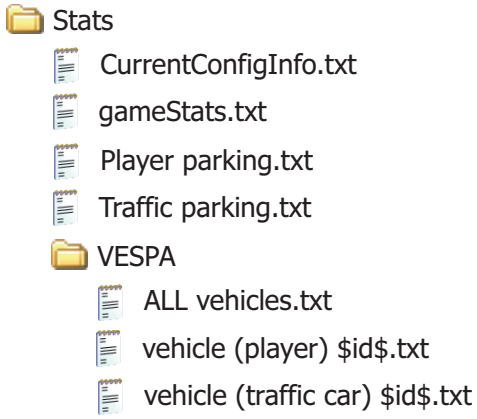


Figura D.13: Estructura del directorio de estadísticas

A continuación se detalla el contenido de cada tipo de fichero de estadísticas creado:

«*CurrentConfigInfo.txt*»: contiene todos los valores de los parámetros (fichero «*ParamConfig.txt*») y de la configuración (fichero «*config*») usados, así como también el tiempo de ejecución de la partida. Ver Figura D.14.

«*gameStats.txt*»: contiene las estadísticas propias del juego, indicando para cada jugador: puntuación, equipo al que pertenece, uso del sistema de compartición de datos (VESPA) y nivel de habilidad del jugador. Ver Figura D.15.

«*Player parking.txt*»: precedido por una leyenda, contiene las estadísticas de los aparcamientos logrados por los vehículos controlados por los jugadores, a razón de un aparcamiento por línea e incluyendo para cada aparcamiento: identificador del vehículo, tiempo que ha costado aparcar según el reloj del sistema (valor únicamente orientativo, ya que no es fiable en ordenadores poco potentes en los que el juego funcione con cambios bruscos de fotogramas

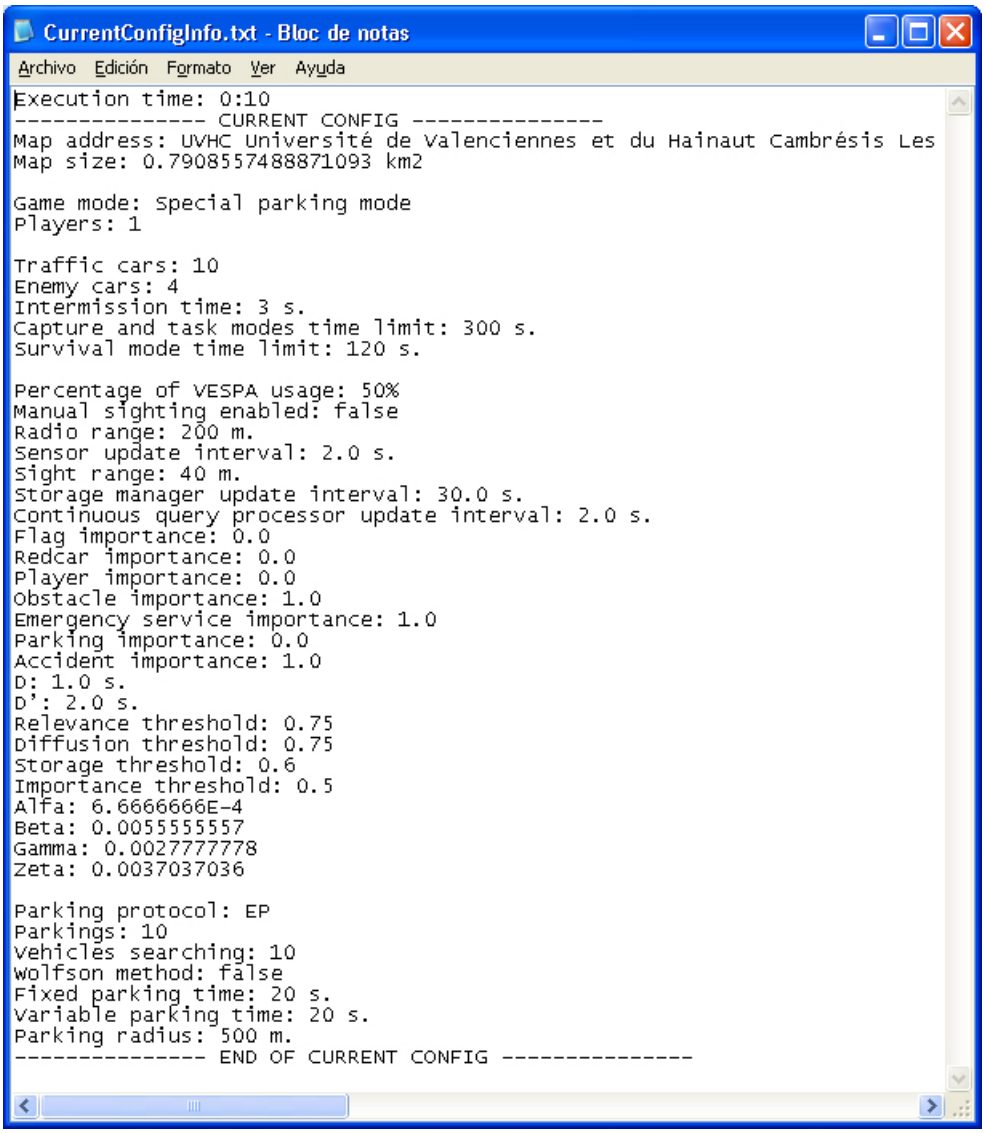


Figura D.14: Ejemplo de contenido del fichero «*CurrentConfigInfo.txt*»

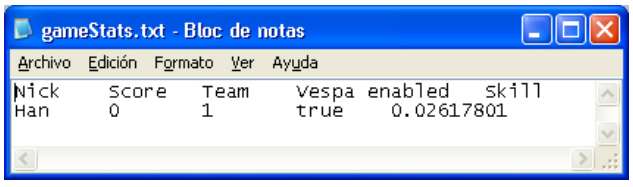


Figura D.15: Ejemplo de contenido del fichero «*gameStats.txt*»

por segundo), ciclos de juego que ha costado aparcar, indicador de uso del sistema de compartición de datos (VESPA), indicador de si el aparcamiento se ha realizado en una plaza indicada por el protocolo de aparcamiento de VESPA y, por último, protocolo de aparcamiento usado (Ninguno, según *Encounter Probability* (*EP*), según tiempo de búsqueda o según distancia). Ver Figura D.16.

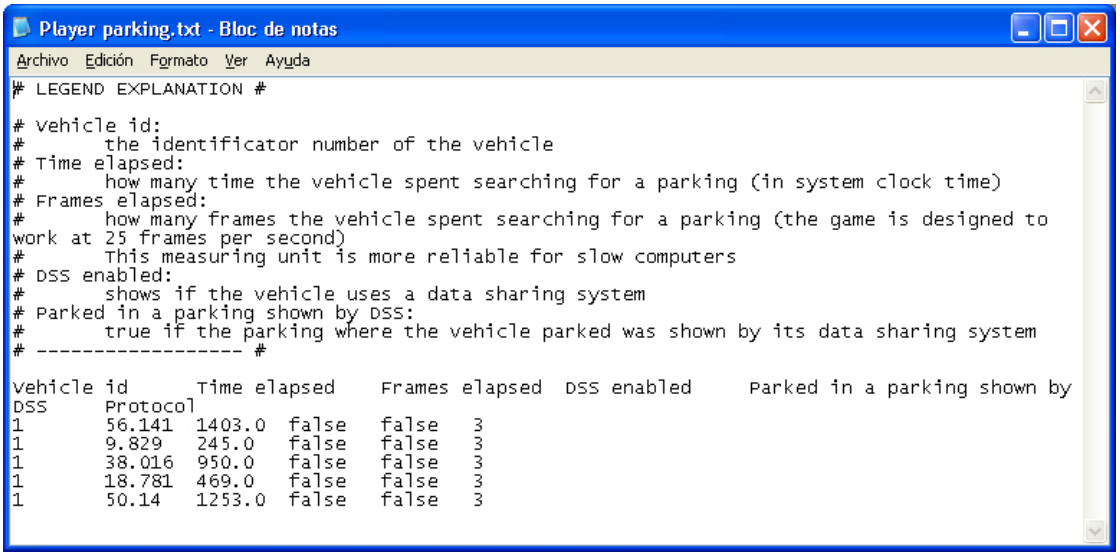


Figura D.16: Ejemplo de contenido del fichero «*Player parking.txt*»

«***Traffic parking.txt***»**:** idéntico al anterior pero conteniendo los aparcamientos logrados por los vehículos controlados por el computador. Ver Figura D.17.

Estadísticas VESPA: respecto a las estadísticas relacionadas con VESPA, se genera un fichero por cada vehículo dotado de dicho sistema y un fichero adicional que recoge la suma de todos los datos de todos los anteriores ficheros.
Los ficheros tienen la siguiente estructura (ver Figura D.18):

- Número de eventos detectados por el vehículo.
- Número de eventos creados que sean nuevos respecto del total.
- Número de eventos creados que sean actualización de eventos pasados respecto del total.
- Total de eventos procesados.
- Del total de eventos procesados, cantidad y porcentaje de eventos enviados al módulo de diseminación.

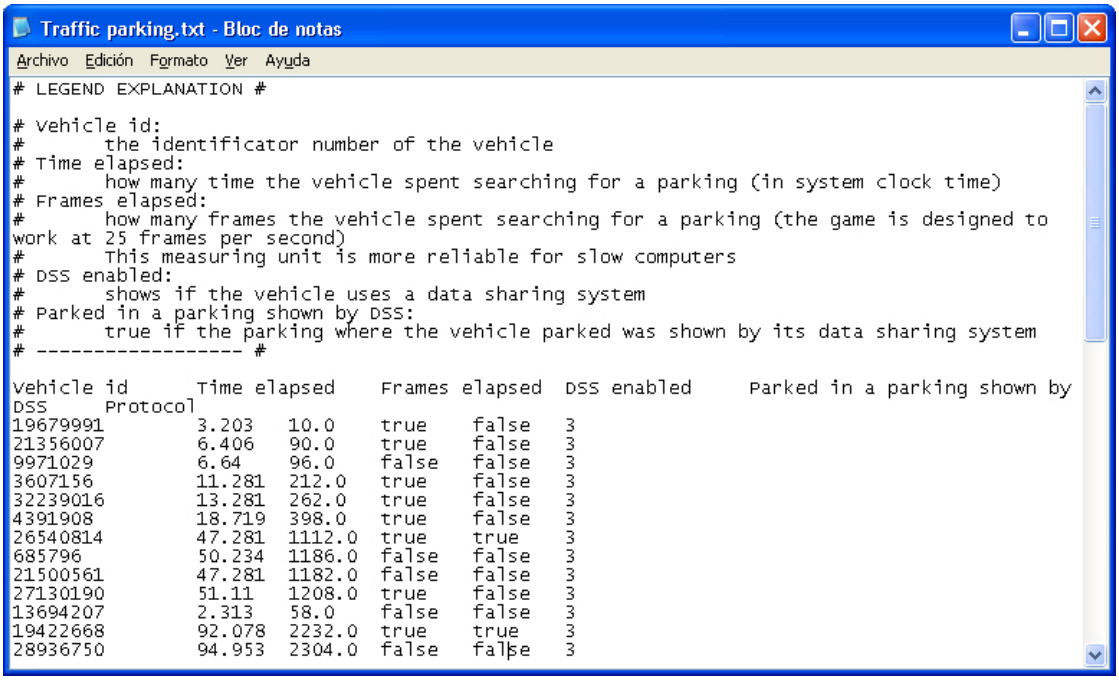


Figura D.17: Ejemplo de contenido del fichero «*Traffic parking.txt*»

- Del total de eventos procesados, cantidad y porcentaje de eventos que se han almacenado.
- Número de eventos recibidos que sean nuevos respecto del total.
- Número de eventos recibidos que ya se hubieran recibido con antelación respecto del total.
- Tiempo medio de espera antes de la redifusión de un evento.
- Número de redifusiones.
- Número de cancelaciones de redifusión (por haber recibido el mismo evento de vuelta)
- Número de eventos eliminados de la caché de almacenamiento.
- Número de eventos eliminados de la caché en cada operación de mantenimiento.
- Número de operaciones de mantenimiento de la caché efectuadas.
- Número de eventos mostrados al conductor.
- Número de eventos mostrados al conductor en cada operación *query*.
- Número de operaciones *query* efectuadas.
- Número máximo de eventos almacenados en la caché.

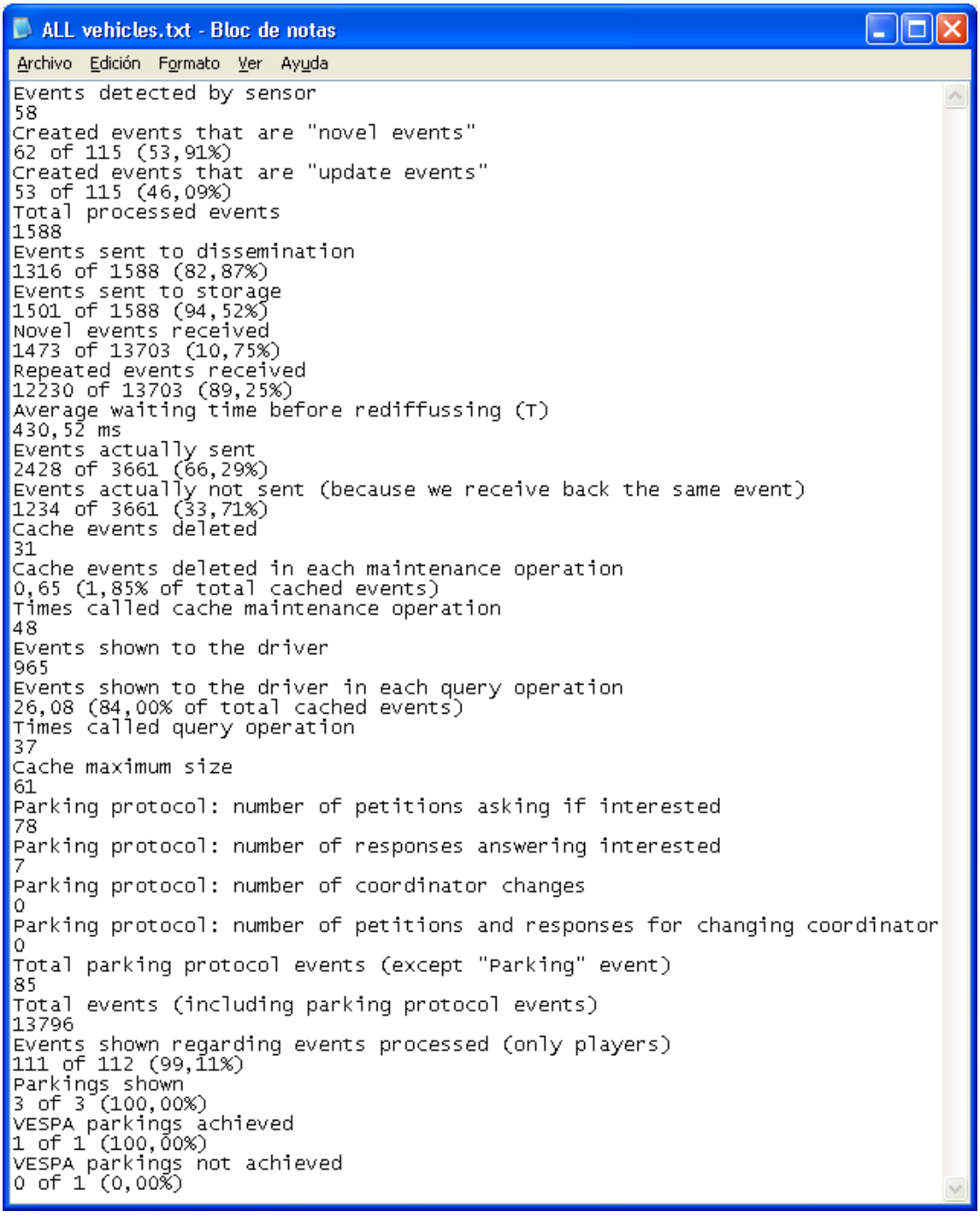


Figura D.18: Ejemplo de contenido del fichero «*ALL vehicles.txt*»

- Número de peticiones del protocolo de aparcamiento preguntando por vehículos interesados en aparcar.
- Número de respuestas del protocolo de aparcamiento indicando el interés del vehículo en aparcar.
- Número de cambios de coordinador del protocolo de aparcamiento.
- Número de peticiones y respuestas del protocolo de aparcamiento en la búsqueda de nuevo coordinador.
- Total de eventos del protocolo de aparcamiento exceptuando los eventos *Parking*.
- Número total de eventos incluyendo también los del protocolo de aparcamiento.
- Número de eventos considerados relevantes para el conductor respecto a los procesados.
- Número de eventos de aparcamiento mostrados respecto del total de eventos creados.
- Número de plazas de aparcamiento indicadas por VESPA logradas.
- Número de plazas de aparcamiento indicadas por VESPA no logradas.

D.3. Rendimiento del juego

El rendimiento del juego depende de los siguientes aspectos: número de jugadores, número de vehículos del tráfico y vehículos enemigos, modo de juego, número de nodos del escenario escogido y uso del sistema VESPA. Todos estos aspectos se pueden modificar en la configuración de la partida, de forma que los usuarios con ordenadores menos potentes pueden variar la configuración para conseguir un buen rendimiento. Además, en el fichero de configuración *ParamConfig.txt* se han habilitado dos parámetros para desactivar el fondo dinámico de los menús y para deshabilitar el uso de transparencias en el juego: «Low CPU usage menu» y «Disable in-game transparencies», lo cual incrementa notablemente el rendimiento en ordenadores poco potentes.

En las figuras D.19 y D.20 se puede observar el uso de *CPU* y de memoria utilizado para algunas de las configuraciones mencionadas en el capítulo 3.8.

Al tratarse de un juego en red es importante conseguir un tamaño reducido de los paquetes de red enviados. En la tabla D.2 se observa las cantidades mínima y máxima de datos requeridos por cada tipo de elemento enviado en red. Estos datos se refieren a los paquetes enviados por el servidor a cada cliente. El tamaño

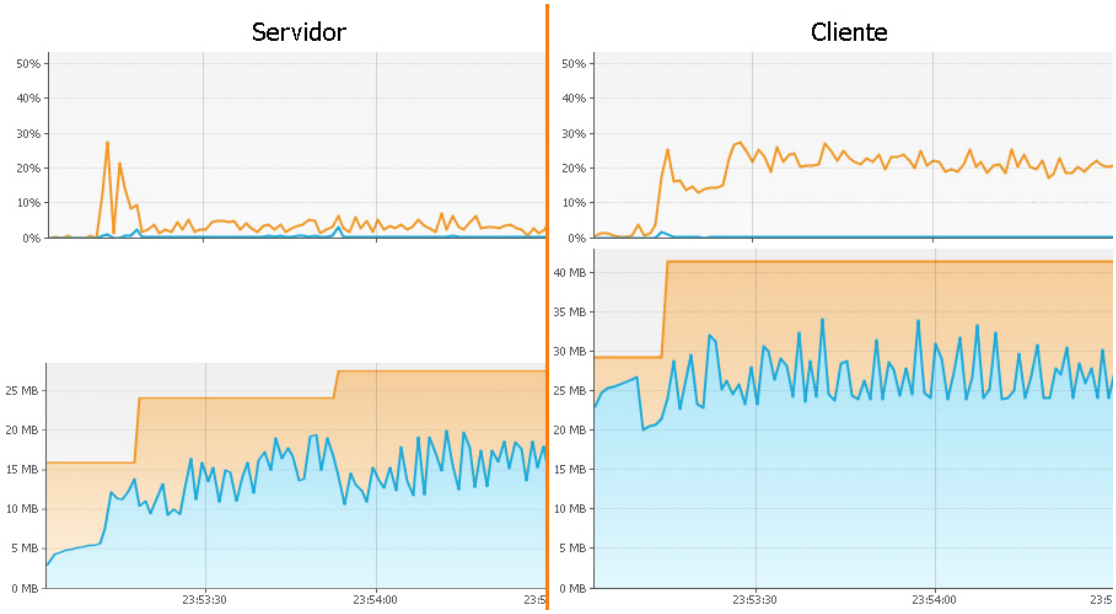


Figura D.19: Rendimiento con servidor dedicado para la configuración: 2 jugadores, 25 tráfico, 4 enemigos, mapa «Trementines». Arriba: uso CPU (naranja), abajo: uso de memoria (azul).

de los paquetes enviados por los clientes al servidor es fijo y es de 58 B/ciclo (1,45 KB/s).

	Mínimo		Máximo	
	B/ciclo	KB/s	B/ciclo	KB/s
Jugador	17	0.42	39	0.97
Tráfico	13	0.32	35	0.87
Enemigo	13	0.32	35	0.87
Ambulancia	12	0.3	34	0.85
Parking	9	0.22	23	0.57
Bandera	8	0.2	22	0.55

Tabla D.2: Tamaño mínimo y máximo de envío en red (servidor → cliente) según tipo de elemento

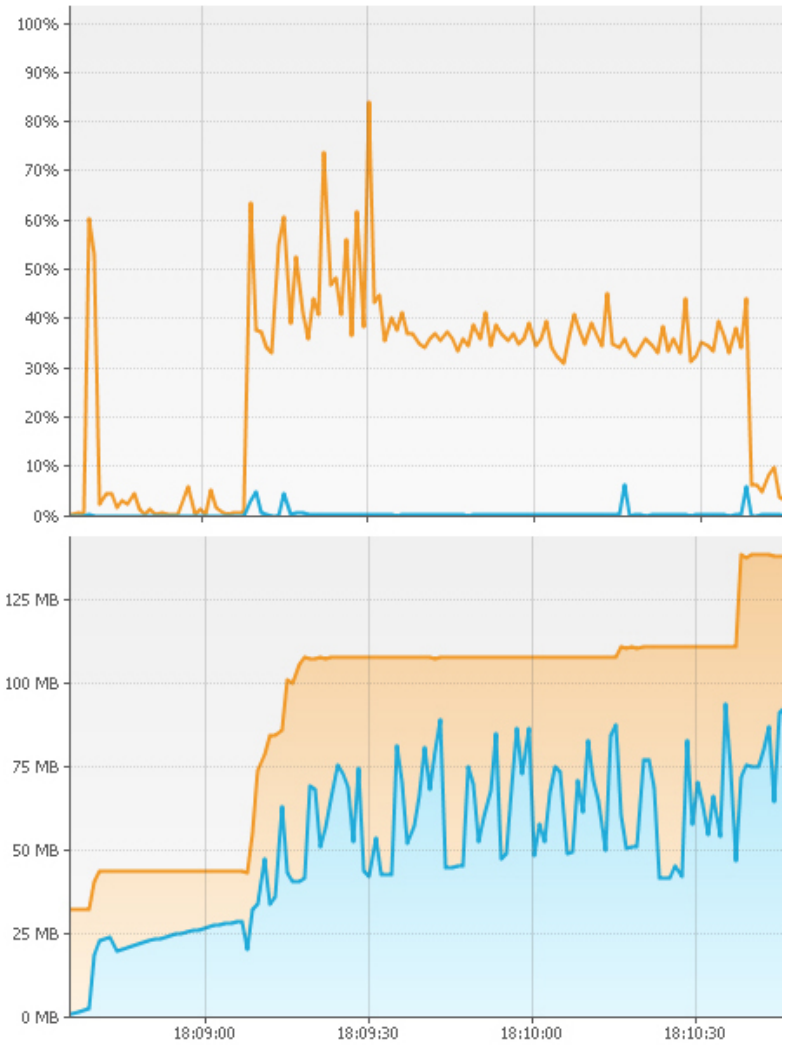


Figura D.20: Rendimiento con servidor no dedicado para la configuración: 1 jugador, 50 tráfico, 8 enemigos, mapa «Trementines». Arriba: uso CPU (naranja), abajo: uso de memoria (azul).

Anexo E

Artículo IMMoA'13

En este anexo se adjunta una copia del artículo presentado para el *workshop* IMMoA'13 (International Workshop on Information Management for Mobile Applications), realizado conjuntamente con Sergio Ilarri y Eduardo Mena.

Vanet-X: A Videogame to Evaluate Information Management in Vehicular Networks

Sergio Ilarri
IIS Department
University of Zaragoza
Zaragoza, Spain
silarri@unizar.es

Eduardo Mena
IIS Department
University of Zaragoza
Zaragoza, Spain
emena@unizar.es

Víctor Rújula
IIS Department
University of Zaragoza
Zaragoza, Spain
han.vikktor@gmail.com

ABSTRACT

Vehicular Ad Hoc Networks (VANETs) are attracting considerable research attention, as they are expected to play a major role for Intelligent Transportation Systems (ITS). Thus, according to a recent survey by ABI Research[1], about 62% of new vehicles will be equipped with vehicle-to-vehicle (V2V) communications by 2027. Vehicular networks offer new opportunities for the development of interesting mobile applications for drivers, but at the same time they also bring challenges from the data management point of view. Thus, for example, techniques should be developed to estimate the relevance of the information exchanged among the vehicles and to propagate the relevant data in the network efficiently and effectively. As testing the proposals in a real large-scale scenario is impractical, simulators are often used.

In this paper we present *Vanet-X*, an online multiplayer driving videogame that we have developed to help in the difficult evaluation task of data management strategies for VANETs. The idea behind the proposal is to exploit the potential of players around the world driving vehicles in the videogame to effortlessly collect data that can be used to extract some conclusions and fine-tune the proposed data management strategies. So, for example, the videogame allows to evaluate if a certain data management strategy is able to provide useful information to the driver/player (i.e., if the presented information represents an advantage for him/her). We argue that this videogame can be a good complement for existing simulators. As a proof of concept, we have performed some preliminary tests that show the potential interest of the proposal.

1. INTRODUCTION

The widespread availability of mobile devices and the development of wireless communication technologies (such as Wi-Fi, WAVE, etc.) have encouraged the development of services for drivers within the context of *Intelligent Transportation Systems* (*ITS*). In particular, *Vehicular Ad Hoc Networks* (*VANETs*) have become an attractive research area [1, 14, 15, 20, 24, 26, 30]. In these vehicular networks, the vehicles can exchange information directly by using short-range wireless communication technologies. This decentralized architecture provides some advantages over other solutions such as the use of 3G communications: e.g., no need of an infrastructure, quicker transmission of safety-related data in the vicinity, localized communications without the need of a centralized server, and free of charges (which also encourages the participation of peers in the network). Numerous types of events can be relevant for drivers (e.g., accidents, traffic congestions, an ambulance asking the right of way, available parking spaces, etc.). These events can be exchanged in the vehicular network and stored locally by the vehicles. Then, a query processor can periodically evaluate the interest of those events and decide if they should be shown to the driver; there may be implicit queries (e.g., information about an accident in the direction of travel will be relevant for any driver) and explicit queries (e.g., a driver may indicate his/her interest in finding an available parking space or in receiving information about other specific types of events).

However, although VANETs offer interesting opportunities for the development of data services for drivers, they also bring new challenges. Thus, several difficulties arise from the point of view of data management [5]. As an example, estimating the relevance of events in order to disseminate them effectively and efficiently in the network is a challenge [2]. Similarly, disseminating information about a scarce resource (e.g., an available parking space) to many vehicles can lead to competition situations among them to try to reach the resource [7]. As a final example, the relevance of events must also be considered in order to decide if a specific event received by a vehicle should be shown to the driver or not [3].

A big challenge is how to evaluate the data management techniques proposed. Evaluating them in a real scenario with a significant number of vehicles is simply impractical and expensive. Therefore, simulations are frequently used in this field. However, even with simulations the evaluation task can be very time-consuming. For example, many proposals depend on a number of parameters that can be fine-tuned for a given scenario (e.g., see [2, 31]), and determining a good choice of parameters for general evaluation is quite challenging. On the other hand, crowdsourcing strategies where users play the role of drivers could help to

[1] `http://www.abiresearch.com/press/v2v-penetration-in-new-vehicles-to-reach-62-by-202`.

introduce human behavior and facilitate new tests initiated by the users themselves.

So, in this paper we propose a complementary approach that can be used in conjunction with the use of simulators. In particular, we argue that we can benefit from players having fun with a driving game to easily collect interesting data that can be used to extract some conclusions and fine-tune the proposed data management strategies. The videogame is inspired by the classic videogame *Rally-X* (`http://www.klov.net/game_detail.php?game_id=9259`, videogame released in 1980) but it is a new development, with different goals, game modes, and spirit. So, the basic idea is that the vehicles can receive information through the vehicular network and different data management techniques can be plugged in the videogame (e.g., different data dissemination strategies). Data received from other vehicles, if evaluated as interesting by the local query processor in the car, are shown on a radar and can provide a competitive advantage to the player. During the game, a variety of data are collected (e.g., number of messages received by the vehicles, network overhead, time required by the vehicles to complete their goals in the game, etc.), that can be analyzed later. So, while playing, players contribute to collect data for a variety of scenarios, and these data can be exploited to evaluate the effects of particular data management strategies.

The structure of the rest of this paper is as follows. In Section 2 we describe the high-level architecture of the videogame and its features. In Section 3 we summarize the main behaviors implemented for the computer-managed vehicles. In Section 4 we present some basic aspects about the way the data are collected for later analysis. In Section 5 we present the results of the first experiments that we have developed as a proof of concept. In Section 6 we present some related work. Finally, in Section 7 we present our conclusions and some lines of future work.

2. ARCHITECTURE AND FEATURES

Vanet-X is a car videogame that can be played by multiple players connected to the Internet (see Figure 1 for a snapshot showing parking spaces).

2.1 Main Features

We summarize some features of the game as follows:

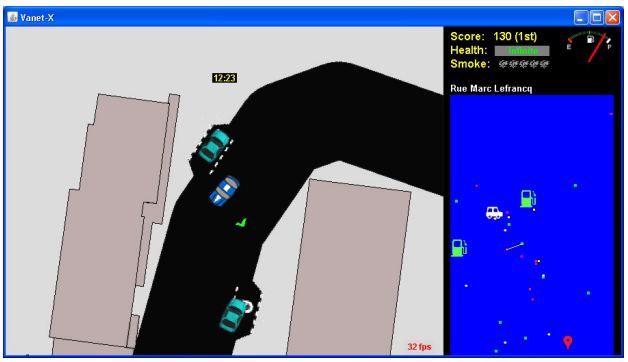


Figure 1: Cars and parking spaces in Vanet-X

- It is implemented in Java as a Java applet, so only a Java Virtual Machine and a browser is needed to play. A desktop application version is also available.
- Both real (human) and computer-controlled players can participate in the game. Human players can join a game through the Internet.
- The game can be configured to execute on a server and create new games when necessary. Alternatively, the computer of any user can play the role of a server and start a new game that other users can join.
- Any real map can be used in the game, by selecting and downloading the data of the desired area from *OpenStreetMap* (`http://www.openstreetmap.org/`).
- To increase the playability, real maps are combined with some extra elements, such as enemy cars, smoke emission devices to disturb enemies (see Figure 2), evolution of events in game time rather than in real-world time, higher maximum speeds for cars controlled by humans, when the driver has a task to go to a certain building he/she has to park nearby and then go by foot to the destination (he/she will be a vulnerable target for enemy cars, that will try to hit him/her, as shown in Figure 3), the car can get damaged and be repaired by paying a certain price (points accumulated during the game), there is infinite or limited fuel depending on the game mode (requiring refueling in a petrol station when running out of fuel in the second case), etc.

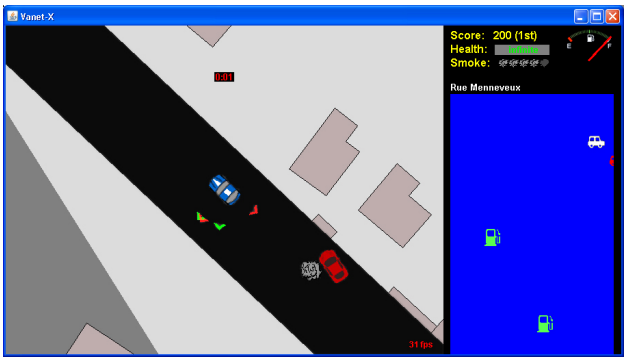


Figure 2: Trying to escape from an enemy vehicle

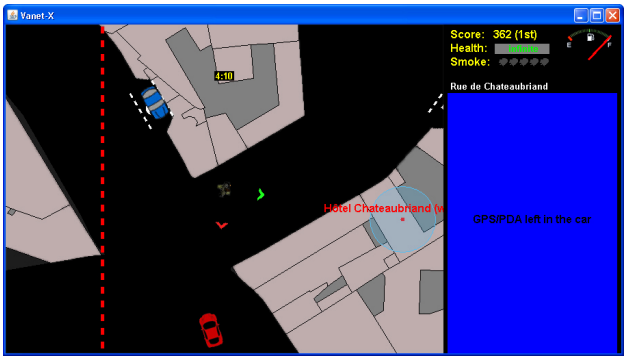


Figure 3: Driver going by foot

- A wide range of game modes is available (see Table 1). Thus, for example, we offer games where the goal is to collect some items along the roads, and task-based

games where the players have to complete a series of goals in sequence (with tasks such as parking the vehicle, going to a certain building/business or address, etc.) as soon as possible to win the game. As shown in the table, some game modes can be cooperative, competitive, or both. For the tasks implying going to a certain location, the task may require reaching that location with the car, park and then get there by foot, or just park as near as possible (in this last case, the score for completing the task will be inversely proportional to the distance between the parking location and the final destination). Competitive games involves from 1 to 4 teams in the game, being the winner the team that obtains more points during the game.

Game mode	Multiplayer mode	Inmortal	Possibility to get out of the car and walk	Infinite fuel
Capture the flag (capture 5 flags)	cooperative competitive	no	no	configurable
Capture the enemy cars (1 or more)	cooperative competitive	yes	no	no
Solve tasks (1-3 tasks)	cooperative competitive	no	yes	configurable
Survival (1 or 2 tasks)	cooperative competitive	no	yes	configurable
Park (find one available parking spot)	competitive	yes	yes	no

Table 1: Summary of game modes

- Some default data management strategies, inspired by the work performed in the VESPA project [2, 3, 4, 6, 7], have been implemented. Different tuning parameters can be modified through the graphical user interface of the videogame (see Figures 4 and 5). Moreover, the design of the videogame allows an easy integration of other data management alternatives.

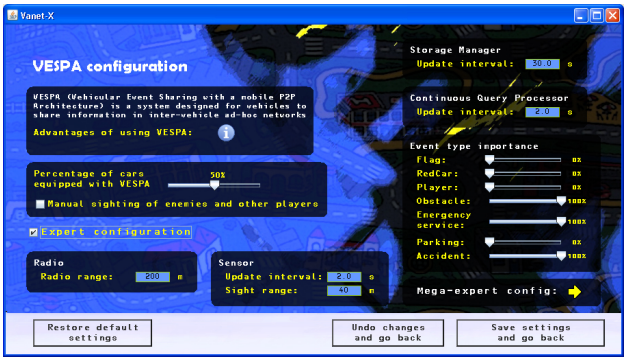


Figure 4: Data management: basic options

- There is a "radar" (e.g., on the right part of Figure 2 we show a basic radar, and on the right part of Figure 1 a radar in debug mode that shows some extra elements about the scenario) that can provide some information to the players. For example, a player can see the following on the radar: his/her location, the petrol stations, and the destination location (if any). Besides, if the option to use a data sharing strategy for that vehicle has been enabled, it will also show data about interesting events received from other vehicles, such as free parking spaces, enemy vehicles, items to pick up (e.g., flags in Figure 6), priority vehicles like ambulances, etc.

Figure 5: Data management: advanced options

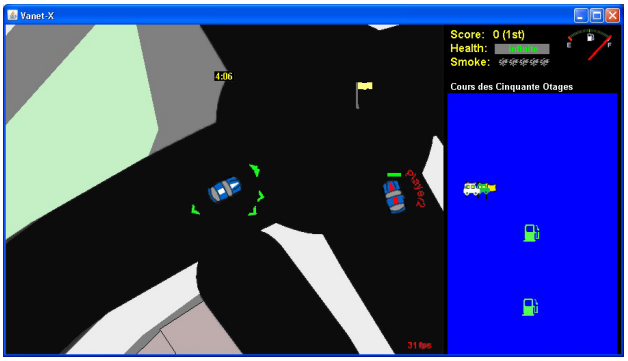


Figure 6: Picking up flags during the game

From a more technical point of view, we have used the Java programming language to develop the video game. Besides, some auxiliary libraries have been useful. For example, we use *Apache Xerces2 Java* (`http://xerces.apache.org/#xerces2-j`) to extract data from the XML files obtained from OpenStreetMap, *JLayer* (`http://www.javazoom.net/javalayer/javalayer.html`) to decode and reproduce MP3 files for the game music, *Guava-12.0* (`https://code.google.com/p/guava-libraries/`), etc.

2.2 Basic Architecture

The basic architecture of the videogame is presented in Figure 7 (the part concerning the collection of statistics about the game is not shown here, as it will be described in Section 4). At a high-level, we can briefly describe the main components as follows:

- A *client* application receives commands from the player, sends them to the server, and receives from the server information about the objects that should be rendered on the screen (see Figure 8).
- The *server* receives the input from the clients, updates the current status of the game (e.g., by considering the movements performed by the vehicles and the tasks that they complete), and generates new goals and events as needed (see Figure 9). The server is multithreaded, with a thread per vehicle that performs a

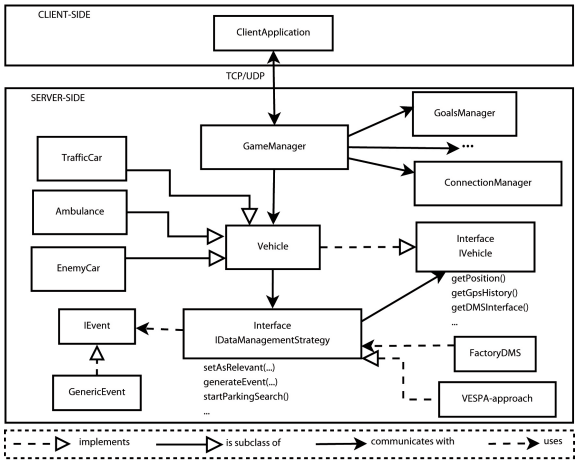


Figure 7: Basic architecture of Vanet-X

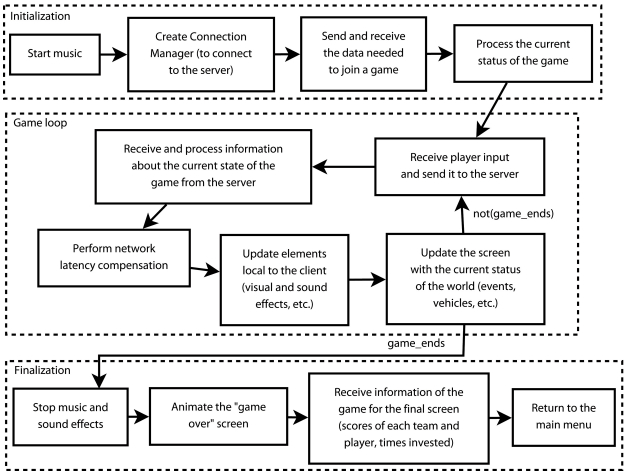


Figure 8: Basic functioning of a client

basic cycle of "while a vehicle is alive, perform actions and check for potential collisions".

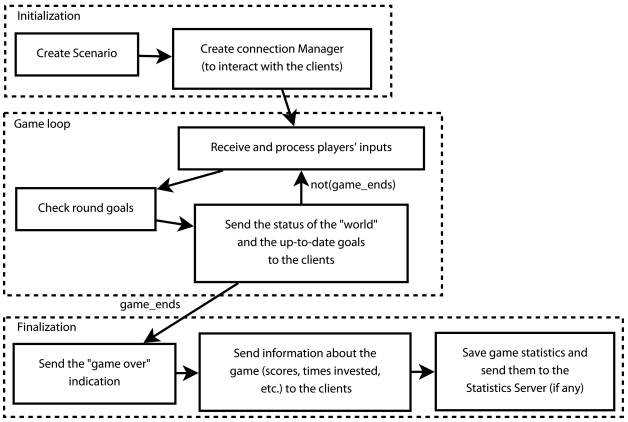


Figure 9: Basic functioning of the server

- An interface *IDataManagementStrategy* declares the methods that should be implemented by a *data management strategy* to allow its integration with the videogame (e.g., a method to define the types of events that are interesting for the driver, a method to generate an event, etc.).
- Another interface *IVehicle* is implemented by the *vehicles* to allow interacting with them (e.g., to obtain a reference to the data manager in the vehicle or to obtain information about the GPS location).

Any data management strategy can potentially be integrated in this framework, as long as it implements the interface *IDataManagementStrategy* and calls the appropriate methods to inform the vehicles (interface *IVehicle*). So, we can easily plug in different alternative data management techniques for testing.

3. BEHAVIORS OF THE VEHICLES

We have implemented several behaviors for the vehicles controlled by the computer, which adapt the steering behaviors proposed in [23]. In particular, we consider the following basic behaviors:

- *Seek* implies directing the vehicle towards a certain static target, by adjusting its direction and speed.
- *Flee* is the opposite behavior to *Seek*, as it implies getting as much further as possible from the target.
- *Pursuit* is similar to *Seek*, but in this case the target is a moving object. So, the expected movement of the target is estimated, to try to catch it.
- *Evasion* is the opposite behavior to *Pursuit* (i.e., based on *Flee* instead of *Seek*).
- *Arrival* implies the progressive reduction of speed as the vehicle approaches the target.
- *Obstacle avoidance* provides vehicles with the ability to dodge vehicles and other obstacles.
- *Wander* generates a random trajectory, to represent a vehicle traveling around with no clear objective. This is useful, for example, to represent a vehicle that is searching for an available parking space in the vicinity.
- *Path following* allows a vehicle to circulate within the boundaries of a certain path.
- *Unaligned collision avoidance* is a behavior that tries to avoid the collision of vehicles moving in different directions. Thanks to this behavior, vehicles can estimate a potential collision risk with other vehicles in the near future, to try to avoid it.

Of course, all the vehicles exhibit the whole set of behaviors at the same time, applying a priority ordering in case several behaviors could be applied at the same time and are in conflict to each other. Based on the previous basic behaviors, we have defined the schema of a normal behavior for different types of vehicles: enemy cars (that try to catch the players or flee from the players, depending on the game mode), ambulances (as representatives of emergency vehicles which may ask the right of way), and traffic cars (that represent neutral cars in the game). As an example, the basic behavior of traffic cars is shown in Figure 10.

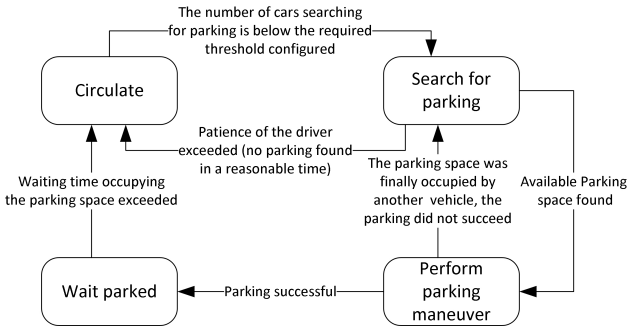


Figure 10: Basic behavior of a traffic car

4. DATA COLLECTION AND EXPLOITATION

In this section, we summarize the strategy applied for data collection during the game and the corresponding exploitation of results. If a certain configuration option that activates the collection of statistics during the game is enabled, several data are collected: data about the scores obtained by the players, the time needed by vehicles (the ones controlled by humans as well as those managed by the computer) to perform certain tasks (such as parking), and other measures about the performance of the data management strategy applied (e.g., events created, events that are considered relevant by each vehicle, etc.). When the game ends, all these data are stored in several files on the game server, along with a file that contains information about all the configuration parameters used in that game (e.g., game mode, configuration parameters used for the data management strategy considered, the wireless communication range simulated, etc.).

To centralize the data collected, it is possible to set up a *Statistics Server*, which is a process executing continuously on a certain computer. In this way, the clients playing the game automatically connect to the Statistics Server when a game ends, in order to communicate the statistics collected during the game. Besides, it is possible to connect to the *Master Server* by using a terminal client (called *Statistics Client*) that allows seeing and modifying the configuration parameters as well as retrieving the statistics files generated. Another option is to avoid the use of a Statistics Server and collect the statistics in the computer that plays the role of a server for a game. If we consider configuration settings where there is a predefined game server and all the clients connect to it to start a new game or join an existing game, this option also keeps the statistics in a single location. However, if there are several game servers then the statistics would have to be centralized manually.

Figure 11 provides an overview of the way the different components of the game, and particularly the Statistics Server, are distributed in a network. Notice that we actually distinguish between a *Master Server* and a *Game Server*. The Master Server is executing on the server machine and a client first connects to it (so, it is the entry point for clients); then the Master Server checks if a Game Server is available and if not it creates one; finally, it returns the port number of its Game Server to the client, as the client will interact with the Game Server during the game.

It should be noted that, as we collect information about the performance of human players, the skills of those players

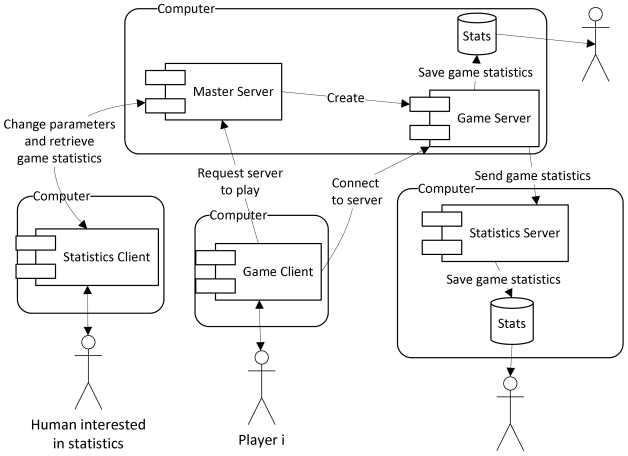


Figure 11: Deployment of components in a network

with the game will have an impact on the results and this has to be taken into account when exploiting the results. Indeed, directly comparing the achievements of several human players without considering their game skills could lead to wrong conclusions. For example, *player1* without a data sharing system could perform better than *player2* with a data sharing system, but we should not necessarily conclude that the use of such a data sharing system is harmful. In other words, we should always compare players with the same skills. For this reason, each human player is assigned a certain *skill level* (which may change along time, as the player improves his/her performance) and the statistics about players are tagged with the skill level corresponding to that player. Besides, players that have a skill level below a certain threshold are (by default) not allow to participate in games with collection of statistics enabled, as performance data about them are assumed to be unreliable and besides their clumsiness could interfere with the normal development of the game. The skill level of a player is computed based on his/her ability to complete missions in the game (tasks per time unit).

5. EXPERIMENTAL EVALUATION

We have performed a few preliminary experiments to evaluate the interest of our proposal. As a use case for testing, we focused on the case of available parking spaces, as these are events that represent scarce resources, which implies additional challenges for data management (i.e., the competition among vehicles should be minimized).

5.1 Data Management Strategies

As a data sharing strategy for the vehicles, we considered the following options.

5.1.1 VESPA-P: *VESPA With No Reservation*

First, we adapted the proposal in [2], developed in the context of the system *VESPA* (*Vehicular Event Sharing with a mobile P2P Architecture*) [4, 6], which is based on the computation of an *Encounter Probability* (*EP*).

The EP between a vehicle and an event estimates the likelihood that the vehicle will meet the event, based on geographic computations that estimate the spatio-temporal relevance of the event. For example, the relevance decreases

with the distance between the event and the vehicle, with the time since the event was generated (e.g., consider the case of information about an available parking space, which can be unoccupied only for a limited amount of time), and the direction of the vehicle (e.g., if it is approaching the event or not). In particular, the directions of both the vehicle and the event are estimated and several *penalty coefficients* (α, β, γ, and ζ) are used to weigh the importance of four estimated parameters: the minimum distance to the event over time (Δd), the time until the closest position to the event (Δt), the age of the event at the closest position (Δg), and the angle between the vehicle and the event (c).

So, when a vehicle receives an event it computes its EP and disseminates the event again if the computed EP exceeds a certain *dissemination threshold* (DT). The intuition is that vehicles should disseminate data that are relevant for them (as those data are also probably relevant for the neighboring vehicles). Two other thresholds are managed: the *storage threshold* (ST) and the *relevance threshold* (RT). The ST determines the minimum value of the EP for an event to be stored locally in the vehicle, and the RT the minimum value needed to show the event to the driver.

Besides, the proposal in [2] proposes a contention-based approach for data dissemination in order to limit the network overhead in the dissemination of messages (basically, when there are several candidate vehicles to re-disseminate an event, the message will be disseminated only by the vehicle located further away from the vehicle that disseminated the message previously). Several parameters are used in the protocol, such as D (the maximum time to wait before rediffusing) and D' (time to wait for an acknowledgement that a message sent previously was received by some other vehicle).

5.1.2 VESPA+P: *VESPA With Reservation Protocol*

Communicating the availability of a single parking space to many vehicles could lead to an unfruitful competition among the vehicles to try to reach the same parking space, leading to dissatisfaction of the drivers and parking times that could even exceed those that would be obtained if no data sharing system were used. For this reason, the work presented in [7] proposed an enhancement to the previous approach *VESPA-P* for the case of scarce resources such as parking spaces. It provides an allocation protocol that coordinates a procedure that ensures that the information about an available parking space is communicated to a single interested vehicle.

5.1.3 Blind: *No Data Sharing*

Finally, we also considered an approach where no data sharing strategy is used. In this case, the vehicles receive no information and the only data available for the drivers are what they see with their own eyes. For vehicles trying to find available parking spaces, this will lead to a *blind search*.

5.2 Experimental Settings

The basic configuration of the videogame for the experimental evaluation is as follows. The communication range considered for the vehicles is 200 meters and a maximum of 50% of the vehicles are assumed to be equipped with a data sharing application. The penalty coefficients used to compute the EP for VESPA are: α=1/1500 ($\Delta d \leq 500$ meters), β=1/180 ($\Delta t \leq 60$ seconds), γ=1/360 ($\Delta g \leq 120$ seconds), and ζ=1/270 ($c \leq 90°$); these are parameters that can be considered for a "medium" (not small, not large) dissemination area, according to [2]. The RT and the DT are both set to 75%, and the ST is 60%. The query processor on each vehicle re-evaluates the relevance of the events received with a refreshment period of 2 seconds, showing on the radar the events that are considered relevant. For the dissemination protocol, D is set to 1 second and D' to 2 seconds.

5.3 Experimental Results

We have simulated a varying number of vehicles moving in an area of 1 squared kilometer around the street "Sophie Oury" in the city of Valenciennes (France). In this scenario, we measured the time needed by the vehicles to find free parking spaces near certain destinations. In Figure 12 we show the reduction on the average time needed by a human player to find an available parking space near the target. The experimental results show the interest of sharing data among the vehicles (with both *VESPA-P* and *VESPA+P*), as these data can later be shown on the radar to provide interesting information to the drivers. Besides, according to these results, using a reservation protocol to avoid the competition problem (*VESPA+P*) is particularly beneficial.

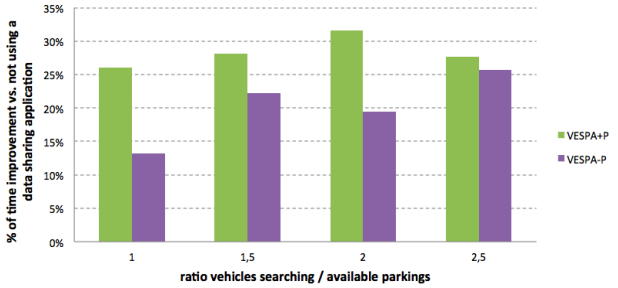


Figure 12: Time to park by a human

In Figure 13 we compare the performance of human players (vehicles controlled by humans) and computer players (vehicles controlled by the computer), by showing the reduction on the average time needed to find an available parking space near the target when using *VESPA+P*. According to these experimental results, we can see that the human players get more benefit from the use of the data sharing strategy. The difference may be due to the way the artificial intelligent behavior of the computer vehicles is implemented.

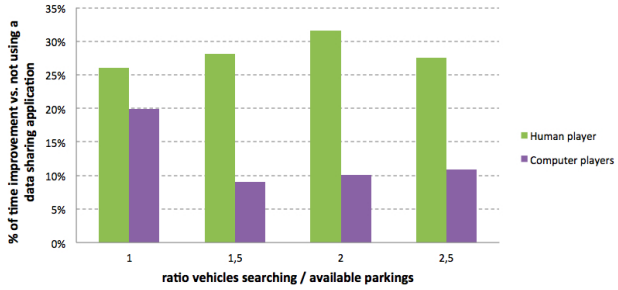


Figure 13: Time to park: human vs. computer

The experimental results obtained correspond to data collected during a total of 14 hours playing the videogame (about 400 parking actions by the human player during this

game time). The results are consistent with our intuition and with other experimental results obtained previously by using a simulator. Nevertheless, more tests are needed to validate the results and evaluate other scenarios. For example, we started to obtain some first preliminary results with games played by more than one human player. It is also interesting to perform experiments with other types of events (e.g., accidents, obstacles on the roads, etc.); with information about them, drivers could try to avoid those hazards and so decrease the total travel time.

6. RELATED WORK

As far as we know, this is the first attempt to develop a videogame whose hidden purpose is to help with the evaluation of information management strategies for vehicular networks.

Nevertheless, the idea of trying to benefit from human actions to improve or evaluate a system is not new. Exploiting the power of people to perform large-scale tasks that are costly, time-expensive, or hard, is called *crowdsourcing* [33]. For example, *mCrowd* [32] benefits from sensors available on iPhone devices to perform collaborative tasks such as image tagging or road traffic monitoring. As another example, *reCAPTCHA* [29] exploits *CAPTCHAs* [22] (Completely Automated Public Turing test to tell Computers and Humans Apart), as a security measure to avoid web access to programs, in order to recognize words from scanned books that are challenging for OCR (Optical Character Recognition) systems. According to [10], "The practice of crowdsourcing is transforming the Web and giving rise to a new field".

Particularly relevant for our work with Vanet-X are those proposals that achieve the crowdsourcing results through the use of a videogame. A notable example is the *ESP game* [27], where players implicitly help to label images while playing the game. The use of videogames as learning tools is a clear example of the benefits of using educative videogames; as an example, *CodeSpells* [11] is a fantasy videogame where players have to write spells in Java. Other games with a hidden purpose exist, as commented in [28]. The multiplayer online game *Planet PI4* [16] intends to serve as a testbed environment for Peer-to-Peer (P2P) game architectures. It is also interesting to mention that the term *gamification* has appeared to denote a variety of software that is inspired somehow by videogames [8, 9].

There exist some driving videogames that, as Vanet-X, are based on the use of real road maps or city layouts, such as *Mini Maps* (`https://apps.facebook.com/minimaps/`) and *Push-Cars 2: On Europe Streets* (`http://www.push-cars.com`). However, unlike in Vanet-X, in these games the players do not contribute to any crowdsourcing task or data management strategy evaluation.

Finally, a good number of simulators of vehicular networks and mobility generators have been developed, such as *TraNS* [21], *SUMO* [19], *Veins* (Vehicles in Network Simulation) [25], *GrooveNet* [17], or *VanetMobiSim* [13]. Some interesting surveys can be found in [12, 18]. As commented along the paper, we argue that the videogame-based approach can be an interesting complement (but not a replacement) to the use of existing simulators to evaluate information management strategies for vehicular networks. Besides, mobility generators and vehicle simulators could potentially be used to generate neutral traffic for Vanet-X.

7. CONCLUSIONS AND FUTURE WORK

We have developed a videogame that can be used to evaluate data management strategies for vehicular networks, as a complement to existing simulators. Whereas the opportunity of crowdsourcing through a videogame is attractive, several challenges arise. Thus, the goal of developing a fun videogame required the introduction of several elements that would not appear in a real scenario (like enemy cars), which could have an impact on the results, but on the other hand this will attract people to play. Moreover, the results obtained can depend not only on the benefits offered by the data management strategy but also on the ability of the specific player. So, whereas the videogame can provide an ideal tool to collect many data for a variety of scenarios, the experimental results obtained have to be judged with caution (e.g., we label the collected data with the skill level of the player). Even with these limitations, we argue that the videogame helps to collect with less effort data that can be used to fine-tune a protocol and/or obtain some initial conclusions, prior to the evaluation in more realistic scenarios.

Additional information regarding the videogame is available at `http://sid.cps.unizar.es/Vanet-X/`, including a playable version of the videogame, some videos, and screenshots. This is a first step that shows the potential interest of exploiting videogames to evaluate data management strategies for vehicular networks. As future work, we would like to optimize and improve the videogame, as well as to develop a complete methodology and architecture to collect the data, evaluating the interest of the results obtained in other scenarios and in a larger scale.

8. ACKNOWLEDGMENTS

This research work is currently supported by the CICYT project TIN2010-21387-C02-02 and DGA-FSE. The data management strategy adapted and used as an example in the videogame has been proposed in the context of the VESPA project, and we would like to warmly acknowledge the collaboration with Dr. Thierry Delot in that project.

9. REFERENCES

[1] J. J. Blum, A. Eskandarian, and L. J. Hoffman. Challenges of intervehicle ad hoc networks. *IEEE Transactions on Intelligent Transportation Systems*, 5(4):347–351, 2004.

[2] N. Cenerario, T. Delot, and S. Ilarri. A content-based dissemination protocol for VANETs: Exploiting the encounter probability. *IEEE Transactions on Intelligent Transportation Systems*, 12(3):771–782, 2011.

[3] T. Delot, N. Cenerario, and S. Ilarri. Vehicular event sharing with a mobile peer-to-peer architecture. *Transportation Research Part C: Emerging Technologies*, 18(4):584–598, 2010.

[4] T. Delot and S. Ilarri. Data gathering in vehicular networks: The VESPA experience (invited paper). In *Fifth IEEE Workshop On User MObility and VEhicular Networks (LCN ON-MOVE 2011)*, pages 801–808. IEEE Computer Society, 2011.

[5] T. Delot and S. Ilarri. Introduction to the Special Issue on Data Management in Vehicular Networks. *Transportation Research Part C: Emerging Technologies*, 23:1–2, 2012.

[6] T. Delot and S. Ilarri. The VESPA Project: Driving advances in data management for vehicular networks. *ERCIM News*, (94):17–18, July 2013. Special Theme on "Intelligent Vehicles as an Integral Part of Intelligent Transport Systems".

[7] T. Delot, S. Ilarri, S. Lecomte, and N. Cenerario. Sharing with caution: Managing parking spaces in vehicular networks. *Mobile Information Systems*, 9(1):69–98, 2013.

[8] S. Deterding, D. Dixon, R. Khaled, and L. Nacke. From game design elements to gamefulness: Defining "gamification". In *15th International Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek'11)*, pages 9–15. ACM, 2011.

[9] S. Deterding, M. Sicart, L. Nacke, K. O'Hara, and D. Dixon. Gamification: Using game-design elements in non-gaming contexts. In *2011 Annual Conference on Human factors in Computing Systems (CHI'11) – Extended Abstracts*, pages 2425–2428. ACM, 2011.

[10] A. Doan, R. Ramakrishnan, and A. Y. Halevy. Crowdsourcing systems on the World-Wide Web. *Communications of the ACM*, 54(4):86–96, 2011.

[11] S. Esper, S. R. Foster, and W. G. Griswold. On the nature of fires and how to spark them when you're not there. In *44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*, pages 305–310. ACM, 2013.

[12] J. Harri, F. Filali, and C. Bonnet. Mobility models for vehicular ad hoc networks: A survey and taxonomy. *IEEE Communications Surveys & Tutorials*, 11(4):19–41, 2009.

[13] J. Härri, F. Filali, C. Bonnet, and M. Fiore. VanetMobiSim: Generating realistic mobility patterns for VANETs. In *Third International Workshop on Vehicular Ad Hoc Networks (VANET'06)*, pages 96–97. ACM, 2006.

[14] H. Hartenstein and K. P. Laberteaux. A tutorial survey on vehicular ad hoc networks. *IEEE Communications Magazine*, 46(6):164–171, 2008.

[15] G. Karagiannis, O. Altintas, E. Ekici, G. J. Heijenk, B. Jarupan, K. Lin, and T. Weil. Vehicular networking: A survey and tutorial on requirements, architectures, challenges, standards and solutions. *IEEE Communications Surveys & Tutorials*, 13(4):584–616, 2011.

[16] M. Lehn, C. Leng, R. Rehner, T. Triebel, and A. Buchmann. An online gaming testbed for peer-to-peer architectures. *ACM SIGCOMM Computer Communication Review*, 41(4):474–475, 2011.

[17] R. Mangharam, D. S. Weller, and R. Rajkumar. GrooveNet: A hybrid simulator for vehicle-to-vehicle networks. In *Second International Workshop Vehicle-to-VehicleCommunications (V2VCOM'06)*, pages 1–8, 2006.

[18] F. J. Martinez, C. K. Toh, J.-C. Cano, C. T. Calafate, and P. Manzoni. A survey and comparative study of simulators for vehicular ad hoc networks (VANETs). *Wireless Communications & Mobile Computing*, 11(7):813–828, 2011.

[19] J. E. Michael Behrisch, Laura Bieker and D. Krajzewicz. SUMO – Simulation of Urban MObility: An overview. In *The Third International Conference on Advances in System Simulation (SIMUL'11)*, pages 63–68. IARIA, 2011.

[20] S. Olariu and M. C. Weigle, editors. *Vehicular Networks: From Theory to Practice*. Chapman & Hall/CRC, 2009.

[21] M. Piorkowski, M. Raya, A. L. Lugo, P. Papadimitratos, M. Grossglauser, and J.-P. Hubaux. TraNS: Realistic joint traffic and network simulator for VANETs. *SIGMOBILE Mobile Computing and Communications Review*, 12(1):31–33, 2008.

[22] C. Pope and K. Kaur. Is it human or computer? Defending e-commerce with Captchas. *IT Professional*, 7(2):43–49, 2005.

[23] C. W. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, pages 763–782. Miller Freeman Game Group, 1999.

[24] M. L. Sichitiu and M. Kihl. Inter-vehicle communication systems: A survey. *IEEE Communications Surveys & Tutorials*, 10(1–4):88–105, 2008.

[25] C. Sommer, R. German, and F. Dressler. Bidirectionally coupled network and road traffic simulation for improved IVC analysis. *IEEE Transactions on Mobile Computing*, 10(1):3–15, 2011.

[26] Y. Toor, P. Mühlethaler, A. Laouiti, and A. de La Fortelle. Vehicle ad hoc networks: Applications and related technical issues. *IEEE Communications Surveys & Tutorials*, 10(1–4):74–88, 2008.

[27] L. von Ahn and L. Dabbish. Labeling images with a computer game. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'04)*, pages 319–326. ACM, 2004.

[28] L. von Ahn and L. Dabbish. Designing games with a purpose. *Communications of the ACM*, 51(8):58–67, 2008.

[29] L. von Ahn, B. Maurer, C. McMillen, D. Abraham, and M. Blum. reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[30] T. L. Willke, P. Tientrakool, and N. F. Maxemchuk. A survey of inter-vehicle communication protocols and their applications. *IEEE Communications Surveys & Tutorials*, 11(2):3–20, 2009.

[31] B. Xu, A. M. Ouksel, and O. Wolfson. Opportunistic resource exchange in inter-vehicle ad-hoc networks. In *Fifth IEEE International Conference on Mobile Data Management (MDM'04)*, pages 4–12. IEEE Computer Society, 2004.

[32] T. Yan, M. Marzilli, R. Holmes, D. Ganesan, and M. Corner. mCrowd: A platform for mobile crowdsourcing. In *Seventh ACM Conference on Embedded Networked Sensor Systems (SenSys'09)*, pages 347–348. ACM, 2009.

[33] M.-C. Yuen, I. King, and K.-S. Leung. A survey of crowdsourcing systems. In *Third International Conference on Privacy, Security, Risk and Trust (PASSAT 2011) and Third International Conference on Social Computing (SocialCom 2011)*, pages 766–773. IEEE, 2011.

Anexo F

Manual de usuario

Guía de usuario

Vanet-X

Autor:
Víctor Rújula (victor.rujula@gmail.com)

21 de agosto de 2013

Índice

Nota del autor

VANET-X esta localizado en inglés ya que, por la imposibilidad de localizarlo en diferentes idiomas, se decidió usar el idioma con un mayor público objetivo. Por ese motivo cuando se haga referencia en este documento a los diferentes títulos de los menús o a las opciones que aparecen en dichos menús, se nombrarán con la designación original, en inglés, indicándose junto a ellos la traducción correspondiente.

1. Instalación

Este juego no requiere de instalación, únicamente debe ejecutarse el archivo JAR que puede obtenerse mediante descarga del sitio web oficial o mediante la compilación del código fuente proporcionado.

1.1. Requerimientos del sistema

Se ha comprobado que esta aplicación funciona correctamente con la siguiente configuración:

- Windows XP SP 3 / Ubuntu 11.10 / Mac OS X 10.6.1
- 2.81 GHz AMD Phenom(tm) II X3 720 processor
- 3.25 GB RAM
- NVIDIA GeForce GTX 260 @ 1920 x 1080 px.
- Java VM 1.7.0_09

Los requisitos imprescindibles para la ejecución de VANET-X son los siguientes:

- Teclado y ratón.
- Resolución de video igual o superior a 1024 x 768 px.
- Java VM 1.6 o superior.

1.2. Compilación

El proceso requerido para la compilación difiere dependiendo del sistema operativo usado.

1.2.1. Linux / Mac OS X

Usando el script shell Se debe extraer el fichero comprimido y ejecutar el script shell *compilar.sh* desde el interior de la carpeta extraída.

```
\$> sh compilar.sh
```

Nota: se crea un certificado para firmar el fichero JAR (si no se había creado previamente), para lo cual es necesario seguir las instrucciones en pantalla. Para ello se crea un fichero de almacenamiento de claves «keystore», situado en la carpeta desde la que se ejecuta el comando.

Usando Ant Se debe extraer el fichero comprimido y ejecutar Ant desde el interior de la carpeta extraída.

```
\$> ant
```

Nota: se crea de forma transparente al usuario un certificado para firmar el fichero JAR.

1.2.2. Windows

Usando Ant Se debe extraer el fichero comprimido y ejecutar Ant desde el interior de la carpeta extraída.

```
\$> ant
```

Nota: se crea de forma transparente al usuario un certificado para firmar el fichero JAR.

Nota: Para instalar **ant**, se debe descargarlo (por ejemplo de `http://ant.apache.org/bindownload.cgi`), extraer el fichero zip en algún lugar y añadir a la variable del sistema PATH el directorio *bin* obtenido de la extracción. De esta forma se habilita el uso del comando **ant** para usarlo desde la línea de comandos.

1.3. Ejecución

El proceso requerido para la ejecución difiere dependiendo del sistema operativo usado.

1.3.1. Linux / Mac OS X

Una vez se haya completado la compilación, se debe ejecutar el script shell *ejecutar_jar.sh* para ejecutar el juego como una aplicación de escritorio Java

```
\$> sh ejecutar_jar.sh
```

o ejecutar el script shell *ejecutar_ applet.sh* para ejecutarlo como un applet usando la aplicación *appletviewer*.

```
\$> sh ejecutar_applet.sh
```

1.3.2. Windows

Una vez se haya completado la compilación, se debe ejecutar el fichero llamado *RallyX3.jar*, creado en el directorio «dist»

```
\$> dist\RallyX3.jar
```

1.4. Finalización de la aplicación

Para finalizar la aplicación, únicamente se necesita pulsar sobre el botón «X» (Cerrar) de la ventana de la aplicación.

2. Menús del juego

En esta sección se explicarán las funciones realizadas por las diferentes pantallas del menú. En la figura 1 se puede observar cómo están distribuidas estas pantallas.

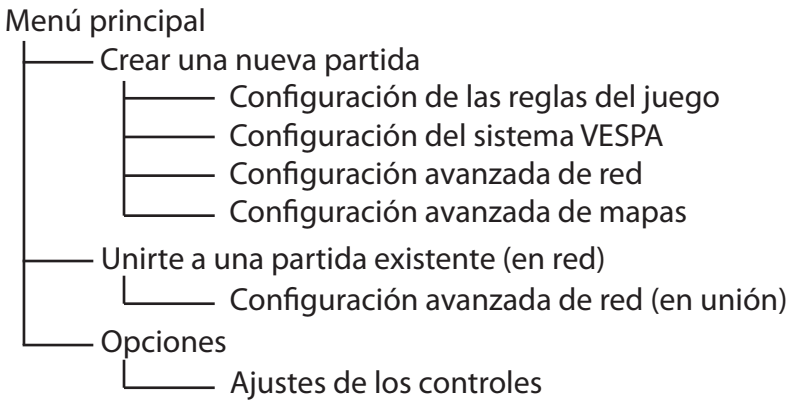
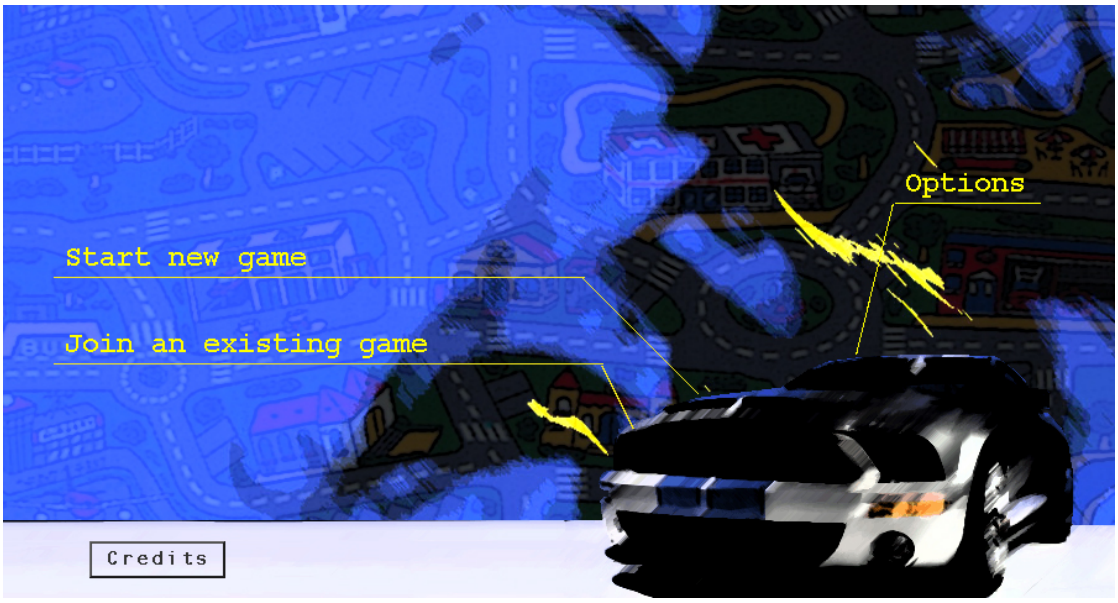


Figura 1: Diagrama de navegación menús

2.1. Menú principal



En el menú principal, se puede escoger entre «Start a new game» (crear una nueva partida), lo cual nos dirige a configurar y posteriormente comenzar una nueva partida, bien sea de un solo jugador o multijugador en red, o «Join an existing game» (unirse a una partida existente), lo cual nos permite unirnos a una partida en red en curso.
También se puede modificar diversas opciones (como el volumen, los controles o el directorio de juego) pulsando sobre «Options» (opciones).
Pulsando en el botón «Credits» (créditos) se accede a una pantalla en la que se muestra información sobre el autor, una breve introducción a VESPA y los agradecimientos.

2.2. Crear una nueva partida

En la sección «Start new game» (crear una nueva partida), se debe seleccionar el apodo que se mostrará a los demás jugadores, el mapa en el que se desea jugar y el modo de juego deseado.
Además, también se puede ver la dirección IP del computador (tanto la IP pública como la privada) en el campo de texto junto al botón «view IP» (ver IP). Pulsando sobre este botón se alterna entre mostrar un tipo u otro de dirección IP.
El resto de jugadores que deseen unirse a la partida necesitarán conocer la dirección IP pública, por lo que es importante anotarla o recordarla.

En esta pantalla de menú existen cuatro botones de configuración.
El primero, «VESPA configuration» (configuración de VESPA), permite cambiar constantes, valores y modos relacionados con VESPA, así como también establecer el porcentaje de vehículos del tráfico equipados con el sistema VESPA.

El segundo, «Advanced network configuration» (configuración avanzada de red), permite cambiar el puerto en el cual el servidor escuchará a la espera de peticiones de conexión de nuevos jugadores, así como también el puerto que usará nuestro cliente para conectarse.
Es importante asegurarse de abrir en el NAT/firewall los puertos seleccionados tanto en TCP como UDP.
En esta pantalla también aparece un desplegable en el cual se debe seleccionar la dirección IP correspondiente al adaptador de red que deseamos usar para crear la partida.

El tercero, «Advanced map configuration» (configuración avanzada de mapas), nos dirige a una nueva pantalla en la cual se podrá añadir, eliminar y previsualizar los mapas.

El último, «Game rules configuration» (configuración de las reglas del juego), permite cambiar algunas opciones como la dificultad, el número de equipos y el número de vehículos neutrales (del tráfico).

Cuando se hayan establecido todas las opciones como se desea, se debe pulsar en el botón «Start» (comenzar) para comenzar la partida. Si por el contrario deseamos volver a la pantalla anterior, se debe pulsar en el botón «Go back» (volver).

2.3. Configuración de las reglas del juego

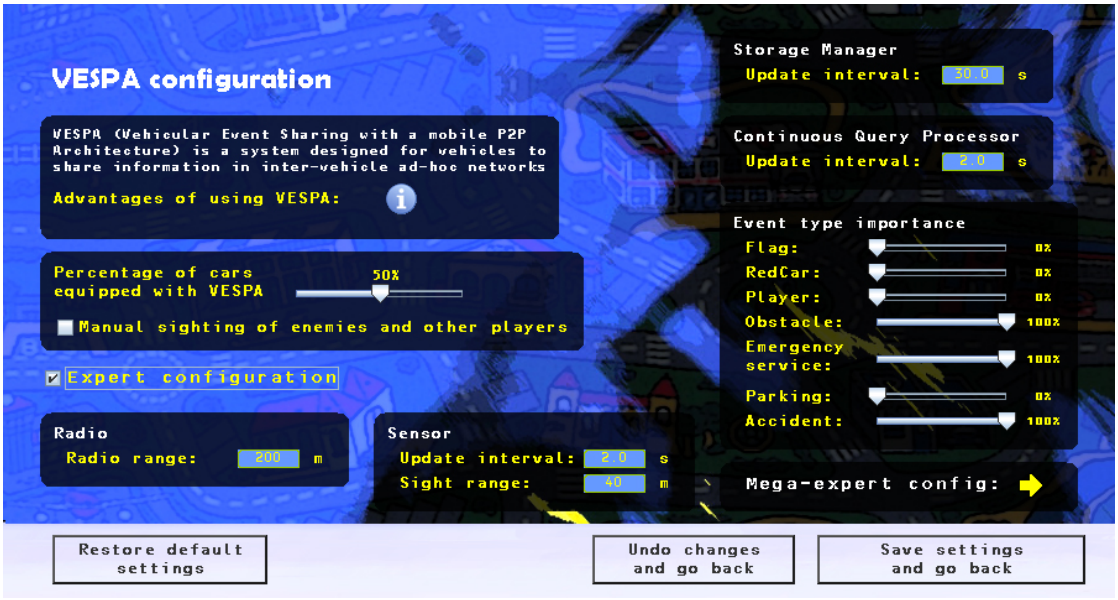
En este menú, en la mitad izquierda, se puede personalizar la dificultad cambiando los factores separadamente.
Se debe tener en cuenta que cuando el jugador asume el rol de perseguidor (modo de juego «capture the red cars») el valor del daño se revierte.
Ejemplo: Un valor bajo de daño del vehículo, en un modo de juego en el que el jugador huya de los vehículos controlados por el computador significa que el jugador tiene una resistencia mayor de la normal, mientras que en un modo de juego en el que el jugador sea el que persigue a los vehículos del computador significa que dichos vehículos tienen menos resistencia y por lo tanto es más fácil para el jugador cazarlos.

En la mitad derecha de la pantalla, se puede cambiar el número del máximo de equipos, el número de vehículos neutrales (del tráfico), el número de vehículos

enemigos y el número de rondas establecido como límite en los modos de juego competitivos.
Debajo, se puede configurar el tiempo límite de los diferentes modos de juego así como también el tiempo de espera entre rondas.

2.4. Configuración del sistema VESPA

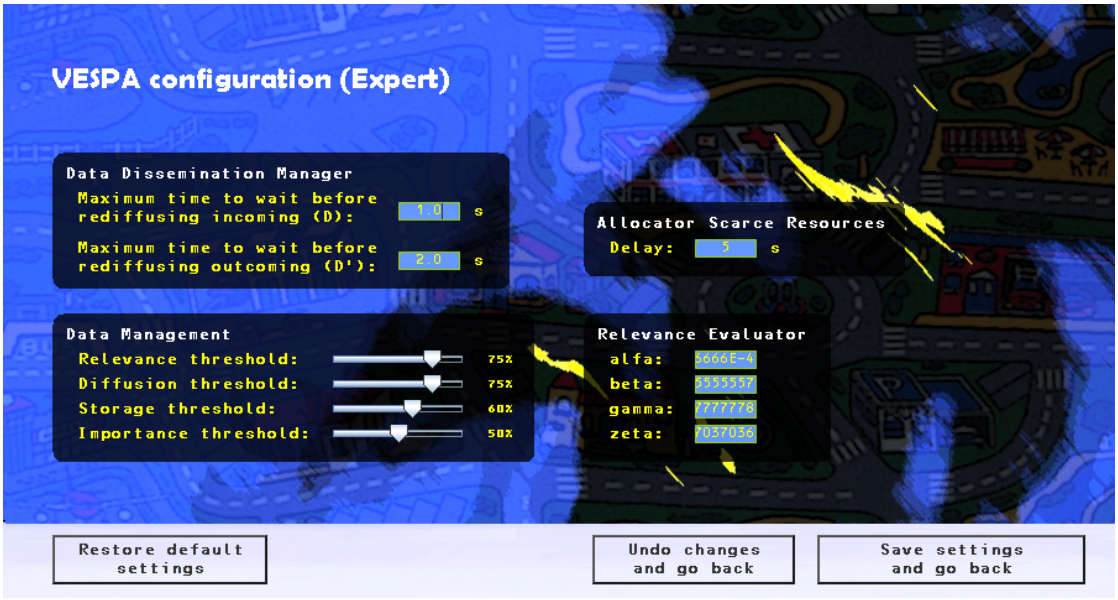
En este menú es posible configurar una gran variedad de parámetros relacionados con el funcionamiento del protocolo VESPA.
En la pantalla inicial se encuentra una muy breve descripción del sistema VESPA y sus ventajas (accesible manteniendo el puntero encima del símbolo de información), también un selector del porcentaje de vehículos del tráfico que se desea que estén equipados con el protocolo VESPA y una casilla de verificación que en el caso de estar activada indica que la generación de eventos avisando de enemigos y otros jugadores se realizará mediante observación en lugar de autogeneración desde los propios vehículos.
Por último existe otra casilla de verificación «Expert configuration» (configuración experta) que de activarse despliega más aspectos configurables del protocolo.



Las opciones que se pueden configurar son las siguientes:

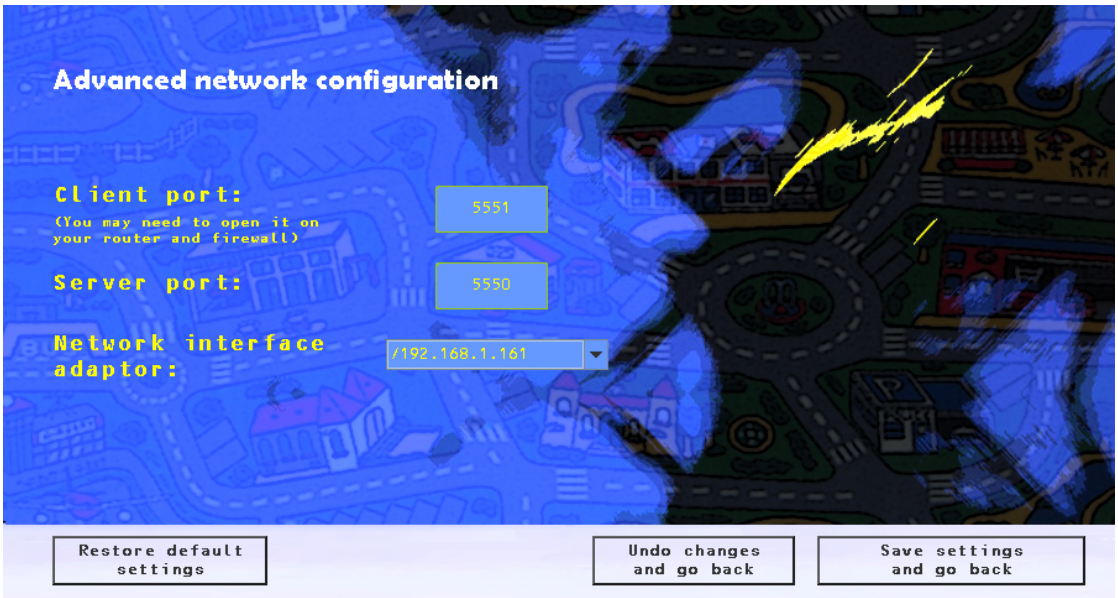
- Alcance de la radio («Radio range») y alcance de visión («Sight range»), que indican la distancia a la que se pueden transmitir y observar los eventos respectivamente.
- Intervalo de actualización («Update interval») del sensor («Sensor»), del gestor de almacenamiento («Storage Manager») y del procesador de consultas continuo («Continuous Query Processor»), que indica cada cuántos segundos entrarán en funcionamiento dichos módulos.
- Importancias de los distintos tipos de eventos («Event type importance»), que indica la importancia que se dará a un evento de dicho tipo, de forma que cuanto mayor sea mayor será la posibilidad de que se muestre al conductor.

Además, pulsando sobre la flecha amarilla «Mega-expert config» se accede a una segunda página con más elementos para configurar.



En esta segunda página de configuración se permite modificar más parámetros de VESPA. Dichos parámetros solo deberían ser modificados por una persona con unos mínimos conocimientos del protocolo ya que mediante su modificación se puede alterar gravemente el comportamiento del sistema.

2.5. Configuración avanzada de red



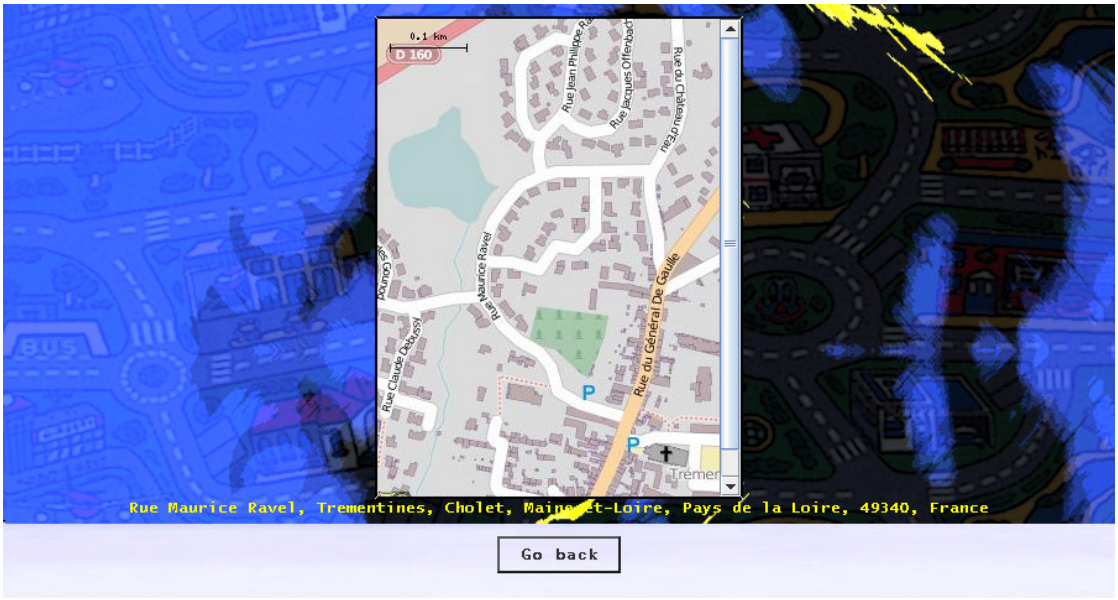
En esta pantalla de menú existen dos campos de texto y un desplegable. En el campo de texto superior, se puede modificar el puerto que se usará para

conectar con el servidor que se cree (esto es necesario ya que la aplicación está internamente separada en un servidor y un cliente, incluso para el caso de jugar un único jugador).
En el segundo campo de texto, se puede modificar el puerto en el que la aplicación escuchará las peticiones de unión de los jugadores que deseen unirse a la partida. En el desplegable aparecen las direcciones IP asociadas con cada adaptador de red habilitado, debiéndose elegir la que corresponde al adaptador que nos permite comunicarnos con la red en la que se hayan el resto de jugadores.

Nota: es importante comprobar que los puertos seleccionados están abiertos en el NAT/firewall en los modos TCP y UDP.

2.6. Configuración avanzada de mapas

En esta pantalla de menú, en la que se pueden descargar nuevos mapas que se quedarán almacenados en el directorio de juego para futuras partidas, existen dos secciones diferenciadas. La mitad derecha, en la cual se puede buscar una dirección, previsualizar el área resultante y finalmente añadirla como nuevo mapa, y la mitad izquierda, donde hay una tabla que contiene los mapas agregados y en la cual podemos previsualizar o eliminar dichos mapas.



Añadir un nuevo mapa al juego es un proceso muy sencillo.
Todo lo que se necesita es escribir la dirección deseada en el campo de texto «Address keywords» (palabras clave de la dirección) y pulsar en el botón «Search» (buscar). Esto realiza una búsqueda a través del servicio *Nominatim* de *OpenStreetMap*. Si hay resultados, éstos aparecerán en el desplegable «Search results» (resultados de la búsqueda).
Posteriormente, se debe seleccionar el tamaño deseado del mapa (usando el control deslizante «Map size» (tamaño del mapa).
En el campo de texto «Alias» se debe escribir un alias representativo para poder reconocer el mapa desde el desplegable de selección de mapa del menú de creación de la partida. Si se desea, es posible previsualizar el área entorno a la dirección seleccionada pulsando el botón «View map» (ver mapa).
Finalmente, se debe pulsar el botón «Add map» (añadir mapa) con el fin de añadir dicho mapa al juego.

Consejo: los resultados devueltos por la búsqueda de una dirección están limitados en número. Por este motivo, si se desea buscar una dirección con muchos posibles resultados, y no se obtienen los resultados deseados, debe procurar suministrarse una descripción más precisa de la dirección.

Consejo: cuanto más pequeño sea el tamaño del mapa mejor rendimiento del juego se obtendrá. Además, los mapas con muchos detalles pueden no ser capaces de ejecutarse en tamaños grandes o incluso medianos.

Nota: los mapas añadidos se guardan en la carpeta «Maps» del directorio de juego. Por cada mapa añadido se crea un fichero XML con el contenido descargado de OpenStreetMap, un fichero JPG con la imagen que se usará para la previsualización (también descargada de OpenStreetMap) y un fichero de texto que contiene datos necesarios para el uso de dicho mapa en el juego.

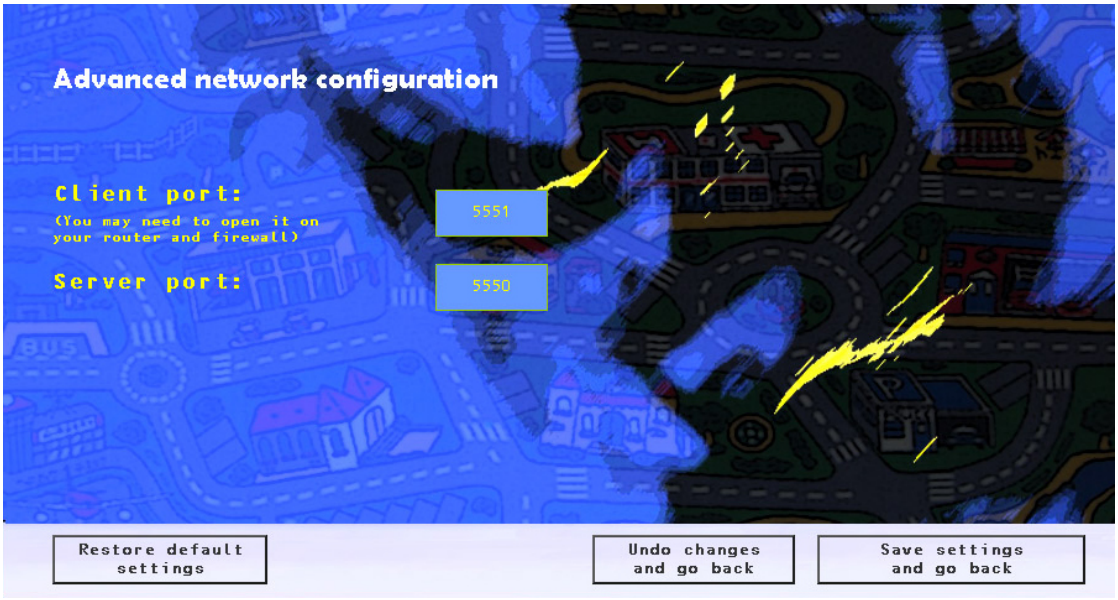
Nota: al previsualizar un mapa todavía no descargado se creará un fichero temporal con la imagen a mostrar. Dicho fichero se localizará en el directorio «temp» del directorio de juego, y será automáticamente eliminado al terminar la ejecución del juego.

2.7. Unirte a una partida existente (en red)

En esta pantalla se configuran los parámetros necesarios para la unión a una partida en red actualmente en juego.
Con este fin, se debe rellenar la dirección IP pública del servidor en el campo «server IP» (IP del servidor) , introducir un apodo para que el resto de jugadores nos reconozcan y pulsar en el botón «Join» (unirse).

Adicionalmente, en esta pantalla del menú, se pueden modificar los puertos que la aplicación usará para la conexión pulsando en el botón «Advanced network configuration» (configuración avanzada de red).

2.8. Configuración avanzada de red (en unión)



En esta pantalla del menú existen dos campos de texto.
En el primero se puede modificar el puerto que la aplicación usará para comunicarse con el servidor de la partida.
En el segundo, se debe introducir el puerto en el que el servidor de la partida está a la escucha de nuevas peticiones de unión a la partida.

Nota: es importante verificar que el puerto seleccionado en el primer campo de texto esté abierto en el NAT/firewall en los modos TCP y UDP.

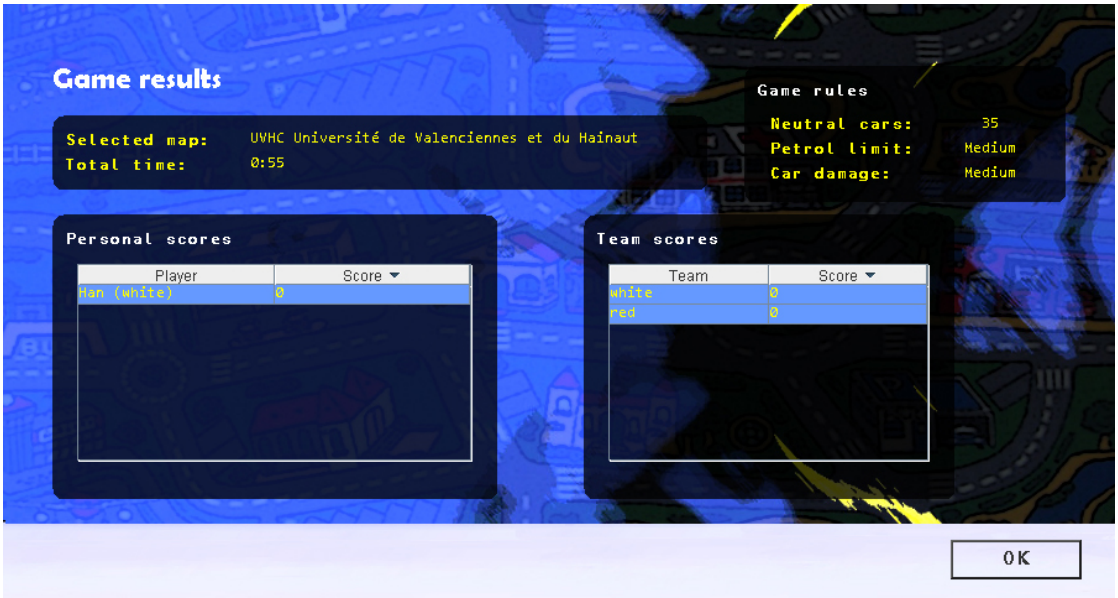
2.9. Opciones

En esta pantalla del menú se puede configurar el volumen de la música (control deslizante superior) y el volumen de los efectos de sonido (control deslizante inferior).
También desde aquí se puede ver y modificar los controles del juego (botón «Change controls») y seleccionar el directorio de juego donde serán almacenados los ficheros de configuración y los mapas descargados.

2.10. Ajustes de los controles

Desde esta pantalla se pueden modificar los controles del juego. Para ello no hay más que seleccionar el campo de texto correspondiente a la acción que se desea cambiar y pulsar la tecla que se desee usar.

2.11. Resumen de la partida



Esta pantalla aparece cuando se ha finalizado un partida, bien sea porque el anfitrión decide acabarla o porque se hayan completado o fallado los objetivos, y también cuando el jugador decide abandonar la partida en curso.
En ella se muestran detalles como el escenario y las reglas usadas, la duración de la partida y la puntuación de los jugadores desglosada por equipos en el caso de una partida competitiva.

2.12. Pantallas de error

En ciertas ocasiones puede aparecer una pantalla de error. En ella se mostrará al usuario la información referente al error ocurrido y le devolverá al menú principal.

Hay dos pantallas de error diferentes: error durante la conexión a la partida y error durante la ejecución de la partida.

Error al conectar a la partida

Esta pantalla de error aparece cuando no ha sido posible unirse a una partida. Esto puede estar causado porque los datos (dirección IP y puerto) sean erróneos o porque la partida ya haya finalizado.

Error durante la ejecución

Esta pantalla aparece cuando ha ocurrido un error en tiempo de ejecución que ha causado que la partida finalizase.

Contiene dos botones, uno para volver al menú principal y otro para habilitar la vista detallada del error, en la que se mostrará la traza del error ocurrido. En el caso de que haya ocurrido más de un error, aparecerán distribuidos en diferentes pestañas según cada hilo de ejecución.

3. El juego

En esta sección se mostrarán todos aquellos aspectos necesarios de conocer para poder participar en las partidas, como son los diferentes modos de juego, elementos de la partida (tipos de vehículos, tipos de terreno, etc.), controles del jugador y funcionalidades del menú de in-game.

3.1. Modos de juego

Se han desarrollado cinco modos de juego diferentes: *Capture the flag*, *Capture the red cars*, *Solve the tasks*, *Task endurance survival* y *Special parking mode*. Excepto el último, que solo está disponible en modo competitivo, los demás pueden ser jugados tanto en modo cooperativo como competitivo.

A excepción del cuarto modo, los demás funcionan mediante un sistema de rondas. Cuando comienza una ronda, se tiene unos pocos segundos para conducir libremente antes de que aparezcan los objetivos y los perseguidores. Pero aún sin enemigos hay que tener cuidado, ¡las colisiones con otros vehículos también producen daños!. Después de este tiempo de preparación, la ronda empieza y se muestra una cuenta atrás con el tiempo disponible para completar los objetivos. Cuando éstos hayan sido completados, los vehículos enemigos (si los había) desaparecerán, se rellenará la salud del jugador y la ronda finalizará.

Jugando en modo cooperativo las rondas son infinitas, se juega hasta que todos los jugadores mueran, pero en el modo cooperativo las rondas están limitadas por un valor que se puede modificar en el menú «Game rules configuration» (configuración de las reglas del juego).

Nota: para repostar en una gasolinera es preciso que el vehículo esté completamente detenido.

3.1.1. *Capture the flag*

En este modo de juego se deben capturar todas las banderas (normalmente cinco) mientras se huye de los vehículos enemigos que intentan destruirnos.

Hay que tener cuidado ya que se recibe un poco de daño en las colisiones contra los vehículos del tráfico, y mucho daño colisionando con los enemigos, los cuales tienen vida infinita.

3.1.2. *Capture the red cars*

En este modo de juego se deben capturar todos los vehículos enemigos, los cuales tratan de escapar del jugador.

El jugador únicamente recibe daño colisionando con vehículos del tráfico, mientras que los vehículos enemigos solo reciben daño con las colisiones con el jugador (no reciben daño colisionando contra otros vehículos).

3.1.3. *Solve the tasks*

En este modo de juego se deben completar diversas tareas antes de que el tiempo se agote o los vehículos enemigos destruyan al jugador.

Existen diferentes tipos de tareas:

- Conducir hasta una dirección o sitio de interés
- Aparcar en una plaza cercana a una dirección o sitio de interés (por cercana se entiende que sea una de las tres más próximas al objetivo).
- Llegar a pie a una dirección o sitio de interés, para lo cual previamente se ha tenido que aparcar el vehículo y así poder salir de él.

Nota: en las tareas que requieran llegar al objetivo a pie, es recomendable aparcar lo más cerca posible ya que el jugador es más lento y vulnerable mientras va a pie por lo que es una presa fácil para los vehículos enemigos.

3.1.4. *Task endurance survival*

Este modo de juego tiene una mecánica totalmente diferente del resto de modos de juego.
Los otros modos usan un sistema estricto de rondas -en los cuales si una ronda es completada satisfactoriamente los enemigos desaparecen y el jugador avanza a la siguiente ronda, y en caso contrario se pierde la partida-, sin embargo, es este modo de juego, los enemigos no desaparecen entre rondas ni tampoco se pierde la partida si no completas los objetivos de la ronda, sino que simplemente se pierde la oportunidad de ganar la recompensa de dicha ronda.

La partida se acaba cuando todos los equipos tienen una cantidad negativa de dinero (que sustituye a los puntos en este modo de juego), hasta ese momento todos los jugadores que sigan vivos podrán seguir jugando.

Si una ronda es completada satisfactoriamente, todos los jugadores todavía vivos recuperan un 50 % de vida y los muertos recuperan el 100 %. En ambos casos dichos jugadores perderán dinero, pero recuperar vida es más caro si el jugador está muerto que si solamente dañado.

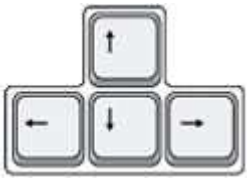
En cada ronda, el jugador debe completar una tarea, que puede ser lograda conduciendo, andando o aparcando en uno de los tres aparcamientos más cercanos (según se indique en la tarea), y la cual puede consistir en acudir a una dirección o a un sitio de interés de un determinado tipo (p.ej. ir a una farmacia).

3.1.5. *Parking special mode*

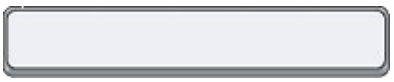
Este modo de juego, creado expresamente para facilitar la toma de estadísticas de aparcamiento, consiste en completar una serie de tareas de aparcamiento mediante un sistema de rondas, de forma muy similar al modo «Solve the task».
Las diferencias con ese modo radican en que en cada ronda hay un doble objetivo, que se debe cumplir de forma secuencial, en el que primero se requiere acudir a un determinado punto dado por una dirección o sitio de interés, y a continuación se requiere encontrar un aparcamiento situado a menos de 500m.

3.2. Controles

Los controles del juego son muy sencillos, ya que únicamente es preciso usar las flechas de dirección, la barra espaciadora y la tecla control.



Las flechas de dirección se usan para controlar el movimiento del vehículo:
La flecha superior (↑) acelera.
La flecha inferior (↓) frena y da marcha atrás.
La flecha izquierda (←) gira a la izquierda (rota el coche en sentido contrario a las agujas del reloj).
La flecha derecha (→) gira a la derecha (rota el coche en el sentido de las agujas del reloj).



La barra espaciadora se usa para crear una nube de humo.

La tecla Control es usada para aparcar y salir o entrar del vehículo.

La tecla Escape permite desplegar el menú in-game en el cual se pueden visualizar los controles, modificar el volumen y abandonar la partida.

3.3. Elementos

Durante las partidas el jugador encontrará diferentes tipos de vehículos, terrenos y objetivos. A continuación se explicarán los más relevantes de estos elementos del juego.

3.3.1. Vehículos

Hay varios tipos diferentes de vehículos:

- **Jugadores**: reconocibles por ser azules (con una banda central de color en el caso de necesitar diferenciar los diferentes equipos). Son los vehículos controlados por los jugadores.

- **Vehículos enemigos**: reconocibles por ser rojos. Son los coches controlados por el computador que, dependiendo del modo de juego, tratan de cazar al jugador o huir de él.
  También es posible reconocerlos por el sonido del motor que se puede escuchar cuando se acercan.

- **Ambulancias**: reconocibles por su forma y por su luz estroboscópica roja y azul. Además es posible escuchar su sirena cuando se acerca. Controladas por el computador. Aparece de forma aleatoria y desaparece cuando ha transcurrido un tiempo mínimo y no está a la vista de ningún jugador. Puede haber una única ambulancia sobre el escenario, siendo avisados los jugadores de su aparición y desaparición.

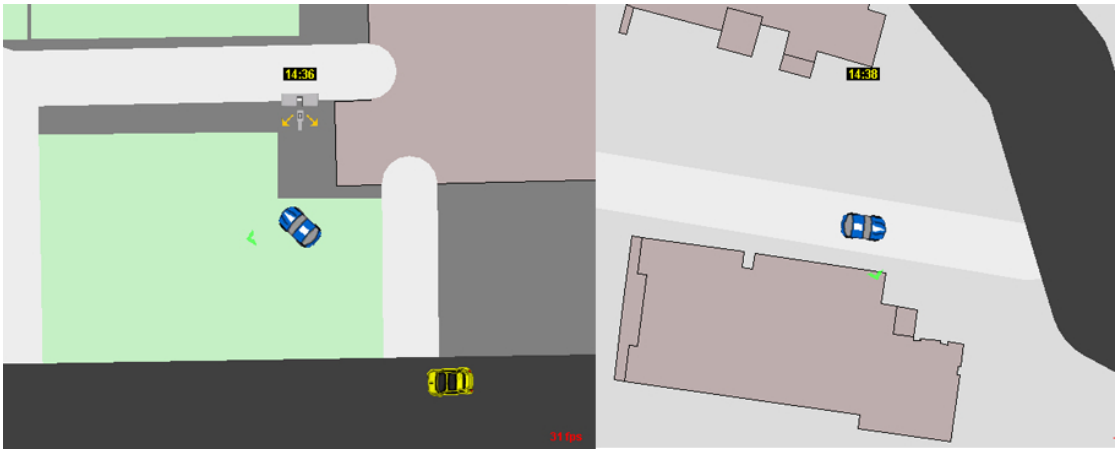
- **Vehículos del tráfico**: el resto de vehículos que no coinciden con las descripciones anteriores. Controlados por el computador.

3.3.2. Terrenos

Existen diferentes tipos de terrenos, algunos de los cuales son infranqueables para los vehículos:

- **Calzada pavimentada**: de color gris oscuro, es el terreno por el que circulan los vehículos del tráfico.
- **Calle peatonal**: de color blanco, solo los vehículos de los jugadores y los enemigos pueden circular en ella, sufriendo además una ralentización a la marcha.
- **Agua**: de color azul, es infranqueable.
- **Carril bici**: de color verde con línea de separación discontinua, únicamente los vehículos de los jugadores pueden circular por él, sufriendo una ralentización a la marcha.
- **Obras**: de color marrón, únicamente los vehículos de lo jugadores y los vehículos enemigos pueden atravesarlas, viéndose ralentizados.
- **Playa y césped**: de color marrón claro y verde respectivamente, solamente los vehículos de los jugadores y los enemigos pueden circular por ellos, sufriendo además una ralentización a la marcha.
- **Zonas urbanizadas y edificios**: son áreas de color gris claro y marrón bordeado en negro respectivamente, que son infranqueables para cualquier tipo de vehículo.

Nota: cuando el jugador circula a pie se le aplican las mismas reglas que si lo hiciera a bordo del vehículo.



3.3.3. Objetos y lugares de interés

Hay algunos objetos y lugares importantes que deben ser reconocidos ya que pueden ser utilizados como objetivo de la ronda según el modo de juego elegido.

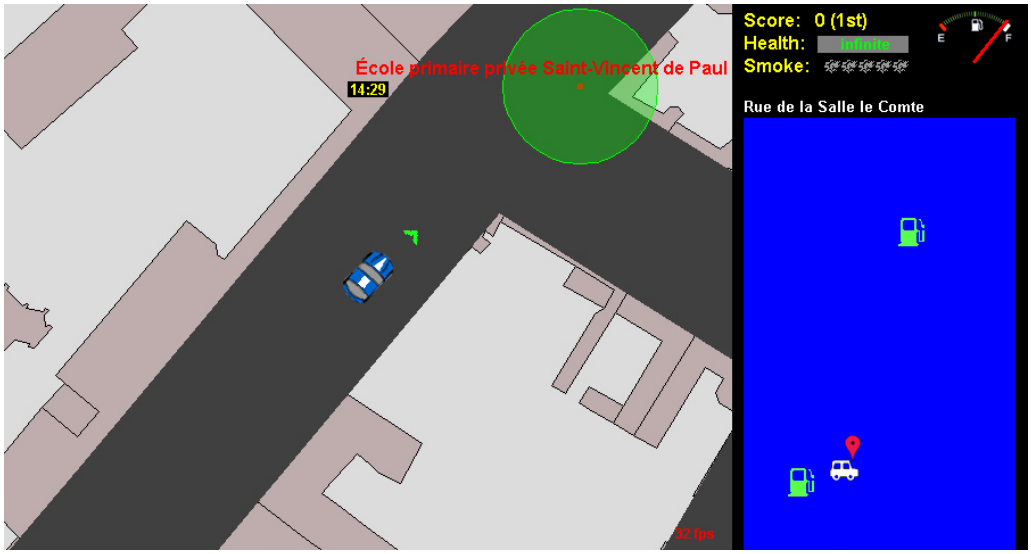
- **Banderas**: Una bandera que debe ser tomada para lograr puntos y rellenar las nubes de humo.

- **Gasolineras**: Un lugar donde se puede repostar combustible.

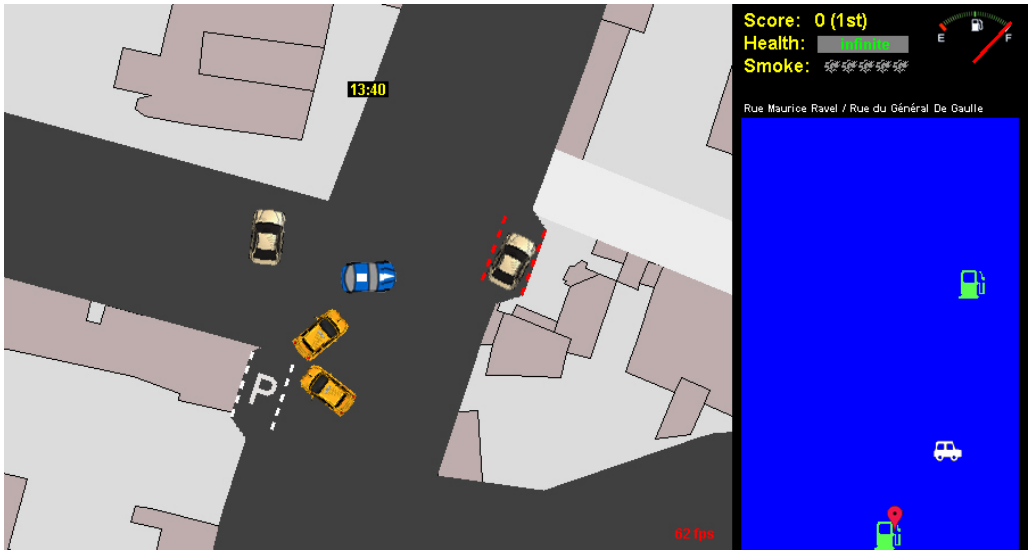
- **Sitios de interés**: lugares como:
  - cafés
  - restaurantes de comida rápida
  - bancos
  - farmacias
  - escuelas
  - tiendas

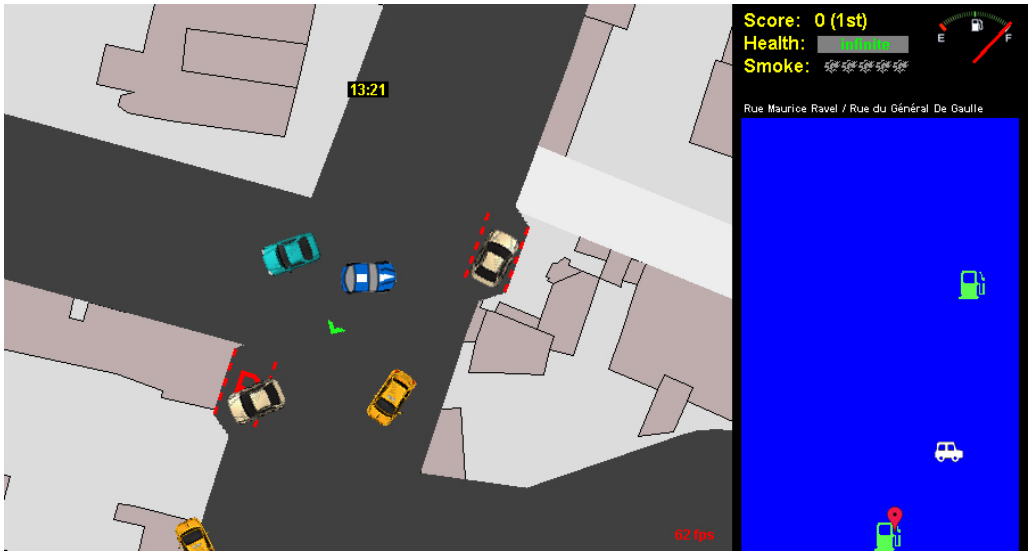
- hostales
- moteles
- hoteles
- museos

Son reconocibles por estar marcados con un círculo verde con el nombre pintado en letras rojas.



- **Plazas de aparcamiento**: son los lugares en los cuales, según el modo de juego, el jugador puede abandonar el vehículo para continuar a pie. También son los objetivos de diversos modos de juego.
  Para realizar un aparcamiento, el jugador debe aproximarse a una plaza vacía y una vez en su interior (cuando las líneas discontinuas cambien a color rojo) pulsar la tecla Control. Para abandonar el aparcamiento el procedimiento es el mismo.





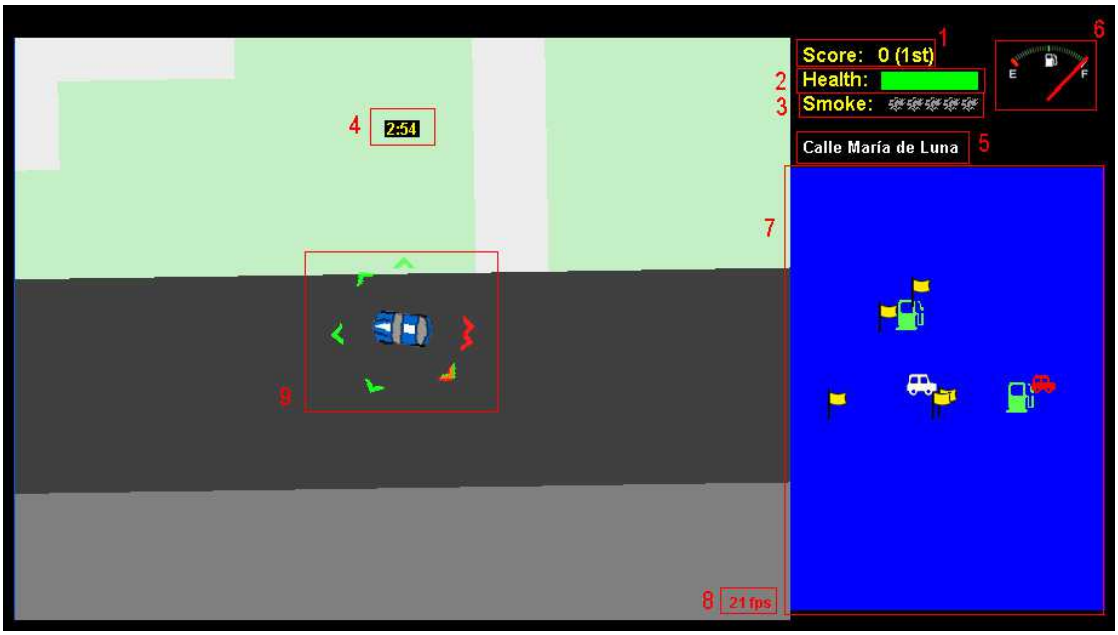
Nota: es importante tener en cuenta que el tiempo que una plaza permanece ocupada por un vehículo del tráfico varía (por defecto es entre 20 y 40 segundos), pero algunas plazas nunca se llegarán a desocupar.

3.3.4. Otros

- **Nube de humo**: se trata de un elemento temporal creado por los jugadores que, cuando es colisionada por un vehículo, le causa una reducción temporal de la velocidad así como una pérdida del control de la dirección del vehículo. Permanece únicamente durante diez segundos desde que es creada.

3.4. Interfaz gráfica de usuario

La interfaz gráfica de usuario muestra información útil durante la partida.



La información que contiene es la siguiente:

1) **Puntuación**: muestra la cantidad de puntos logrados hasta el momento.

2) **Indicador de salud**: muestra la cantidad de daño recibido por el vehículo. Su color varía de forma gradual desde el verde hasta el rojo según se reciban más daños.

3) **Nubes de humo disponibles**: muestra cuantas nubes de humo quedan disponibles. La cantidad máxima es cinco.

4) **Cuenta atrás**: muestra cuanto tiempo queda para lograr los objetivos de la ronda.

5) **Localización actual**: muestra el nombre de la calle en la que se encuentra el jugador. En el caso de estar situado sobre una intersección, se muestra el nombre de todas las calles de dicha intersección.

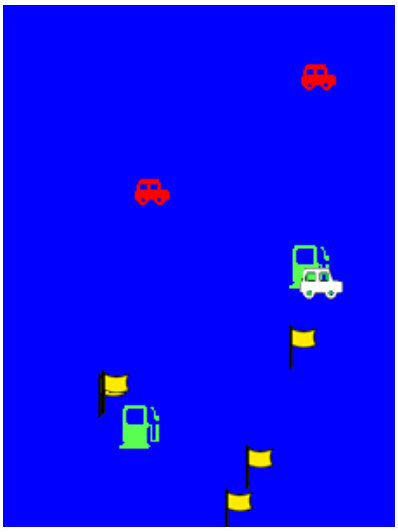
6) **Indicador de gasolina**: muestra de forma gráfica la cantidad de combustible restante.

7) **Radar**: muestra un mini-mapa en el que se representan los eventos VESPA y los lugares de interés.

8) **Indicador de FPS**: muestra el rendimiento actual de la aplicación medido en fotogramas por segundo.

9) **Flechas**: son flechas de colores que muestran en qué dirección respecto del jugador se encuentran los objetivos y los perseguidores (si los hay). Las flechas rojas indican la presencia de un perseguidor, mientras que las verdes indican los objetivos.

El radar muestra de forma gráfica los eventos considerados relevantes en el sistema VESPA, junto con otros lugares considerados de interés para la consecución de los objetivos de la ronda.



Cada icono representa un evento (salvo los que representan un icono siempre visible). Se pueden mostrar diferentes iconos refiriéndose al mismo incidente, ya que varios vehículos han podido detectarlo. Por lo tanto, no se recibirá un único icono en el punto donde el incidente ha tenido lugar, sino varios iconos en torno a dicho lugar.

Estos son los iconos siempre visibles:

 : nuestro jugador.

 : gasolinera.

 : objetivo (lugar de interés o dirección).

Estos son los iconos de eventos VESPA:

 : bandera.

 : accidente u obstáculo en la calzada.

 : servicio de emergencia.

 : otro jugador.

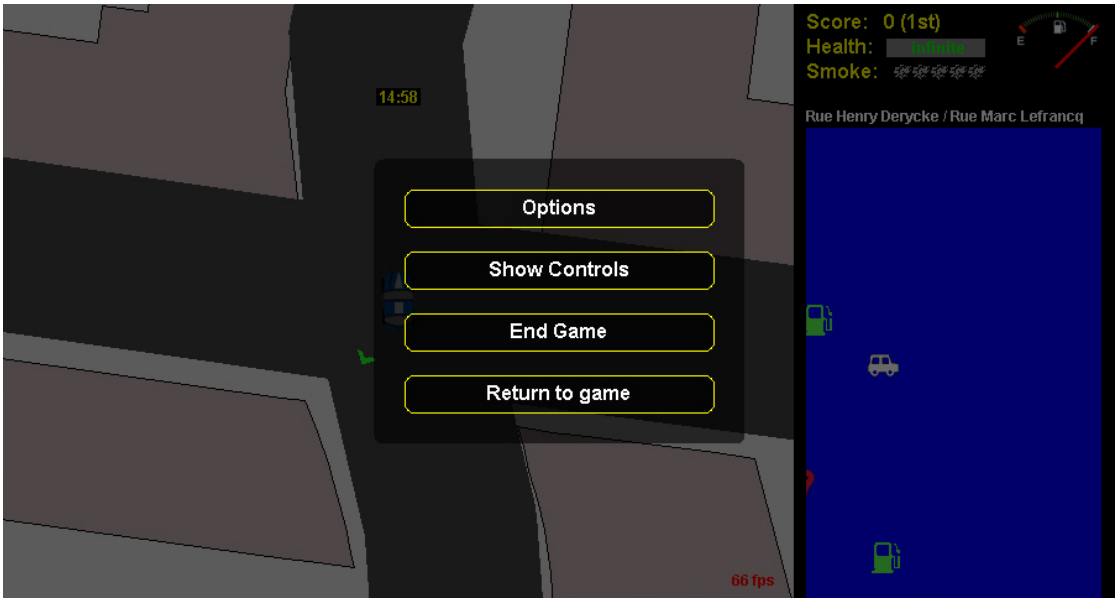
 : vehículo enemigo.

 : plaza de aparcamiento libre.

Al comienzo de cada ronda se muestra un recuadro de información que contiene el tipo de modo de juego, el número de ronda actual y el límite si lo hay, los objetivos, los perseguidores y el tiempo límite para completar la ronda.

3.5. Menú in-game

Durante el juego se puede acceder a un menú interno que permite realizar diversas acciones. La navegación por este menú únicamente puede realizarse mediante el uso del ratón.

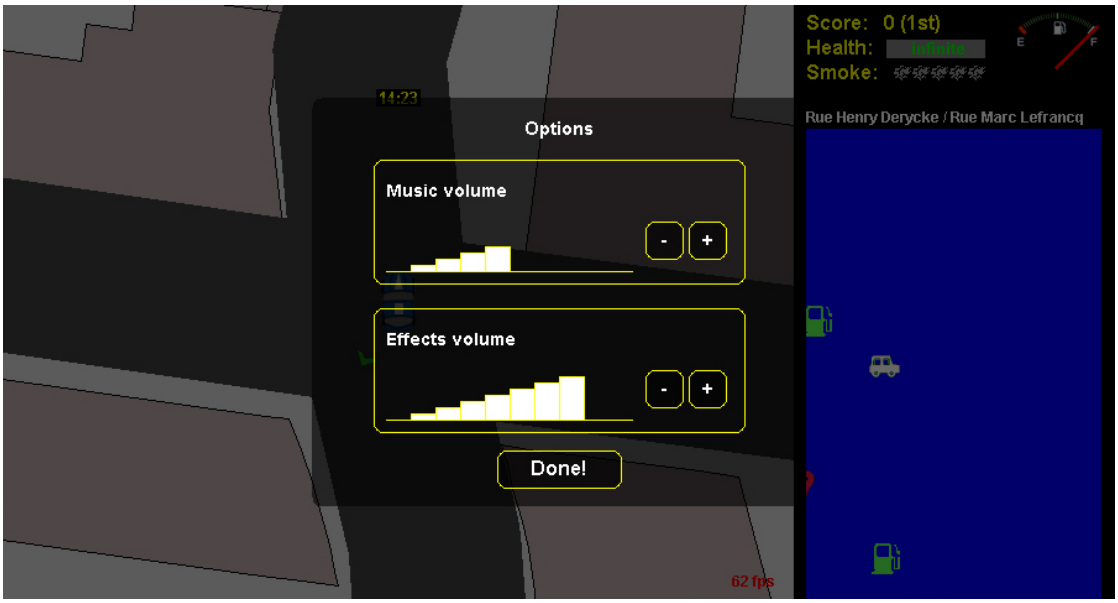


Estas acciones se categorizan en los siguientes apartados:

- Opciones (contiene el cambio de volumen de la música y los efectos)
- Mostrar controles
- Abandonar la partida
- Volver a la partida

A continuación se vera cada una de las diferentes opciones.

«Options» (Opciones)

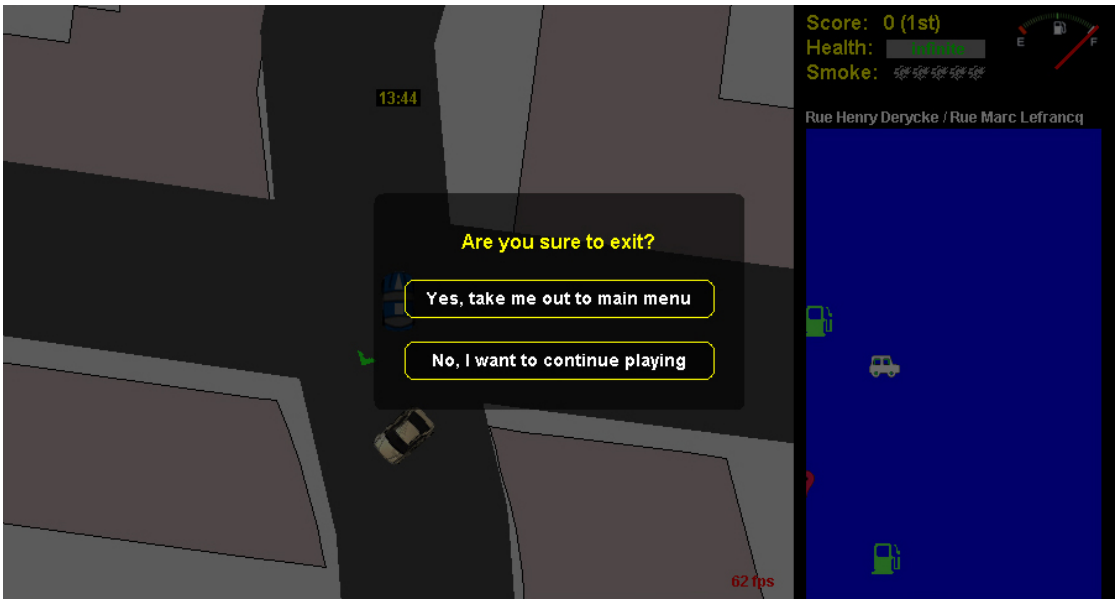


Se muestra una nueva pantalla en la que se puede modificar individualmente el volumen de los efectos de sonido y el volumen de la música.

«Show Controls» (Mostrar controles)

Se muestra una nueva pantalla en la que aparecen las diferentes acciones realizables por el jugador y las teclas actualmente asignadas a dichas acciones.

«End Game» (Abandonar la partida)



Aparece una pregunta de confirmación y en caso afirmativo se abandona la partida (y en el caso de ser el anfitrión se finaliza la partida de forma que el resto de jugadores también la abandonan).

«Return to game» (Volver a la partida)

En caso de seleccionar esta opción, el menú desaparece y se vuelve a tener control sobre el movimiento del jugador.

3.6. Pericia del jugador

La «pericia» del jugador es un valor que se almacena en la configuración del juego y que determina el nivel de habilidad del jugador.
Este valor se actualiza con cada nueva partida y es usado para impedir que jugadores noveles accedan a partidas en las que se recogen estadísticas y distorsionen los resultados.
Se mide en tiempo por tarea y se calcula dividiendo el tiempo de la partida por el número de rondas completadas.

$$\text{nivel de habilidad} = \frac{\text{tiempo total de la partida}}{\text{número de rondas completadas}}$$

Nota: hay que tener en cuenta que este cálculo únicamente puede realizarse de forma conjunta para todo el equipo, por lo que todos ganarán el mismo nivel de habilidad independientemente de su nivel real en la partida.

4. Características de ayuda a la explotación

En VANET-X se han desarrollado varias características con la función de facilitar la recopilación de los datos obtenidos referentes a las estrategias de gestión de información utilizadas.

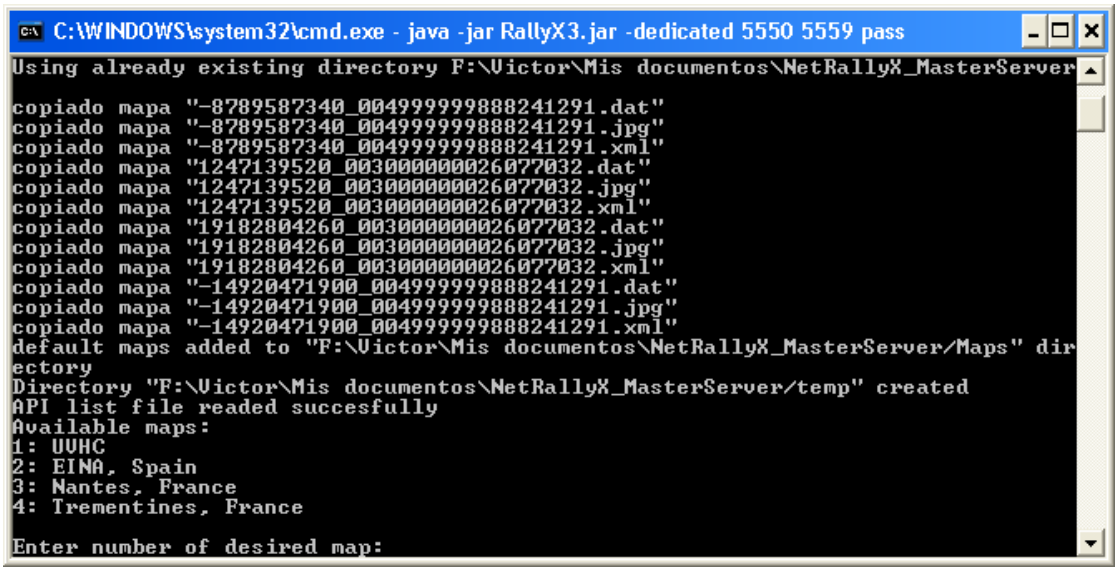
Una es el servidor dedicado, que permite tener un servidor en permanente funcionamiento en un computador de forma que los jugadores puedan unirse a dicho servidor.
Éste servidor dedicado tiene un mayor rendimiento que el servidor normal a costa de no incorporar el cliente necesario para que pueda jugarse desde la misma instancia del juego. Consiste en un proceso ligero a la escucha de nuevas peticiones de conexión de clientes, y sólo cuando éstas se produzcan se creará una instancia del servidor y se creará una nueva partida (suponiendo que no existiera ninguna).

Otro es el servidor de recogida de estadísticas, un proceso ligero, capaz de estar en permanente funcionamiento, que recopila los ficheros estadísticos enviados por los servidores y servidores dedicados al finalizar las partidas.

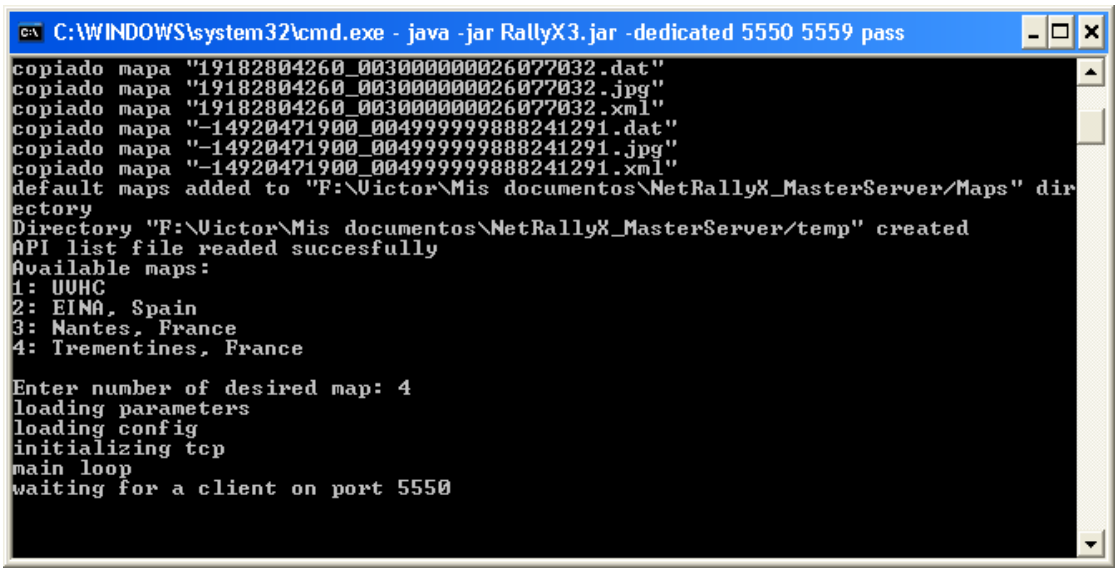
4.1. Creación del servidor dedicado

Para la creación de una instancia del servidor dedicado debe ejecutarse el juego con el siguiente comando: `java -jar RallyX3.jar -dedicated masterPort serverPort pass`, siendo `masterPort` el puerto en el que se escucharán las peticiones de conexión, `serverPort` el puerto en el que funcionará el juego y `pass` la contraseña que se requerirá a los terminales que traten de conectarse remotamente para gestionar el servidor (p.ej. `java -jar RallyX3.jar -dedicated 5550 5559 batmobile`).

Al iniciarse el servidor se requerirá introducir el directorio de juego deseado (o dejarlo en blanco para seleccionar el directorio por defecto) y también el mapa elegido para la partida.



A continuación el servidor se inicializará y quedará a la escucha de nuevos clientes.

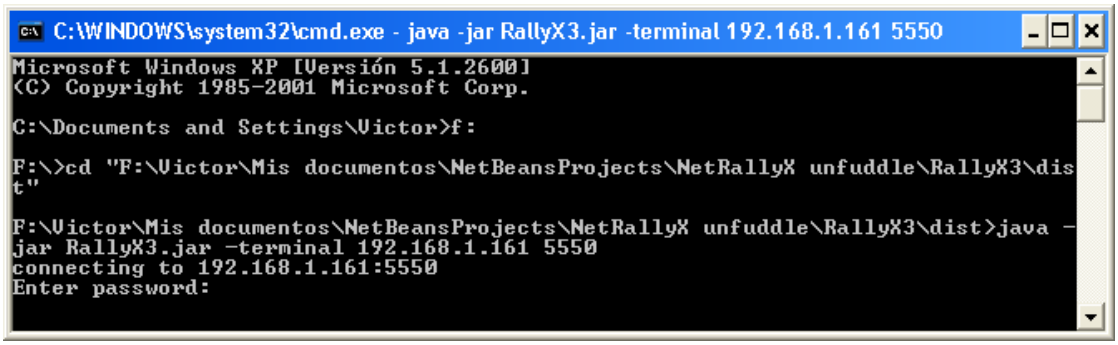


El servidor está preparado para reiniciarse de forma automática en caso de algún fallo.

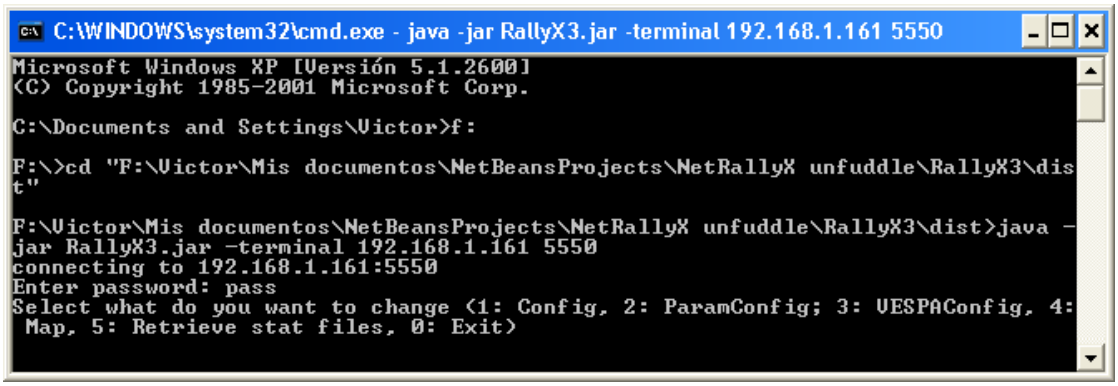
4.2. Acceso remoto al servidor dedicado mediante el terminal

Para la creación de una instancia del terminal debe ejecutarse el juego con el siguiente comando: `java -jar RallyX3.jar -terminal ip port`, teniendo como

argumentos la dirección IP y el puerto en el que se ubica el servidor dedicado con el que se desea conectar (p.ej. `java -jar RallyX3.jar -terminal 192.168.1.161 5550`).
Si existe un servidor dedicado en funcionamiento en dicha ubicación se requerirá introducir la contraseña establecida para dicho servidor.



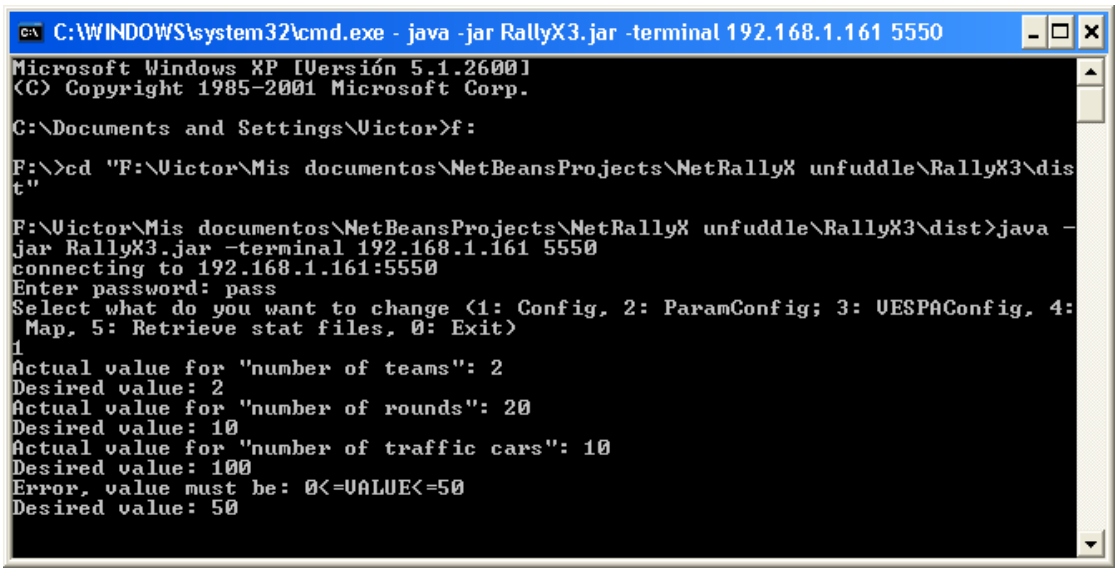
Si no existe el servidor o la contraseña es incorrecta, el proceso finalizará.



Si la contraseña es correcta, aparecerá el menú con las siguientes opciones:

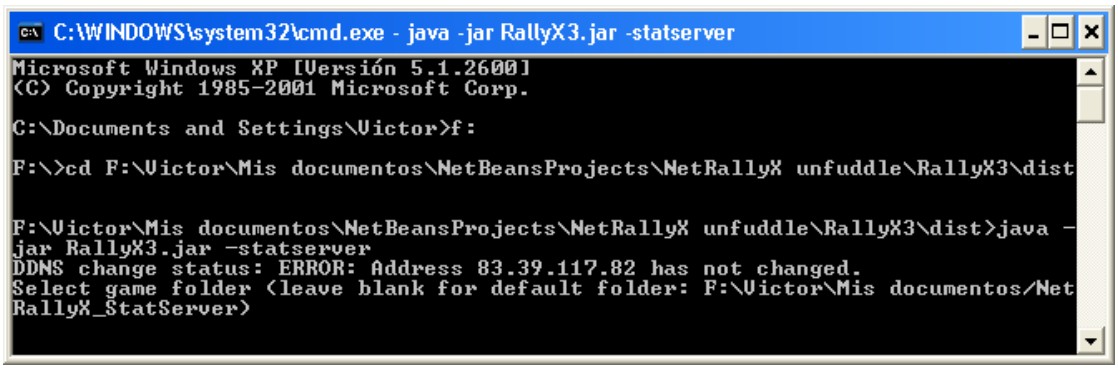
- modificar configuración del juego: modifica todas las opciones del juego que se configuran desde las pantallas de los menús, excepto las relativas a VESPA.
- modificar *ParamConfig.txt*: modifica los valores de dicho fichero.
- modificar configuración de VESPA: modifica la configuración del sistema VESPA.
- modificar mapa: se solicita la elección de un nuevo mapa para la partida.
- recuperar los ficheros estadísticos creados: crea en el directorio que se desee una copia de todos los ficheros de estadísticas existentes en el directorio de juego del servidor dedicado, dando la opción de borrar los originales al finalizar.

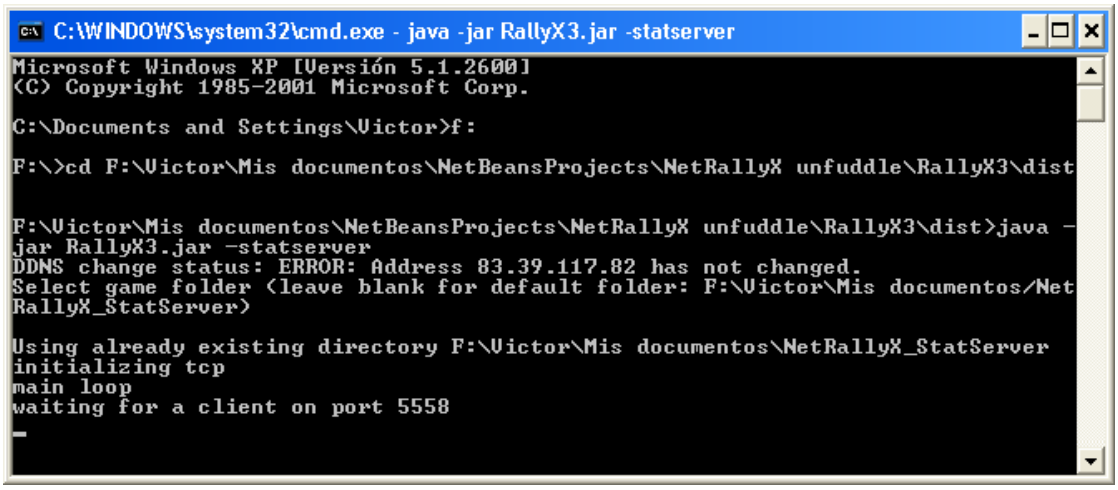
- salir: finaliza el proceso.



4.3. Creación del servidor de recogida de estadísticas

Para la creación de una instancia del servidor de recogida de estadísticas debe ejecutarse el juego con el siguiente comando: `java -jar RallyX3.jar -statserver`.





Nada más iniciarse, el proceso actualiza la dirección IP a la que apunta la *DDNS* (DNS dinámica) `nrxss.twilightparadox.com`, que es con la que conectan los servidores. Acto seguido, se requiere seleccionar el directorio donde se escribirán las estadísticas que se reciban, después de lo cual el proceso permanecerá a la espera de conexiones de los servidores.

5. Configuración técnica avanzada

Existen diversos parámetros del juego que pueden ser modificados de forma externa a los menús de la aplicación. Estos parámetros modifican aspectos clave en el funcionamiento del juego por lo que debe tomarse especial precaución en su modificación.

5.1. Modificación de los APIs usados para la obtención de los mapas

En el directorio de juego, en el interior de la carpeta «Maps», se encuentra un fichero «OSM_APIs.txt» que contiene un listado de URLs apuntando a diferentes APIs de OpenStreetMap que pueden ser usadas.
El juego, cuando trate de añadir un nuevo mapa, probará primero con la primera API de la lista y si ésta devuelve un mensaje de error se probará con la siguiente, y así sucesivamente. Si se llega al final de la lista sin lograr descargar el mapa, se hará uso de la API oficial, cuya URL se contiene dentro del código del juego.

5.2. Modificación del fichero de configuración *ParamConfig.txt*

En el directorio de juego se encuentra el fichero de texto «ParamConfig.txt», que contiene diversos parámetros del juego que pueden ser modificados por el usuario.

El fichero está estructurado de forma que por cada parámetro existe una primera línea con el nombre (que comienza por el carácter #) seguida de otra línea con el valor que toma.

La lista de los parámetros incluidos y sus descripciones es la siguiente:

Nombre	Valor por defecto	Descripción
radioMaxMapa	0.0050	Radio máximo del selector de tamaño al añadir un nuevo mapa. En unidades UTM.
radioPeqMapa	0.0030	Radio mínimo del selector de tamaño al añadir un nuevo mapa. En unidades UTM.
MaxBytesUDP	6000	Tamaño máximo de los mensajes de red (en bytes)
MaxExtrapolation	5	Número máximo de ciclos sobre los que se podrá aplicar la técnica de extrapolación cuando sea necesaria.
MaxInterpolation	2	Número de ciclos sobre los que se aplicará la técnica de interpolación. Aumentar este valor reducirá los «saltos de posición» del resto de vehículos a costa de aumentar la latencia del jugador.
NUM_PARKINGS_FIJOS	30	Número de plazas de aparcamiento «falsas» (están permanentemente ocupadas).
NUM_PARKINGS	10	Número de plazas de aparcamiento útiles (se ocupan y desocupan a lo largo de la partida).
NUM_FX	20	Número máximo de efectos visuales simultáneos (humos, mareos, etc.).

timeoutWaitingPlayers	60	Tiempo que puede estar el servidor esperando a que se conecte el primer jugador antes de abortar la creación de la partida.
timeoutWaitingPlayersDedicated	180	Tiempo que puede estar el servidor dedicado esperando a que se conecte el primer jugador antes de abortar la creación de la partida.
framesPropagacionExplosion	5	Número de ciclos en los que se envía el aviso de que ha habido una explosión al cliente. Este valor debe aumentarse en caso de altas latencias, ya que de lo contrario no se visualizarán las explosiones.
maxTC	50	Valor máximo para el campo «Number of neutral cars» de la pantalla «Game rules configuration».
maxRC	8	Valor máximo para el campo «Maximum number of enemy cars» de la pantalla «Game rules configuration».
maxNodos	10000	Número máximo de nodos de un mapa. No se permitirá descargar mapas con un número de nodos mayor a este valor.
WOLFSON (ON=1)	0	Aplicación del método WOLFSON en la implementación de VESPA.
maxPlayers	8	Número máximo de jugadores que se puedan unir a una partida.
cochesBuscandoParking	10	Vehículos del tráfico que están buscando aparcamiento (número constante en el tiempo).
TIEMPO_CAMBIO	20000	Tiempo mínimo que permanecerá un vehículo del tráfico aparcado. En milisegundos.
MARGEN_TIEMPO_CAMBIO	20000	Tiempo máximo (adicional sobre el mínimo) que permanecerá un vehículo del tráfico aparcado. En milisegundos.

RADIO_PARKING_VALIDO	500	Distancia máxima a la que se considera que un aparcamiento está cercano a un objetivo. En metros.
PARKING_PROTOCOL (0=Time,1=Distance,2=EP,3=None)	2	Tipo de protocolo de reserva (de aparcamiento) usado en la implementación de VESPA.
DISTANCIA_CERCA	700	Distancia máxima respecto del jugador a la que se recibirán datos del resto de elementos. Se debe aumentar si se quiere tener conocimiento de la posición del resto de vehículos en el radar cuando la opción «#debugRadar (ON=1)» está activada.
debugRadar (ON=1)	0	Muestra en el radar el resto de vehículos y las plazas de aparcamiento y su ocupación.
VESPA_StatisticsOn (ON=1)	1	Activa la recogida de datos estadísticos de VESPA.
Parking_Player_StatisticsOn (ON=1)	1	Activa la recogida de datos estadísticos de aparcamientos de jugadores.
Parking_Traffic_StatisticsOn (ON=1)	1	Activa la recogida de datos estadísticos de aparcamientos de vehículos del tráfico.
Game_StatisticsOn (ON=1)	1	Activa la recogida de datos estadísticos del resumen de la partida.
Stat server IP	nrxss.twilightparadox.com	*URL* o dirección IP de la ubicación del servidor estadístico al que se desea enviar las estadísticas recogidas.
Stat server port	5558	Puerto en el que escucha el servidor estadístico al que se desea enviar las estadísticas recogidas.
Cambio VESPA y protocolo aparcamiento automatico (0=desactivado)	0	Cambio automático entre diferentes implementaciones de VESPA (*VESPA+P*, *VESPA-P* y *sin VESPA*). Se debe indicar cada cuantas rondas se desea que se realice el cambio.
Allow untrained players ignoring statistics (ON=1)	0	Permite que jugadores sin habilidad suficiente se unan a partidas del modo de juego «Parking special mode».

Minimum skill level to qualify a player as trained	0.0083	Habilidad mínima exigida para que un jugador pueda unirse a partidas del modo de juego «Parking special mode».
Disable in-game transparencies (disable=1)	0	Deshabilita las transparencias prescindibles en los gráficos del juego.
Disable prediction of collisions (disable=1)	0	Deshabilita la predicción de las colisiones en el cliente (aumenta el rendimiento a costa de mayores errores de predicción)
Data management strategy selected	vespa	Implementación de Data Management Strategy deseada.

Nota: la modificación de los valores por defecto puede hacer el juego injugable o perjudicar su rendimiento.

Nota: Para volver a usar los valores por defecto se debe eliminar este archivo. Al volver a iniciar el juego el archivo se creará de nuevo con los valores iniciales.

6. Resolución de problemas

A continuación se describen los problemas más comunes y sus posibles soluciones.

6.1. Bajo rendimiento (framerate bajo)

Posibles causas:

- elección de un mapa demasiado extenso o detallado: debe prestarse atención al número de nodos que posee el mapa y descargar el mismo mapa con un menor tamaño.
- demasiados vehículos: debe probarse a reducir el número de vehículos del tráfico y de vehículos enemigos.
- problema con el pintado de transparencias: se puede comprobar pulsando la tecla `Q` durante el juego para desplegar la información de ronda, que es transparente, y observar si en ese momento el framerate disminuye drásticamente. En caso afirmativo debe deshabilitarse el pintado de transparencias mediante el parámetro `#Disable in-game transparencies (disable=1)` del fichero «ParamConfig.txt» (ver apartado 5.2).

6.2. Error de conexión en los primeros segundos de la partida

Debe asegurarse que los valores de los parámetros `#DISTANCIA_CERCA` y `#debug Radar (ON=1)` del fichero «ParamConfig.txt» coincida en todos los clientes y el servidor.

6.3. Corrupción u obsolescencia de los datos

El formato de los ficheros «ParamConfig.txt», «config» y «data» puede variar en diferentes versiones del juego. Cuando se actualice a una nueva versión, si no se proporciona ninguna herramienta para realizar la conversión, deben eliminarse dichos ficheros para que el juego cree las versiones por defecto al iniciarse.

7. Licencias

- **Developing Games In Java**
  Algunos de los algoritmos descritos en el libro han sido usados en esta aplicación.

OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- **JLayer**
  Available at `http://www.javazoom.net/javalayer/javalayer.html`.

  This program is free software; you can redistribute it and/or modify it under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Library General Public License for more details. You should have received a copy of the GNU Library General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

- **OpenSteer**
  Algunos de los algoritmos y estructuras de datos de esta librería han sido adaptados para el uso en esta aplicación.

  Available at `http://opensteer.sourceforge.net/`.
  OpenSteer is distributed as open source software in accordance with the MIT License. For more information see `http://opensource.org/licenses/mit-license.php`

- **Xerces**
  Available at `http://xerces.apache.org/#xerces2-j`.
  This component is licensed under Apache License Version 2.0. For more information see `http://www.apache.org/licenses/LICENSE-2.0`.

- **Guava (Google Core Libraries for Java)**
  Available at `https://code.google.com/p/guava-libraries/`.
  This component is licensed under Apache License Version 2.0. For more information see `http://www.apache.org/licenses/LICENSE-2.0`.

- **Música**:

  - *Methylchloroisothiazolinone (Instrumental Version)* album *Dirty Wings (Instrumental Version)* by *Josh Woodward (Instrumental Versions)*. Available under a Creative Commons Attribution 3.0 Unported licence.

For more information see http://creativecommons.org/licenses/by/3.0/. Available at http://www.jamendo.com/es/track/760400/methylchloroisothiazolinone-instrumental-version

- *Video game* album *Metamorphosis* by *Howarang Van K*. Available under a Creative Commons Attribution-ShareAlike 3.0 Unported licence. For more information see http://creativecommons.org/licenses/by-sa/3.0/. Available at http://www.jamendo.com/es/track/923426/video-game