UNIVERSIDAD DE  VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

DESARROLLO DE ESCENARIOS VIRTUALES PARA SIMULACIÓN DE CONDUCCIÓN

Autor:
D. Ramón Javier Corchero Floriano
Tutor:
D. David González Ortega
Valladolid, 30 de junio de 2016

TÍTULO:	**Desarrollo de escenarios virtuales para simulación de conducción**
AUTOR:	**D. Ramón J. Corchero Floriano**
TUTOR:	**D. David González Ortega**
DEPARTAMENTO:	**Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Dña. Míriam Antón Rodríguez**
VOCAL:	**D. Mario Martínez Zarzuela**
SECRETARIO:	**D. David González Ortega**
SUPLENTE:	**D. Francisco Javier Díaz Pernas**
SUPLENTE:	**D. José Fernando Díez Higuera**

FECHA:	**8 de julio de 2016**
CALIFICACIÓN:	

Resumen de TFG

Este trabajo de fin de grado ha consistido en desarrollar una plataforma centralizada de almacenamiento y procesamiento de datos que consta de dos partes claramente diferenciadas: una aplicación realizada con el motor gráfico Unity que permite simular un entorno de conducción semialeatorio en distintas situaciones (día, noche) y una aplicación web realizada usando el CMS (*Content Management System*) Drupal que permite gestionar un servicio REST para la comunicación remota con la aplicación para la simulación de conducción.

Se ha pretendido con este proyecto obtener un esqueleto desde el que proceder a integrar otros escenarios y con ello poder recabar un número de datos cada vez mayor sobre los comportamientos de conductores en diferentes situaciones.

Palabras clave

Simulador de conducción, REST, aplicación web, seguridad vial, Unity, Drupal.

Abstract

This project has aimed to create a basis for a centralized storage platform and data processing consisting in two distinct parts: an application developed with the graphics engine, Unity, to simulate a semi-random driving environment in different situations (day, night) and a web application developed with the CMS (Content Management System) Drupal for managing a REST service for remote communication with the driving simulator.

It has been intended to have a skeleton from which the integration of other scenarios can be fulfilled and thus it is possible to obtain an increasingly amount of data regarding the behavior of drivers in different situations.

Keywords

Driving simulator, REST, web application, road safety, Unity, Drupal.

Agradecimientos

"Al equipo de Unity por proporcionarnos a todos de forma gratuita uno de los motores más interesantes del mercado actual"

"A la increíble comunidad detrás de Drupal por su esfuerzo, paciencia y aporte a la mejora del framework"

Índice de contenidos

1. Introducción.

En este apartado se realizará una introducción a este documento, de manera que se tratarán las causas que motivan la realización de este trabajo, de igual forma que los objetivos que se esperan alcanzar mediante su realización. Se describirán de manera general las fases y métodos que se seguirán para la realización de este trabajo y los medios utilizados para ello, al igual que la previsión temporal y metodología organizativa para su realización.

Además, se comentará la estructura que seguirá el resto del documento, describiendo de forma resumida sobre qué tratará cada uno de los siguientes apartados.

1.1. Motivación.

Recurrir a la simulación de eventos de todo tipo está a la orden del día. Según avanza la tecnología es cada vez más viable crear entornos más y más realistas que permitan una fácil inmersión de los usuarios en dichas simulaciones, consiguiendo mejores resultados del uso de los mismos.

Estos simuladores pueden acelerar el proceso de adquisición de habilidades básicas. En concreto vamos a centrarnos en los simuladores de conducción, pero las ventajas son extrapolables a otros casos. Existen numerosas ventajas derivadas del uso de simuladores, entre las que podríamos destacar las siguientes:

- Selección de la simulación a realizar.
- Seguridad del conductor.
- Complejidad de las situaciones a simular.
- Recuperación y análisis de datos.

Los datos de siniestralidad han ido mejorando a lo largo de los últimos años, llegando a ser España en 2015 el quinto país de la Unión Europea con menor número de fallecidos por población (DGT, 2016).

Muchas empresas, como Mercedes-Benz, están invirtiendo enormes cantidades de dinero en los simuladores de conducción (Mercedes-Benz, 2015), ya que casi todas ellas coinciden en que las simulaciones permiten adaptar a los futuros conductores a muchas situaciones que suceden con poca frecuencia pero que suelen acabar en accidentes fatales, tales como desprendimientos, encontrarse con un accidente en medio de la carretera antes de que lleguen los servicios de asistencia, la aparición de animales en medio de la carretera, etc.

Por lo tanto, el desarrollo de entornos de simulación es importante tanto por lo que aportan a los conductores como por el hecho de que permiten recuperar datos que pueden ser analizados para mejorar la seguridad en vehículos o para ver carencias en las enseñanzas o habilidades de las personas que puedan influir en su futura conducción.

1.2. Objetivos generales.

El objetivo de este Trabajo Fin de Grado es crear la base de un sistema servidor/cliente consistente en dos secciones claramente diferenciadas:

- Una aplicación realizada con el motor gráfico Unity que consiste en un simulador con cinco escenarios distintos. Esta aplicación permite al conductor realizar simulaciones y le proporciona varias opciones mientras las realiza, además de poder almacenar sus datos de forma remota en un servidor.

- Un módulo de Drupal que permite, en combinación con otros módulos desarrollados por la comunidad de Drupal, crear una API (*Application Program Interface*) REST (*Representational State Transfer*) que permite recibir datos en formato JSON (*JavaScript Object Notation*) desde la aplicación Unity, almacenarlos y procesarlos.

En la actualidad todo el desarrollo de software va orientado hacia la movilidad y a utilizar las aplicaciones desde cualquier lugar. Con este trabajo se pretende obtener un sistema genérico que permita el acceso a datos del simulador desde cualquier lugar. Más adelante presentaremos varios de los usos que podría tener esta plataforma tanto de cara a la educación como a la empresa.

1.3. Fases y métodos.

Las fases que seguiremos en este proyecto son:

- Breve estudio sobre seguridad vial. Existen a día de hoy estudios muchos más amplios de lo que se podría llegar a explicar en este trabajo y nos remitiremos a ellos. Sin embargo, se darán ciertas nociones de conducción que pueden resultar de utilidad por ser poco conocidas o más novedosas de lo habitual, al igual que ciertos consejos relativos a la conducción eficiente y a evitar la ejecución de infracciones comunes, como realizar una glorieta incorrectamente o no usar los intermitentes de forma adecuada.
- Referenciar algunos de los simuladores del mercado actual, tanto orientado a la educación como al mundo del ocio.
- Un análisis de las tecnologías actuales más conocidas y utilizadas a día de hoy para el desarrollo de gráficos, para la creación de escenarios virtuales.
- Un análisis de algunas herramientas web/*frameworks* para el desarrollo de plataformas y aplicaciones web.
- Elección y estudio del funcionamiento de las herramientas a utilizar finalmente.
- Desarrollo de la aplicación, donde crearemos diferentes escenarios para las simulaciones con los scripts programados en el lenguaje C#.
- Desarrollo del módulo para la plataforma web de gestión de datos de la aplicación anterior.
- Se realizarán algunas pruebas para observar los resultados obtenidos de la conducción de diferentes usuarios. Se mostrarán los datos obtenidos tal y como se podrán visualizar y analizar en la plataforma web.
- Líneas futuras sobre el proyecto separadas tanto en posibilidades para el simulador como en posibilidades para la plataforma web.
- Elaboración de un presupuesto económico básico del coste del trabajo realizado.

1.4. Elementos empleados.

Se han utilizado los siguientes elementos para la realización de este proyecto.

En primer lugar, dos ordenadores portátiles distintos para comprobar la respuesta del simulador con las siguientes características:

Ordenador ASUS

- Procesador: Intel Core I7-6500 CPU @ 2.0GHz 2.60GHz
- Memoria RAM: 20 GB
- Sistema operativo: Windows 10 (64 bits)
- Tarjeta gráfica: Nvidia GeForce 960M

Ordenador Toshiba

- Procesador: Intel Core I3
- Memoria RAM: 4 GB
- Sistema operativo: Windows 7 (64 bits)
- Tarjeta gráfica: Nvidia GeForce 650M

Cómo periféricos de entrada se han utilizado:

- Por un lado, el dispositivo Logitech G27, el cual está formado por volante (con levas incluidas), pedales y palanca de cambios.
- Además, para pruebas más básicas se ha utilizado el controlador Xbox 360 Wireless, uno de los mandos más utilizados del mercado actual.
- También se han realizado simulaciones usando el teclado como método de entrada.

1.5. Estructura de la memoria.

Esta memoria se va a componer de varios apartados que tratarán de exponer tanto información general para poner al lector en situación dentro del problema a solucionar, como de explicarle cómo se ha pretendido solucionar y cuáles han sido las razones para hacerlo de la forma en que se ha hecho.

En este apartado se está viendo una introducción acerca de lo que constará este trabajo y las partes que se irán tratando a lo largo del resto del documento.

En el segundo apartado se hablará acerca de la seguridad vial, donde se tratarán definiciones y principios relacionados con esta, desde el concepto de tráfico hasta el de seguridad vial. No se entrará a realizar una extensiva introducción al respecto, pues existen un enorme número de proyectos mucho más profundos al respecto a los que referirse. Sin embargo, sí que se darán algunas nociones genéricas de algunas normas que a día de hoy siguen produciendo confusión, en concreto las normas de conducción dentro de las glorietas. Para finalizar este apartado se hará un estudio acerca de cómo pueden contribuir los simuladores de conducción a una mejora en la seguridad vial.

El tercer apartado tratará acerca de las diferentes tecnologías disponibles que podrían ser utilizadas para realizar un simulador de conducción, junto con las tecnologías web viables para la creación de un servidor REST que permita recibir y almacenar datos. Para el simulador será necesario un motor gráfico para hacer la física y programar la lógica. Mientras tanto, para la plataforma web será necesario tomar una decisión entre el enorme abanico de lenguajes de programación y tecnologías disponibles a día de hoy. A continuación, se hará una breve comparativa entre las tecnologías vistas y se hará un breve resumen de las razones que llevaron a la elección de tecnologías para el desarrollo final de las aplicaciones del proyecto.

En el apartado cuatro se va a presentar cómo se ha realizado el desarrollo de la aplicación de simulación con Unity. Se explicarán herramientas propias del motor y cómo se han utilizado. También se explicarán algunas de las funcionalidades más destacables del simulador, entre ellas el sistema de indicaciones, el sistema de infracciones y almacenamiento de datos o la implementación del sistema de peatones.

En el apartado cinco se explicará cómo se puede montar una plataforma Drupal y cómo trabajar con esta tecnología mediante la creación de módulos. En concreto se va a presentar el desarrollo de un módulo que proporciona un servicio REST para el almacenamiento remoto de datos y un análisis básico de los datos obtenidos.

En el apartado seis se van exponer algunas de las conclusiones obtenidas, pero principalmente se van a explicar algunos posibles desarrollos que podrían aumentar y mejorar la envergadura del proyecto, de cara a obtener una aplicación más completa tanto a nivel técnico como a nivel funcional.

El apartado final consiste en un desglose del coste económico del proyecto, teniendo en cuenta tiempos de amortización, de materiales y horas de trabajo.

1.6. Distribución temporal del desarrollo del proyecto. Metodología Kanban.

Debido a que la realización de este proyecto ha habido que compaginarla con un trabajo a tiempo de completo como programador informático, necesitaba una metodología que me permitiera dedicar el tiempo necesario para la ejecución del TFG sin establecer unos horarios fijos. Para ello se ha intentado seguir una metodología de desarrollo ágil de software.

Las metodologías ágiles son una serie de técnicas para la gestión de proyectos que han surgido como contraposición a los métodos clásicos de gestión y desarrollo como los modelos en cascada.

Todas las metodologías que se consideran ágiles cumplen con el manifiesto ágil establecido por Beck et al. (2001), que no es más que una serie de principios que se agrupan en 4 valores:

- Los individuos y su interacción, por encima de los procesos y las herramientas.
- El software que funciona, frente a la documentación exhaustiva.
- La colaboración con el cliente, por encima de la negociación contractual.
- La respuesta al cambio, por encima del seguimiento de un plan.

En un equipo de una sola persona no tienen mucho sentido estos valores específicamente. Sin embargo, ser ágil es una habilidad que consiste en ser capaz de ser flexible, de responder a los cambios con rapidez y poder adaptar tus tiempos y esfuerzo y orientarlos de la forma correcta. Y eso es aplicable en todos los aspectos de la vida.

En concreto se ha intentado usar *Kanban* como metodología ágil para el desarrollo de este proyecto. El origen de la metodología *Kanban* debemos buscarlo en los procesos de producción "*just-in-time*" (JIT) ideados por Toyota, en los que se utilizaban tarjetas para identificar necesidades de material en la cadena de producción (Anderson, 2010).

Actualmente, el término *Kanban* ha pasado a formar parte de las llamadas metodologías ágiles, cuyo objetivo es gestionar de manera general cómo se van completando las tareas. Como se ha mencionado, *Kanban* surge en Japón y es una palabra que significa "tarjetas visuales", donde *Kan* es "visual", y *Ban* corresponde a "tarjeta".

La principal ventaja de esta metodología es que es muy fácil de entender, utilizar y actualizar. Además, destaca por ser una técnica de gestión de las tareas muy visual, que permite recuperar información sobre el estado de los proyectos de un vistazo, así como también pautar el desarrollo del trabajo de manera efectiva. La parte visual de *Kanban* es el denominado "tablero *Kanban"*, que consiste en una pizarra dividida en columnas que representa fases del desarrollo. En ellas se sitúan las tarjetas que representan tareas que hay que realizar.

En este trabajo se ha recurrido a la herramienta *Trello* para representar la pizarra de *Kanban* (Figura 1.1).

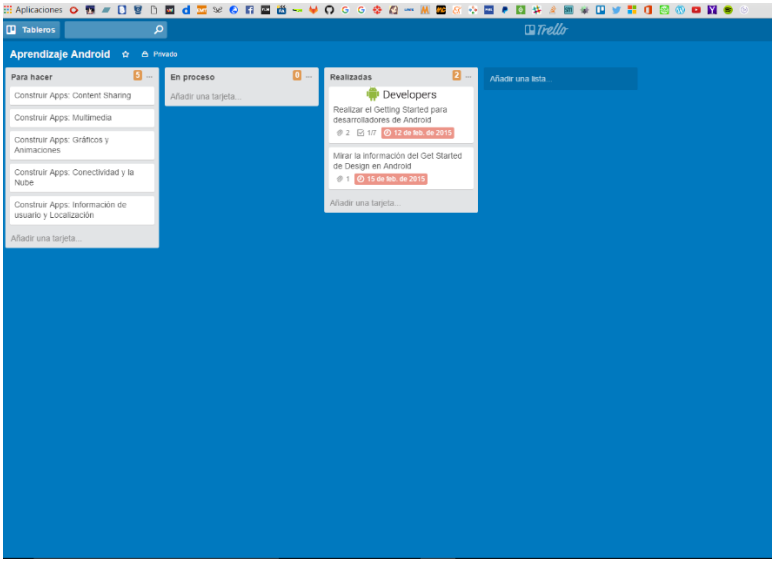


Figura 1.1 – Ejemplo virtual de Tablero Kanban (Trello, 2016).

En *Kanban* al tiempo que tarda una tarjeta en recorrer todos los estados se le denomina *Cycle Time*. Este tiempo permite establecer el tiempo de ejecución medio por tarea. Además del *Cycle Time*, también existe el *Lead Time* que es el tiempo desde que se realiza una petición hasta que se entrega al cliente (Kanban Tool, 2015). En nuestro caso ambos términos se pueden usar indistintamente debido a que somos nuestros propios clientes, pero dependiendo del proyecto puede ser conveniente mantenerlos separados y por eso es importante diferenciarlos.

En nuestro caso se estimaba inicialmente un *Lead Time* de unas 2.5 horas, lo que permitiría el desarrollo de hasta 60 tareas en un TFG de 150 horas. En el apartado séptimo se verá que nuestra estimación se había quedado corta.

La ventaja de utilizar una metodología ágil es que nos hace entender que diseñar y prever de antemano cambios es imposible. Comprender que los cambios suceden lo queramos o no, ya sea por peticiones del cliente, por mala comunicación o por modificaciones tecnológicas imprevistas, nos permite ser capaces de adaptarnos y hacerlos frente, intentando entregar siempre software funcional que permita a un cliente darnos una retrospectiva del trabajo realizado y de esta manera realizar el software que mejor se adapte a lo que quería realmente.

2. Seguridad vial y simuladores de conducción.

En este apartado se tratarán ciertas cuestiones sobre seguridad vial. Además, se comentarán algunos datos relativos a la forma de conducir por las glorietas, donde es fácil cometer una serie de infracciones que un simulador podría ayudar a descubrir a tiempo para evitar que un conductor las cometiera en el futuro. También se nombrarán brevemente algunos de los simuladores de conducción existentes en la actualidad, y se estudiará cómo estos pueden ser útiles para mejorar la seguridad vial.

A continuación, se hablará sobre el concepto de seguridad vial y los accidentes y cuáles son las causas de que se produzcan. Además, se tratará el tema de la conducción eficiente.

2.1. Definiciones.

El tráfico puede definirse como “el desplazamiento de personas, animales y vehículos por las carreteras, calles y caminos”. Los factores que intervienen en el tráfico son: el factor humano, el vehículo y la vía y su entorno, de forma interrelacionada. Concretamente el factor humano interviene en la gran mayoría de los accidentes, y en ello influyen la psicología, la pedagogía, la medicina, las normas, la técnica de conducción, etc.

Los principios fundamentales que deben regular el tráfico son: la responsabilidad, la confianza en la normalidad del tráfico, la conducción defensiva, la seguridad en la conducción, la señalización o conducción dirigida y la integridad personal.

La seguridad vial podría ser definida como la no producción de accidentes. La seguridad vial tiene como finalidad primordial la seguridad, aunque parezca redundante. Esa seguridad no es solo a nivel físico de las personas sino también psicológico, además de la seguridad del entorno, proponiendo métodos para evitar la contaminación y conducir eficientemente reduciendo el uso de carburantes.

2.2. Accidentes.

El accidente puede ser definido como “cualquier evento como resultado del cual el vehículo queda de manera anormal, dentro o fuera de la carretera, o que produce lesiones en las personas y daño a terceros”. La Organización Mundial de la Salud (OMS) estima que más de un millón de personas mueren al año a causa de accidentes de tráfico en todo el mundo (Organización Mundial de la Salud, 2015).

2.2.1. Tipos de accidente.

Solo puede hablarse de accidente involuntario cuando se alude a la parte pasiva de la acción, es decir, a quien se involucra en un siniestro de tráfico sin poder evitarlo. Porque, salvo la intervención de la naturaleza, o a procesos orgánicos fisiológicos del ser humano, gran parte de los siniestros son prevenibles y evitables. Un porcentaje menor de ellos se debe a fallas de fabricación de vehículos, lo cual no excluye atribuirles un "error humano consciente". Los accidentes de tráfico se catalogan según distintas escalas de gravedad, siendo el tipo más grave aquel en el que existen víctimas mortales, bajando la escala de gravedad cuando hay heridos graves, heridos leves, y el que origina daños materiales a los vehículos afectados. También pueden clasificarse los

accidentes en función del número y tipo de vehículos involucrados. Según este criterio podríamos clasificar los accidentes en las siguientes categorías:

- Salidas de la vía, vuelco y pérdida de control.
- Atropellos.
- Colisiones entre dos vehículos.
- Colisiones múltiples o en cadena.

2.2.2. Causas.

Siempre hay una causa que desencadena un accidente vial. Este se puede agravar de forma considerable si por él resultan afectadas otras personas, además de la persona que lo desencadena.

Un accidente también puede verse agravado si no se ha hecho uso adecuado de los medios preventivos que, aunque no lo eviten, podrían reducir su gravedad. Por ejemplo, no llevar puesto o bien ajustado el cinturón de seguridad o no llevar puesto el casco si se conduce un ciclomotor.

Figura 2.1. – Triángulo de seguridad vial: factor humano, vehículo y vía.

La figura 2.1 representa el conocido como triángulo de seguridad vial, que muestra los factores claves en la aparición de los accidentes de tráfico:

Factor humano: Los factores humanos son la causa del mayor porcentaje de accidentes de tráfico. Ejemplos de causas de accidentes debidas al factor humano:

- Conducir bajo los efectos del alcohol, medicinas y estupefacientes es una de las mayores causas de accidentes de tráfico según la Organización Mundial de la Salud (Organización Mundial de la Salud, 2015).
- Realizar maniobras imprudentes y de omisión por parte del conductor.
- Efectuar adelantamientos en lugares prohibidos (Choque frontal muy grave).

- Desobedecer las señales de tráfico, por ejemplo, pasar un semáforo con luz roja o no detenerse frente a una señal de stop.
- Circular por el carril contrario (en una curva o en un cambio de rasante).
- Conducir con exceso de velocidad (que puede derivar en vuelcos, salida del automóvil de la carretera o derrapes).
- Usar de forma incorrecta las luces del vehículo, especialmente durante la noche.
- Condiciones no aptas de salud física y mental/emocional del conductor o del peatón (ceguera, daltonismo, sordera, etc.).
- Peatones que cruzan por lugares de riesgo con la intención de dañarse a sí mismos.
- Inexperiencia del conductor al volante.
- Fatiga del conductor como producto de la apnea o falta de sueño.

Factor mecánico:

- Vehículo en un estado inadecuado para su correcto uso (sistemas averiados de frenos, dirección, aceleración o suspensión).
- Mantenimiento inadecuado del vehículo.

Factor climatológico y otros:

- Niebla, humedad, derrumbes, zonas inestables, hundimientos.
- Semáforos con un funcionamiento incorrecto o deficiente.
- Condiciones inadecuadas de la vía para su correcto uso (grietas, huecos, obstáculos sin señalización).

2.2.3. Consecuencias.

Los accidentes en la carretera han ocasionado numerosos costes sociales a lo largo de la historia, no solo en pérdida de vidas humanas sino también en forma de lesiones temporales o permanentes a personas involucradas en los mismos. Además, frecuentemente las lesiones permanentes acarrean fuertes costes económicos tanto al estado, como a las compañías aseguradoras y a los individuos que los padecen. Se estima que, a principios del siglo XXI, cada año se producen entre 1,5 y 2 millones de muertos por causa de accidentes de tráfico, y en muchos países desarrollados constituye la principal causa de muerte entre los menores de 25 años (Organización Mundial de la Salud, 2015).

La siniestralidad o peligrosidad tiene que ver con la probabilidad de que ocurran accidentes en un determinado tramo de carretera, un determinado tipo de vehículo o un grupo determinado de conductores.

Por otra parte, la vulnerabilidad tiene que ver con la posible ocurrencia de daños en caso de ocurrencia de accidente.

Los elementos de seguridad pasiva y seguridad activa de los vehículos modernos se han previsto para disminuir la vulnerabilidad de las personas involucradas en accidentes.

En la mayor parte de países desarrollados se ha observado, en gran parte por la mejora de la seguridad de los vehículos, que el riesgo mortal por accidente ha disminuido, es decir, en caso de accidente se ha disminuido notablemente la probabilidad de muerte. Sin embargo, aunque ha disminuido la mortalidad, la proporción de lesionados (heridos que no fallecieron) ha aumentado.

La mayor parte de fallecimientos en accidentes de tráfico están asociadas a contusiones graves, tales como pueden ser traumatismos craneoencefálicos, roturas torácicas y perforación o daño grave en órganos internos. Entre los heridos, además de si son graves (riesgo de muerte) o leves (sin riesgo de muerte), debe distinguirse también entre heridos con lesiones permanentes y heridos con lesiones pasajeras.

2.2.4. Accidentes en España.

Según la Dirección General de Tráfico (DGT) durante el año 2015, en las vías interurbanas españolas se han producido 1.018 accidentes con víctimas mortales en los que han fallecido 1.126 personas y 4.843 han necesitado hospitalización (DGT, 2016).

La cifra de fallecidos representa el mínimo histórico desde 1960, primer año en el que se tienen estadísticas, cuando hubo 1.300 muertos, con un escenario de absolutamente distinto (en 1960 había un millón de vehículos y en 2015 el parque automovilístico sobrepasa los 31 millones).

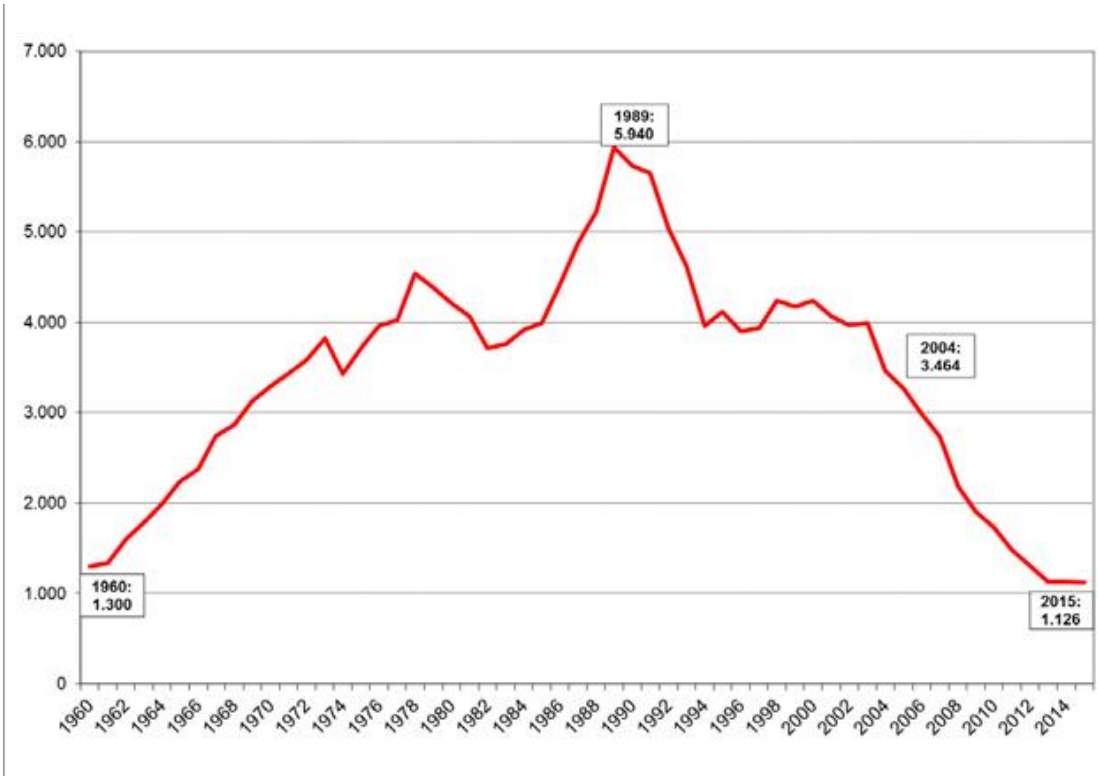


Figura 2.2 - Evolución del número de fallecidos en vías interurbanas 1960 – 2015 (DGT, 2016).

De los datos de la figura 2.2 se puede concluir que las víctimas por accidentes de tráfico se han reducido por duodécimo año consecutivo, descenso reflejado en el promedio de víctimas mortales por día que ha pasado de los 11,6 muertos diarios en carretera en 2000 a los 3,1 fallecidos diarios en 2015.

Con estos resultados de siniestralidad en vías interurbanas, España sigue manteniendo la tasa de mortalidad más baja de la historia, ocupa el quinto lugar de países en el mundo con mejor seguridad vial y es junto a Dinamarca y el Reino Unido los tres únicos países de la UE que cerraron 2015 con menos fallecidos que en 2014. Otros países como Suecia y Holanda mantendrán su misma cifra de fallecidos mientras que Francia, Alemania, Austria, Finlandia o Grecia registrarán incrementos de víctimas mortales.

Con estas cifras, España presenta una tasa de 3,6 muertos por 100.000 habitantes, muy por debajo de la tasa de mortalidad media de la UE, situada en 5,1 por cada 100.00 habitantes.

Datos a tener en cuenta para la contextualización de las cifras

En la siniestralidad de 2015 destacan las siguientes circunstancias:

- Se ha constatado que ha habido un aumento del 4% más de desplazamientos por carretera (un total de 373.504.129 desplazamientos de largo recorrido)
- La edad media de los vehículos implicados en accidentes mortales continúa aumentando. En 2015 se situó en 11,3 para los turismos y en 9,6 para las motociclistas.
- Más infracciones por consumo de drogas. Por ejemplo, se han hecho 3.220 pruebas a conductores infractores con 1.886 positivos (59%).
- Aumenta en 45 el número de fallecidos en vías convencionales respecto a 2014. En cambio, continúa la reducción de fallecidos en vías interurbanas de alta capacidad, con 51 muertos menos que en 2014.
- Persiste un reducido número de usuarios que continua sin utilizar los elementos de seguridad. En 2015, se considera que 175 fallecidos que no hacían uso de los dispositivos de seguridad (cinturón y casco) en el momento del accidente, podrían haber salvado la vida en caso de llevarlos.

Se puede obtener toda la información respecto a las características de la siniestralidad en 2015 en la nota de prensa del 4 de enero de 2016 de la DGT (DGT, 2016). En resumen:

- Los varones siguen teniendo un mayor índice de mortandad frente a las mujeres.
- El rango de edad con mayor índice de fallecidos se sitúa entre los 35 y 44 años, seguido del grupo de 45 a 54 años.
- Salirse de la vía a alta velocidad y las colisiones frontales son las causas más comunes de accidentes mortales.

Respecto al uso de accesorios de seguridad, hasta el 22% de los conductores y pasajeros fallecidos durante el 2015 no llevaban el cinturón de seguridad. La buena noticia es que, en general, ha aumentado el uso de dicho dispositivo desde 2014 para turismos (DGT, 2016).

Entre los menores fallecidos (13 niños menores de 12 años) en viajes de turismos, 4 de ellos no utilizaban ningún accesorio de seguridad en el momento del accidente.

2.2.5. Las glorietas como punto de accidentes.

No hay una regulación específica en la normativa de tráfico sobre la circulación en las glorietas o rotondas, técnicamente definidas como «un tipo especial de intersección caracterizado por que los tramos que en él confluyen se comunican a través de un anillo en el que se establece una circulación rotatoria alrededor de una isleta central». Falta esa regulación, pero sí hay tres principios básicos que estipulan como se debe actuar en ellas.

El primero, que quien va a entrar en una glorieta tiene que ceder el paso a los que se encuentran ya en ella.

El segundo, que una vez dentro, el vehículo que circula por el carril derecho tiene preferencia, como en cualquier otra vía. Como para salir de la glorieta hay que situarse en el carril más externo, esto conlleva un cambio de carril, y del párrafo del artículo 74 del Reglamento General de Circulación (DGT, 2015b) se extrae que siempre que se realice un cambio de carril se debe ceder el paso a los vehículos que ya estén circulando por el carril que se va a ocupar: "*Toda maniobra de desplazamiento lateral que implique cambio de carril deberá llevarse a efecto respetando la prioridad del que circule por el carril que se pretende ocupar (artículo 28.2 del texto articulado)".* De acuerdo además con el párrafo siguiente de dicho artículo, realizar el cambio de carril sin respetar la prioridad de los vehículos ocupantes de dicho carril tiene la consideración de infracción grave.

Y tercero, que para abandonar la glorieta hay que situarse previamente en el carril derecho porque es el exterior, por el que se debe salir de la intersección.

Con esos tres conceptos claros no debería haber confusiones.

En la figura 2.3 se pueden ver varios casos y una breve explicación de cuáles son comportamientos correctos e incorrectos al realizar una glorieta:

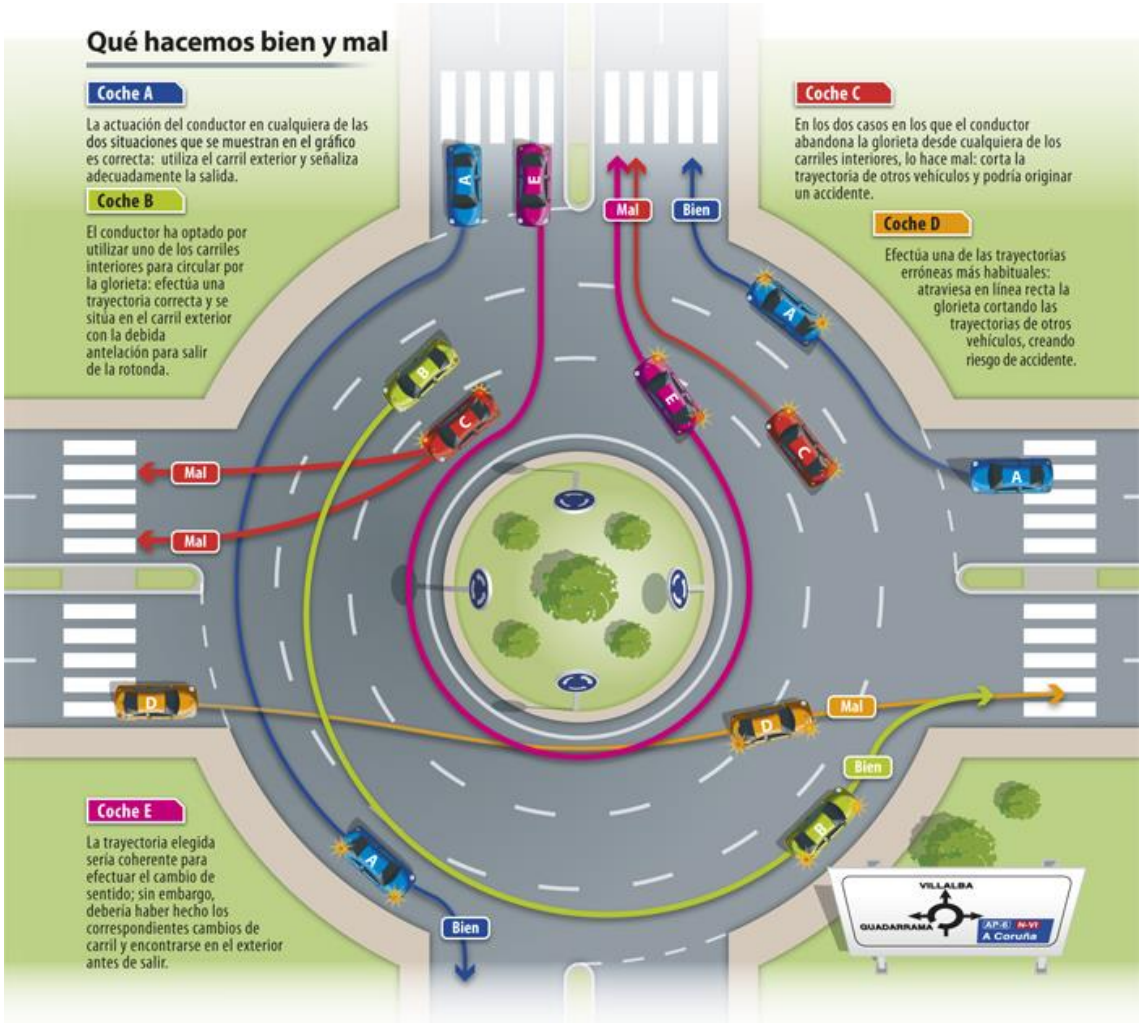


Figura 2.3 – Realizaciones correctas e incorrectas en una glorieta (González, 2014).

Sin embargo, las dudas de muchos conductores se deben a esa falta de regulación específica sobre cómo circular en una glorieta. Esa falta de una normativa específica para circular por una glorieta ha llevado a algunos municipios españoles a diseñar la circulación en sus propias glorietas. Es el caso de la ciudad de A Coruña,

donde hay glorietas que permiten abandonar el anillo directamente desde el carril izquierdo (figura 2.4), al contrario de lo que es regla básica en la mayoría de las glorietas de España (La Voz de Galicia, 2016). Sin embargo, ese sistema no es ilegal porque hay una señalización y unas marcas viales que indican el camino a seguir por los vehículos.

Figura 2.4 – Glorieta en A Coruña.

Otro caso singular se da en Vigo, con las novedosas turborrotondas, que han surgido en varias ciudades como medio para agilizar la circulación urbana (figura 2.5). Unas líneas continuas en el asfalto encauzan el tráfico y en teoría facilitan las cosas porque los conductores tienen menos complicaciones a la hora de elegir el carril una vez que se encuentran dentro de la intersección.

En cualquier caso, aún es pronto para estudiar la viabilidad de estas nuevas opciones.

Figura 2.5 – Turborrotonda.

Está claro que mientras no exista una regulación clara, será difícil concienciar a los conductores sobre cómo deben obrar cuando realizan una glorieta. Sin embargo, si se insistiera en que estos recuerden los principios arriba mencionados, sería más fácil que no se incurriera en infracciones y accidentes al maniobrar en una glorieta.

2.3. Conducción eficiente.

La **conducción eficiente** consiste en una serie de técnicas de conducción que, unidas a un cambio en la actitud del conductor, dan lugar a un nuevo estilo de

conducción acorde a las nuevas tecnologías y sistemas que incorporan los vehículos modernos.

Las Instituciones buscan apoyar la conducción eficiente colaborando en la realización de cursos prácticos, impartidos en las distintas Comunidades Autónomas, tanto a conductores particulares de vehículos de turismo, como a conductores profesionales de vehículos de transporte y profesores de autoescuelas.

Un uso eficiente del coche implica también priorizar la utilización del transporte público y de los vehículos no motorizado (bicicleta y a pie) para desplazamientos cortos. Además, compartir el coche con otros viajeros mejora la eficiencia energética de nuestros desplazamientos y contribuye a agilizar y descongestionar el tráfico. Existen algunas claves que indican si se está siendo eficiente a la hora de conducir, entre las que se van a nombrar diez a continuación (Comisariado Europeo del Automóvil, 2015):

1. Arranque y puesta en marcha:
   - Arrancar el motor sin pisar el acelerador.
   - Iniciar la marcha inmediatamente después del arranque.
   - En motores turboalimentados, esperar dos o tres segundos antes de iniciar la marcha.
2. Uso de la primera marcha:
   - Usarla solo para poner en movimiento el vehículo.
   - Cambiar a la segunda marcha con celeridad, a los 2 segundos o a los 6 metros aproximadamente.
3. Aceleración y cambios de marchas:
   - Según las revoluciones:
     - En los motores de gasolina: en el entorno de las 2.000 rpm.
     - En los motores diésel: en el entorno de las 1.500 rpm.
   - En la figura 2.6 se puede observar los puntos de cambio de marcha recomendados según la velocidad:
     - A 2ª marcha: 2 segundos o 6 metros aproximadamente.
     - A 3ª marcha: de los 20 a 25 km/h.
     - A 4ª marcha: de los 35 a 40 km/h.
     - A 5ª marcha: a partir de unos 50 km/h.
   - Acelerar de forma ágil tras la realización del cambio.

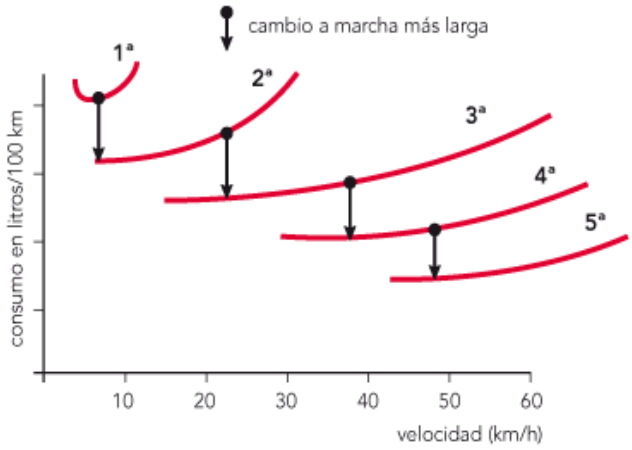


Figura 2.6 - Zonas de cambio a marchas largas.

4. Utilización de las marchas:
   - Circular lo más posible en las marchas más largas y a bajas revoluciones.
   - Es preferible circular en marchas largas con el acelerador pisado en mayor medida que en marchas cortas con el acelerador menos pisado.

- En ciudad, siempre que sea posible, utilizar la 4ª y la 5ª marcha, respetando siempre los límites de velocidad.
- En la figura 2.6 también se marcan los puntos de cambio de marcha más adecuados para reducir el consumo de combustible al máximo durante la conducción. Por otra parte, en la figura 2.7 se puede ver el consumo en litros a los 100km para distintas marchas a una velocidad de 60km/h según la cilindrada.

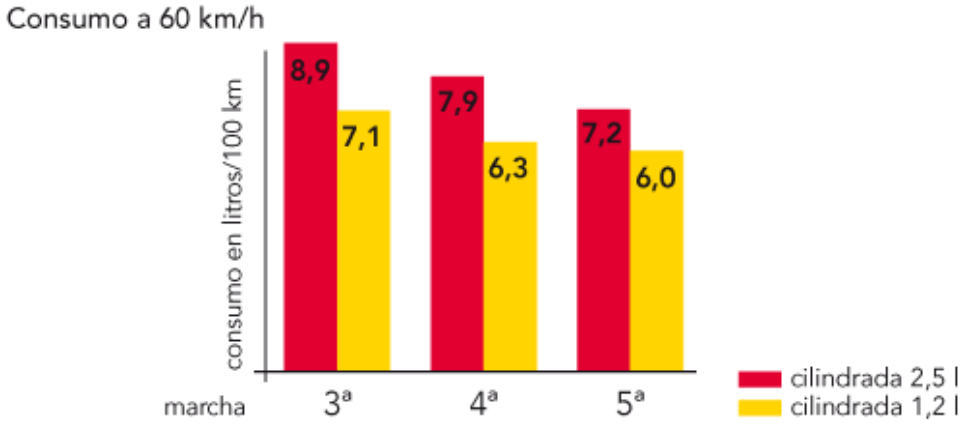


Figura 2.7 – Consumo en litros a los 100km a 60km/h según la cilindrada.

5. Velocidad de circulación:
   - Mantenerla lo más uniforme posible: buscar fluidez en la circulación, evitando los frenazos, aceleraciones y cambios de marchas innecesarios.
6. Deceleración:
   - Levantar el pie del acelerador y dejar rodar el vehículo con la marcha engranada en ese instante.
   - Frenar de forma suave con el pedal de freno.
   - Reducir de marcha lo más tarde posible, con especial atención en las bajadas.
7. Detención:
   - Siempre que la velocidad y el espacio lo permitan, detener el coche sin reducir previamente de marcha.
8. Paradas:
   - En paradas prolongadas (por encima de 60 segundos), es recomendable apagar el motor.
9. Anticipación y previsión:
   - Conducir siempre con una adecuada distancia de seguridad y un amplio campo de visión que permita ver 2 o 3 vehículos por delante.
   - En el momento en que se detecte un obstáculo o una reducción de la velocidad de circulación de los vehículos en la vía, levantar el pie del acelerador para anticipar las siguientes maniobras.
10. Seguridad:
    - En la mayoría de las situaciones, aplicar las reglas de la conducción eficiente contribuye al aumento de la seguridad vial.
    - Pero obviamente existen circunstancias que requieren acciones específicas distintas, para que la seguridad no se vea afectada.

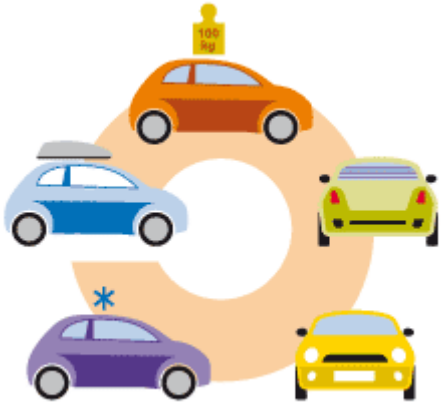


Figura 2.8 – Elementos que pueden aumentar el consumo del vehículo.

La figura 2.8 representa algunos de los elementos que pueden afectar al consumo, entre ellos:

- **Aire acondicionado:** debe usarse de forma racional, manteniendo la temperatura interior del habitáculo en torno a 24ºC. Hay que recordar también que para ventilar el habitáculo lo más recomendable es utilizar siempre que sea posible la circulación de aire forzada del vehículo y las ventanillas a bajas velocidades (entornos urbanos).
- **Otros sistemas del vehículo:** se ha de evitar el uso innecesario de accesorios exteriores (como la baca), luneta térmica y las luces de cruce, ya que incrementan el consumo de carburante.
- **Mantenimiento:** la utilización de aceites sintéticos mejorará las prestaciones del motor, alargando su vida y la del catalizador y reduciendo los contaminantes con un importante ahorro de carburante.

2.4. Simuladores de conducción.

Un simulador es un símil de la realidad. Es quizá la aplicación que más aprovecha las especificaciones de un ordenador como recurso de aprendizaje y que cada día se extiende más en áreas tanto de educación como de administración.

El simulador permite al estudiante aprender de manera práctica, a través del descubrimiento y la construcción de situaciones hipotéticas.

En definitiva, un simulador es una configuración de hardware y software en la que, mediante algoritmos de cálculo, se reproduce el comportamiento de un determinado proceso o sistema físico.

En este proceso se sustituyen las situaciones reales por otras creadas artificialmente de las cuales se aprenden ciertas acciones, habilidades, hábitos, etc., que posteriormente se transfieren a una situación de la vida real con similar efectividad; esta es una actividad en la que no solo se acumula información teórica, sino que se la lleva a la práctica.

En el mundo educativo, los simuladores constituyen un procedimiento, tanto para la formación de conceptos y construcción en general de conocimientos, como para la aplicación de estos a nuevos contextos a los que, por diversas razones, el estudiante no puede acceder desde el contexto metodológico donde se desarrolla su aprendizaje. De hecho, "*buena parte de la ciencia puntera, de frontera, se basa cada vez más en el paradigma de la simulación, más que en el experimento en sí...*" (Pozo y Gómez, 2006).

Mediante los simuladores se puede, por ejemplo, desarrollar experimentos de química en el laboratorio de informática con mayor seguridad. De ese modo, si a un estudiante se le ocurre agregar más de un determinado líquido, la explosión que esto cause será una simple "simulación" y cuando vaya a realizarlo en la práctica él estará informado de las consecuencias de este proceso.

Ventajas que aportan los simuladores a nivel educativo:

1. Apoyan aprendizaje de tipo experimental y conjetural.
2. Permite la ejercitación del aprendizaje.
3. Suministran un entorno de aprendizaje abierto basado en modelos reales.
4. Alto nivel de interactividad.
5. Tienen como finalidad enseñar un determinado contenido.
6. El usuario trata de entender las características de los fenómenos, cómo controlarlos o qué hacer ante diferentes circunstancias.
7. Promueven situaciones excitantes o entretenidas que sirven de contexto al aprendizaje de un determinado tema.
8. El usuario es un ser activo, convirtiéndose en el constructor de su aprendizaje a partir de su propia experiencia.

Todo lo mencionado anteriormente es perfectamente aplicable a los simuladores de conducción actuales. A día de hoy y cada vez más, tanto las administraciones orientadas a la seguridad vial como otros entornos como las empresas privadas, utilizan los simuladores de conducción para concienciar, obtener datos o educar.

Uno de los softwares de simulación de conducción más exitosos es **DriveSim,** que ha sido especialmente diseñado para autoescuelas, pensando en la formación de los futuros conductores (DriveSim, 2016). **DriveSim** ha sido reconocido internacionalmente, resultando finalista en los *Unity Awards* y premiado en el festival *Fun&Serious Games 2014* en la categoría de **Mejor Juego de Estrategia Empresarial**. Las figuras 2.9 y 2.10 muestran dos entornos de este simulador.

Figura 2.9 y Figura 2.10 – Imágenes del simulador DriveSim.

La principal ventaja de este simulador es su **carácter educativo**, ya que con él el usuario puede aprender a conducir desde cero. El programa cuenta con varios módulos: formación inicial en pista, formación vial básica, formación avanzada y conducción eficiente.

Esto permite que las personas que acuden a la autoescuela puedan aprender el manejo básico del coche antes de lanzarse a conducir con tráfico real. Primero, aprenderá el uso de los pedales y de todos los mandos del vehículo. La dificultad se incrementa progresivamente, realizando ejercicios en circuito cerrado, posteriormente en vías urbanas sin otros vehículos y luego podrá realizar ejercicios muy diversos en vías de alto rendimiento como autovías o autopistas e incluso en carreteras de montaña.

El programa está adaptado a las normas de circulación de varios países. Esto quiere decir que, por una autopista en la versión española, el alumno no podrá circular a más de 120km/h y, en caso de hacerlo, el programa emitirá una alarma. Lo mismo ocurre cuando hace un mal uso de los pedales, cuando pisa la línea del arcén, cuando se acerca mucho al vehículo que le precede y con muchas otras infracciones.

Gracias a esto, puede saber cuáles son sus puntos débiles, pudiendo mejorarlos antes de realizar las prácticas en la ciudad. Además, adquirirá los hábitos de ponerse el cinturón de seguridad y regular los mandos, ya que el programa **DriveSim** puede ir acompañado de un asiento de conducción que cuenta con todos los elementos necesarios para que la experiencia de la conducción sea muy realista: un asiento regulable, volante, palanca de cambios, intermitentes, luces, cinturón de seguridad, freno de mano, etc.

En el mundo del ocio existen otros simuladores más orientados a la diversión frente a la conducción responsable. Un gran número de usuarios, entre ellos pilotos de carreras, destacan que este tipo de simuladores suelen proporcionar una respuesta muy realista de los vehículos y unos circuitos muy detallados que les permiten conocer y adaptarse a nuevas competiciones, de tal forma que ya se conozcan los recorridos cuando lleguen a practicarlos en la vida real (iRacing, 2016). Simuladores (que no *arcades*) como iRacing, Project Cars, Gran Turismo o Forza Motorsport están reconocidos mundialmente por su fidelidad en la representación de la respuesta de vehículos de competición (en el caso de Gran Turismo incluso de vehículos de gama baja). Respecto a este tema, el usuario VagaXT realizó un video comparativo (Youtube, 2010) que demuestra la fidelidad con la que el Gran Turismo 5 representa los circuitos, en este caso el circuito Mazda Raceway Laguna Seca. En este vídeo se puede ver a pantalla partida el mismo circuito recorrido con el mismo vehículo, Mazda Miata Turbo, y realmente asombra el parecido entre ambas situaciones, es un logro muy difícil de refutar.

2.5. Contribución de los simuladores de conducción a la seguridad vial.

Cuando una persona se visualiza en un entorno de conducción como una gran ciudad con mucho tráfico y cruces complejos, es fácil que se vea incapaz de afrontar ese reto. Surgen el miedo y las dudas.

Los simuladores de conducción permiten que los usuarios puedan enfrentarse a muchas de estas situaciones en un entorno virtual que no ponga en peligro al propio individuo ni a otros conductores. Permitirán a los usuarios enfrentarse a problemas y a obtener unas habilidades y destrezas en la conducción que les permitan resolver estos problemas, de manera que puedan aplicar esas habilidades cuando se enfrenten a situaciones similares en un entorno real. Los usuarios pueden acostumbrarse a los controles del coche mediante este tipo de simuladores, manejándolos con pedales, volante y cambio de marchas al igual que lo harían en un coche real. Para ello se puede recurrir a todo tipo de periféricos que pueden ir aumentado en complejidad (y por lo tanto en parecido con la realidad), desde mandos a sillas completas de conducción, pasando por volantes como el Logitech G29 o el más antiguo Logitech G27.

Obviamente los simuladores aportan la posibilidad de repetir los escenarios y eventos de una simulación de forma indefinida. Además, es posible simular situaciones de la vida real que serían prácticamente imposibles de simular de forma voluntaria en un entorno real, al menos sin poner en riesgo a personas y vehículos por igual. Situaciones como un desprendimiento, un vehículo en medio de la carretera o un peatón

cruzando por la carretera de forma inesperada son eventos que podrían ser simulados fácilmente en un programa informático, siendo prácticamente imposibles de recrear en la vida real. Esto permitirá a los usuarios de un simulador estar mejor preparados ante situaciones inusuales que se les puedan presentar en el futuro.

Los simuladores permiten realizar estudios de cómo ciertos agentes pueden influir en las capacidades de conducción, tales como el cansancio, el estado físico del conductor, el consumo de medicamentos, drogas o alcohol, enfermedades, los efectos psicológicos o la edad.

Por último, es importante mencionar que los simuladores permiten recopilar datos de forma simple. En el proyecto que nos ocupa se ha creado un sistema que permite que estos datos sean almacenados y procesados por un servidor remoto, permitiendo hacer comparaciones automáticas. A nivel de usabilidad, también permite a los usuarios acceder a información de las simulaciones que han realizado y compararlas con datos medios de otros usuarios. Esto permite que un conductor pueda ver qué fallos o infracciones ha cometido durante su conducción y en qué campos debe mejorar.

3. Tecnologías.

En este apartado se expondrá información acerca de las tecnologías que se van a emplear para el desarrollo de nuestro sistema cliente/servidor.

Se requerirá hacer una elección entre las muchas tecnologías existentes para cada una de las aplicaciones.

Para la creación del simulador será necesario utilizar un motor gráfico que nos dé todas las herramientas necesarias para crear la lógica de los eventos y la física de los elementos. Se mencionarán brevemente los programas de diseño 3D, aunque han sido poco utilizados durante el desarrollo de la aplicación.

Para el desarrollo de la plataforma web de almacenamiento y procesamiento de datos será necesario seleccionar una entre las múltiples tecnologías y lenguajes disponibles. Vamos a dividir el estudio de las alternativas entre las necesarias para el desarrollo del simulador y las necesarias para el desarrollo de la aplicación web.

3.1. Alternativas tecnológicas: Motores gráficos.

Aquí se comentarán y estudiarán las características de las diferentes opciones tecnológicas, tanto de motores de gráficos como de programas de modelado 3D, para posteriormente compararlas y elegir las más adecuadas para nuestro fin.

3.1.1. Motores gráficos.

Un motor gráfico, también llamado motor de juego o *game engine* es un conjunto de herramientas orientado de forma casi exclusiva a proporcionar a los profesionales del desarrollo todas las herramientas necesarias para crear aplicaciones gráficas, generalmente videojuegos.

El motor no es solo el cálculo de físicas o el *renderizado* de objetos tridimensionales. El motor gráfico es un software de alta complejidad orientado a la programación eventual que proporciona un conjunto enorme de clases, funciones y elementos para poder llevar a buen puerto el desarrollo de aplicaciones gráficas.

Existen un enorme número de motores en el mercado: de código libre, orientados al desarrollo 2D, al desarrollo 3D, etc.

Entre ellos podemos mencionar, por ser mundialmente conocidos:

- **Unity**, como el motor para todo. Permite el desarrollo tanto en 2D como en 3D y es uno de los motores más utilizados a día de hoy por los equipos de desarrollo *indie* (denominación que se le da equipos que aspiran al desarrollo de juegos de bajo a medio coste, generalmente proporcionando ideas y explorando campos que las grandes empresas no se atreven a afrontar por miedo a su viabilidad), debido a su versatilidad. En la figura siguiente (Figura 3.1) se puede ver el flujo de vida de un script en Unity. Quizás con ello se pueda entender la complejidad que manejan los motores gráficos:

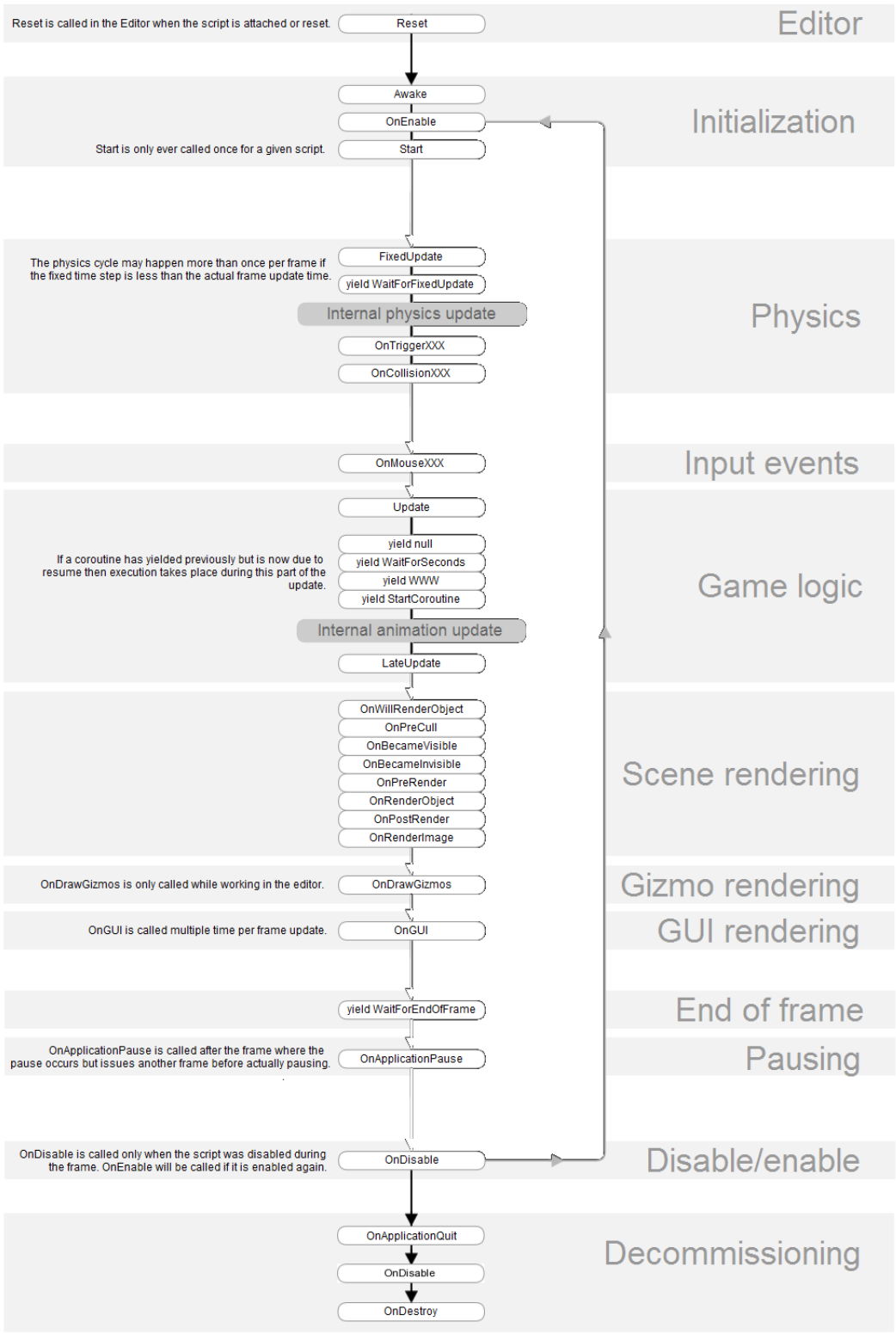


Figura 3.1 – Ciclo de vida de un script en Unity.

- **Unreal Engine**, como el motor de los juegos en primera persona. Un pipeline de *renderizado* exclusivo permite a este motor *renderizar* más elementos en pantalla que la mayoría de motores, logrando resultados visuales sorprendentes.
- **CryEngine** como el motor de los mundos abiertos. Desarrollado por *Crytek* para la creación del juego *Farcry*. Se ha convertido en uno de los motores más utilizados actualmente.

Cabe mencionar que muchísimas empresas de desarrollo acaban creando sus propios motores en lugar de utilizar software de terceros. Por ejemplo, *Square Enix*

decidió montar su propio motor *Crystal Tool* para el desarrollo de Final Fantasy XIII y sus sucesores. Este mismo motor mejorado se está utilizando en el desarrollo de su último juego, Final Fantasy XV.

3.1.1.1. Unity.

Unity es una plataforma de desarrollo flexible y potente para la creación de juegos y aplicaciones gráficas multiplataforma tanto bidimensionales como tridimensionales (Unity, 2015). Realmente tiene dos motores de física por separado, uno para trabajar en 2D (*BoxPhysics*) y otro para 3D (*Nvidia PhysX*).

Figura 3.2 – Plataformas a las que Unity puede exportar sus aplicaciones.

Tiene la posibilidad de moverse libremente entre 21 plataformas distintas (Figura 3.2). En las versiones más recientes del motor, se está optando por darle un gran peso a las plataformas de Realidad Virtual como *Oculus Rift*.

Unity tiene un editor intuitivo y muy personalizable, que hace que la experiencia de usuario sea excelente. Este editor se puede extender mediante la creación de scripts que permiten ejecutar funcionalidades como crear objetos (más allá de los que vienen por defecto) o realizar procesos. En Unity se pueden extender fácilmente los inspectores de elementos. Existen extensiones ya creadas por la comunidad que permiten a cada usuario adaptar las funcionalidades del programa a sus necesidades para cada proyecto. Este motor lleva muchos años en el mercado y se ha ido adaptando con gran rapidez a las exigencias de los nuevos desarrollos. Teniendo en cuenta su orientación multiplataforma, presenta un acabado gráfico reseñable.

La forma de importar archivos es muy sencilla, basta con arrastrarlos a *Unity* y se importarán de forma automática.

La herramienta *Profiler* es, posiblemente, una de las importantes de cara al desarrollo y, desde la versión 5, viene incluida por defecto. Básicamente es una herramienta de análisis que permite depurar la aplicación. Permite detectar el rendimiento de la aplicación en distintas áreas como en la memoria utilizada, la velocidad de ejecución de los scripts o para ver el tiempo *renderizado*.

En Unity todos los objetos son *GameObjects* y la programación se hace orientada a componentes. Un *GameObject* por sí mismo es solamente una posición en el escenario. Los componentes desarrollados (los scripts) son lo que le dan funcionalidad. Los scripts en Unity se pueden programar en C# y en UnityScript, siendo esta última una versión de JavaScript semitipada creada exclusivamente para Unity. En definitiva, los scripts son funcionalidades (comportamiento físico de los objetos, crear efectos gráficos, etc.) que pueden ser añadidas a un *GameObject* para darles un comportamiento.

Unity 4.6 incluyó dos sistemas nuevos que fueron muy bien recibidos por la comunidad: *Mecanim*, que permite modelar las animaciones como máquinas de estados y modelar las interacciones entre ellas como transiciones; y el sistema UI, que permite crear interfaces de usuario complejas como incluir elementos en el propio juego. En la figura 3.3 se puede ver una *State Machine* de *Mecanim* y en la figura 3.4 se puede observar una interfaz de usuario modelada sobre la escena.

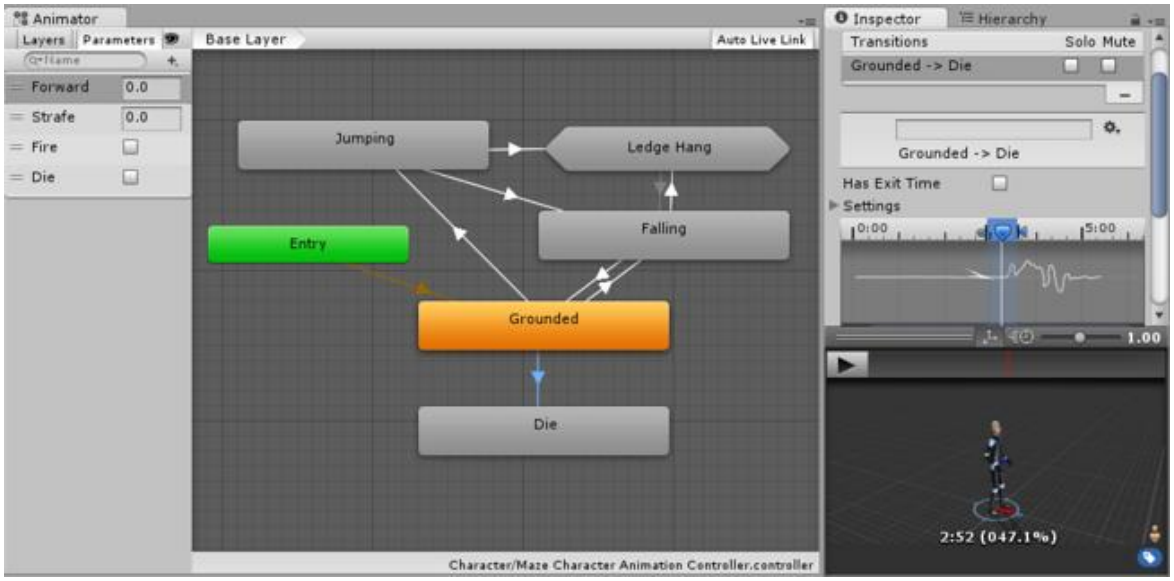


Figura 3.3 – Mecanim Animator.

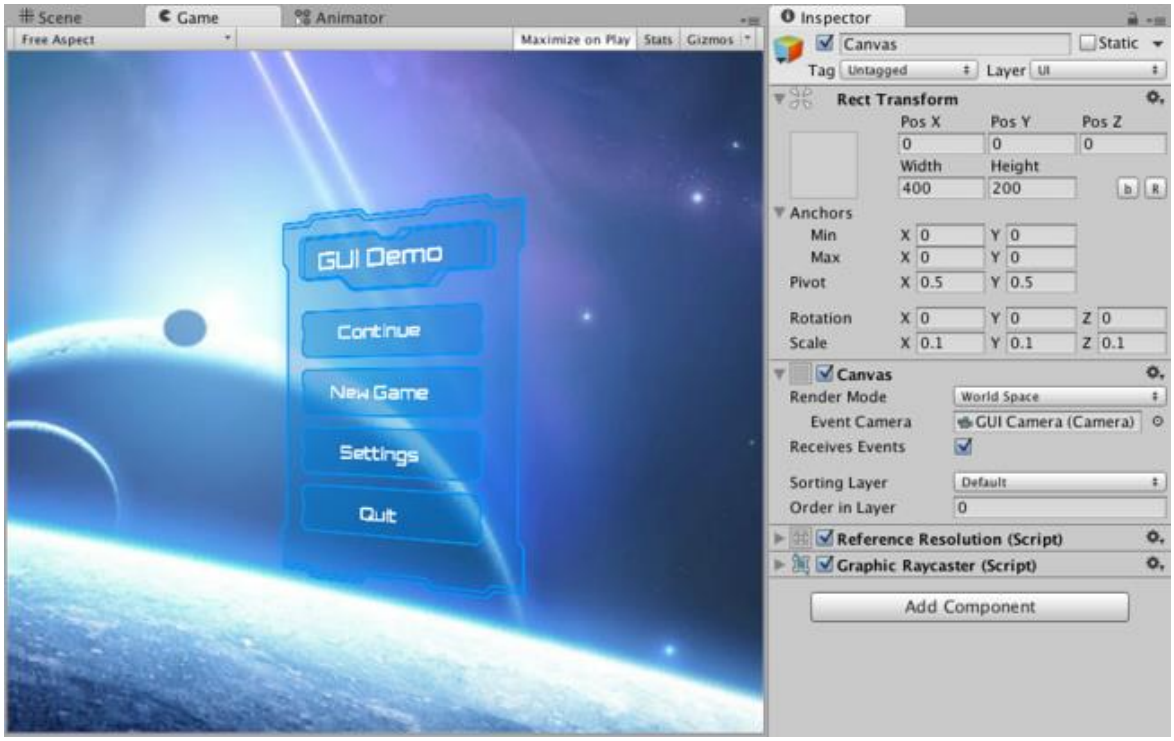


Figura 3.4 – Ejemplo de una interfaz dentro del escenario usando UI.

Unity dispone de una plataforma, denominada *Asset Store*, donde encontrar "*assets*", tanto gratuitos como de pago, que son componentes que encapsulan funcionalidades reutilizables, tales como animaciones, modelos tridimensionales y bidimensionales, audio, proyectos completos, sistemas de partículas, scripts, texturas, materiales, etc.

La comunidad que hay detrás de Unity es gigantesca. Debido a que es un software multiplataforma ha conseguido atraer a muchos equipos de desarrollo que han hecho que la comunidad crezca enormemente en los últimos años.

Podemos ver un ejemplo de proyecto en *Unity* en la figura 3.5.

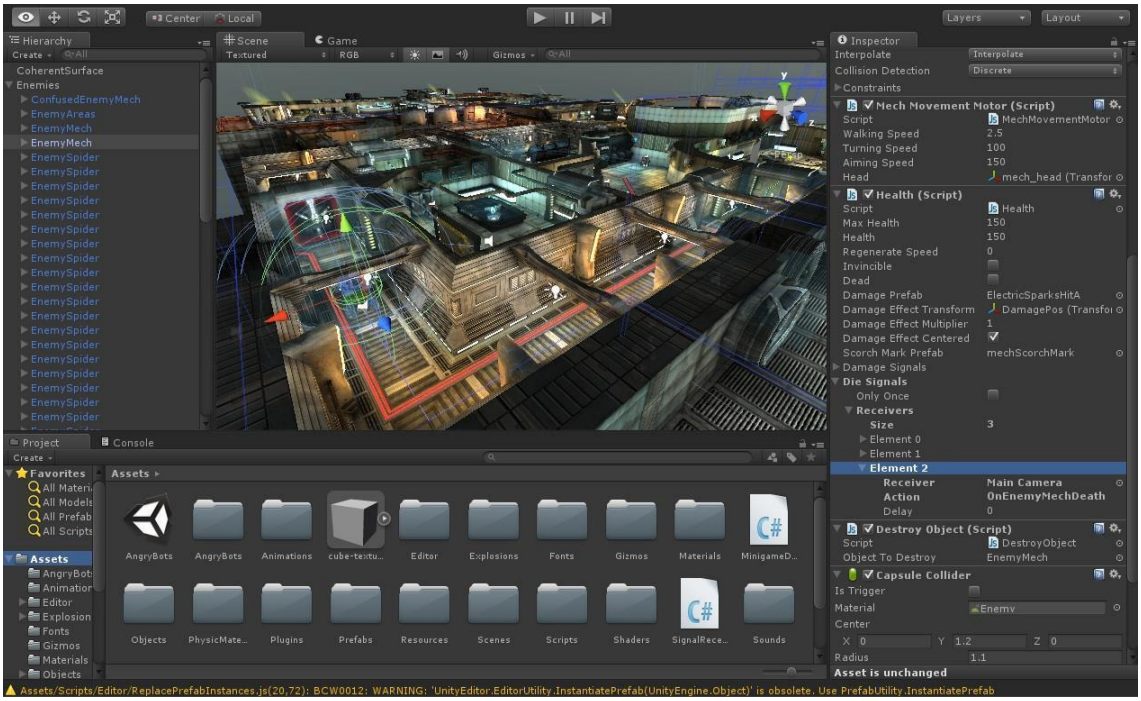


Figura 3.5 – Proyecto *Stealth* en Unity 4.6.

En cuanto a los requisitos del sistema, para desarrollo necesitamos como sistema operativo Windows XP SP2+, 7 SP1+, 8 o Mac OS X 10.8+. En cuanto a la GPU, se necesitan tarjetas de video con soporte para DX9 (modelo *shader* 2.0). Proyectos más complejos requerirán de equipos más potentes para el desarrollo.

Su último motor gráfico: Unity 5.

La quinta versión de este popular motor fue presentada el 3 de marzo de 2015 en la *Game Developers Conference*. Una de las claves de Unity es que el juego puede ser portado a 21 plataformas diferentes empleando un solo código, lo cual facilita mucho a los desarrolladores crear una versión de su juego para varias plataformas. Unity 5 además incluye soporte para *Oculus Rift*.

La nueva versión incluye una "mejora inmensa en las capacidades gráficas", como la iluminación en tiempo real, *shaders* de base física construidos de materiales del mundo real y reflejos HDR (*High Dynaimc Range*). Con la inclusión de *Enlighten* como motor de iluminación, Unity 5 busca poder enfrentarse a otros motores que le aventajaban en calidad visual y poder atraer a desarrolladores de juegos que requieran un gran acabado gráfico y visual (Unity, 2015).

Unity es un motor gráfico históricamente asociado a los juegos para dispositivos móviles, pero el lanzamiento de Unity 5 y su nuevo sistema de *renderizado* suponen nuevas y potentes capacidades para crear juegos con gráficos realistas que ofrezcan una gran inmersión al jugador para dispositivos cada vez más potentes.

Es evidente que en 2016 estamos entrando, por fin, en la guerra *next-gen* con Unity intentando enfrentarse cara a cara a los motores gráficos más potentes como CryENGINE y Unreal Engine 4.

3.1.1.2. Unreal Engine.

Unreal Engine (de ahora en adelante UE) es un conjunto de herramientas para desarrollo de juegos que ha venido usándose durante muchos años en el desarrollo de juegos de alto coste económico (comúnmente denominados juegos Triple A).

Ha sido creado en C++, y permite el acceso a su código fuente, de modo que es posible personalizar y ampliar las herramientas de su editor, la física, audio, animación, *renderizado*, además de la interfaz de usuario, al menos para las personas con capacidades para poder modificarlo. Las últimas versiones de *Unreal* soportan *DirectX* 11 y 12 (Epic Games, 2015). La figura 3.6 representa una imagen del editor de su último motor, Unreal Engine 4.

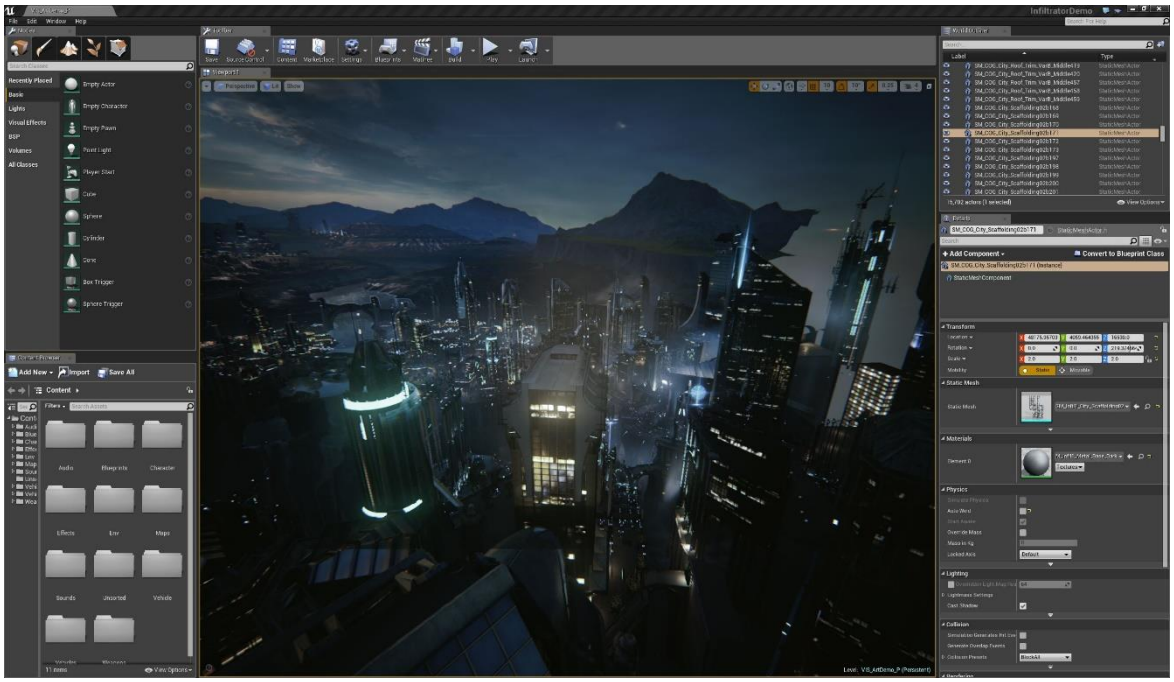


Figura 3.6 – Proyecto Demo en Unreal Engine 4.

Entre las características del Unreal Engine se pueden mencionar su editor Cascade VFX, el sistema de scripting visual Blueprint o las herramientas de animación de modelos (Epic, 2015).

Unreal dispone del editor *Cascade VFX* para la creación de efectos de partículas, que permite una simulación detallada de fuego, nieve, o polvo. Su sistema de iluminación siempre ha sido de alto nivel, aunque muchos de sus desarrolladores quieren que *Unreal Engine* recurra a otros motores de iluminación como ha hecho Unity, evitando el tener que usarlos como librerías externas.

Blueprint Visual Scripting es posiblemente una de las herramientas más interesantes de este motor. Por desgracia, a pesar de los esfuerzos del equipo de desarrollo por hacer de C++ un lenguaje de fácil uso, muchos desarrolladores de juegos se han visto incapaces de hacerse a él de forma rápida, lo que conllevaba múltiples abandonos del motor. El equipo decidió optar por el desarrollo de una herramienta de Scripting Visual que acabarían llamando *Blueprint*. En concreto permite construir la lógica de los scripts mediante bloques relacionados entre sí.

En cuanto a las animaciones, la herramienta *Persona Animation* permite editar esqueletos (*rigging*), crear secuencias de animación y modificar las propiedades de su física. Unreal posee una herramienta para la generación de secuencias dinámicas (al efecto de secuencias animadas sin interacción del usuario) denominada *Matinee Cinematics*.

Una funcionalidad interesante de *Unreal Engine* es que es posible navegar por las funciones de C++ directamente desde los personajes y objetos del juego, accediendo desde ahí a las líneas de código fuente. Es posible cambiar el código mientras el juego está en ejecución para ver esos cambios reflejados inmediatamente en el juego, sin necesidad de pausarlo. Mientras esté el juego en ejecución es posible dejar de verlo desde la cámara del jugador para observarlo desde otro punto de vista.

Aunque no es tan completo como la *Asset Store* de Unity, el *Marketplace* también permite acceder a recursos y proyectos de pago.

Su último motor gráfico: Unreal Engine 4.

Unreal Engine 4 (a partir de ahora UE4) es la última versión del motor gráfico de *Epic Games.* UE4 tiene unas capacidades gráficas impresionantes, incluida la iluminación dinámica y un sistema de partículas que permite manejar un millón de partículas en una misma escena. Un sueño hecho realidad para los artistas 3D.

UE4 te permite portar los videojuegos a PC, Mac, iOS, Android, Xbox One y PlayStation 4. El motor te brinda la posibilidad de crear videojuegos sencillos para plataformas móviles o videojuegos que expriman el hardware más potente disponible del momento en el mercado de consumo.

Posiblemente el juego más actual realizado con UE4 sea *Paragon* (Epic Games, 2016), que está siendo desarrollado por la propia Epic.

Unreal Engine 4 es totalmente gratuito en la actualidad, aunque hay que pagar un 5% en *royalties* a no ser que se gane menos de $3.000 por trimestre y juego. Es una de las opciones ideales para empezar a desarrollar juegos si el estudio de desarrollo no es muy grande.

3.1.1.3. CryEngine.

CryEngine es un motor de juegos desarrollado por *Crytek* para el que sería su primer videojuego, *Farcry*. Tiene soporte para PC, *Playstation 4*, *Xbox One*, *Wii*, *Android e iOS.*

Cuenta con el editor *SandBox*, que permite que varias personas trabajen a la vez en un mismo escenario dividiendo el entorno en capas. Al igual que otros motores, cuenta con la funcionalidad WYSIWYP (*What You See Is What You Play*). En la figura 3.7 se puede observar el editor junto a un escenario creado con *CryEngine 3.*

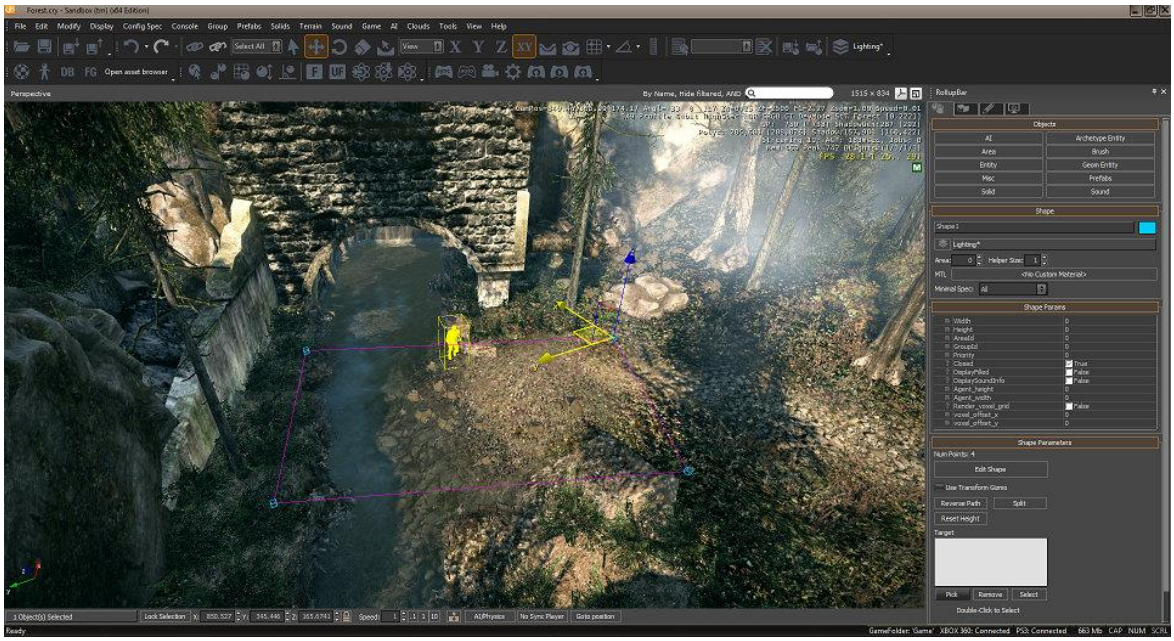


Figura 3.7 – Proyecto en CryEngine.

Durante varios años fue, posiblemente, el motor con mejor iluminación dinámica de escenas del mundo. *CryEngine 2* fue uno de los primeros motores en utilizar DirectX 10 y desde un inicio fue capaz de aprovechar muchas de las capacidades ofrecidas por esta librería (Crytek, 2016).

Dispone de un completo editor de terrenos con muchas opciones de personalización. Como se ha mencionado anteriormente, *CryEngine* es una de las opciones más claras cuando se opta por desarrollar un juego en escenarios amplios.

Además, dispone de técnicas para simular el efecto de la reflexión y refracción en tiempo real que permiten que cualquier tipo de superficie pueda dar reflejo en sus alrededores.

Los personajes son animados mediante *Sandbox 3*, donde podemos ver y editar los estados y las transiciones entre las animaciones del personaje. Pueden seguir caminos o subir y bajar pendientes, además de reaccionar ante ciertos cambios en el entorno. Su editor facial permite analizar audio, para de este modo ver la posición de los labios que deseamos, tomando la forma que sería necesaria para hacer ese sonido. Se les puede configurar para que muestren entre sí hostilidad, amistad u otros sentimientos, mediante características faciales.

En cuanto a la física, se puede trabajar con fuerzas como son la gravedad, las corrientes de viento, explosiones, colisiones y fricción. Es posible realizar efectos como podría ser la ondulación de una bandera o la física de una cuerda.

En cuanto a los lenguajes soportados, es posible trabajar con C++ o LUA. Además, cuenta con un sistema visual de scripting bastante potente similar a *Blueprint* de UE4 o al "*asset*" PlayMaker de Unity, que permite crear la lógica del juego de una forma intuitiva, sin necesidad de escribir nada de código.

CryEngine cuenta con documentación online para poder consultar cualquier información acerca de cómo utilizar este motor, además de con una comunidad razonablemente activa.

Su último motor: CryENGINE.

La última versión del motor se vuelve a llamar CryENGINE, sustituyendo a su predecesor CryEngine 3. Las capacidades y potencia de este motor sobrepasan de largo a motores como Unity (quizá no tanto desde la versión 5 de este último). Tan solo Unreal Engine 4 puede competir en aspectos como las físicas, los sistemas de animación de modelos y la capacidad de iluminar en tiempo real las escenas. Actualmente es libre para uso no comercial.

CryENGINE es un motor gráfico fantástico, pero tiene una gran curva de aprendizaje, por lo que es necesario invertir bastantes horas antes de comenzar a ser productivo con él.

El 22 de marzo de 2016, Crytek anunció una nueva versión de su motor denominada de momento CryEngine V, que soporta DirectX 12 nativo y realidad virtual o VR. Se ha añadido además un nuevo modelo de licencia que permite el PAMAYW (*Pay As Much You Want)* para el uso del motor y acceso al código fuente.

3.1.2.Programas de modelado 3D.

Aquí se tratarán diferentes programas de modelado 3D y sus características, para más adelante poder elegir cuál se adecua más a lo que deseamos.

3.1.2.1. 3ds Max.

3ds Max es un software de modelado, animación y *renderizado* 3D perteneciente a *Autodesk*. Cuenta con soporte web técnico directo, soporte prioritario en los foros y licencias flexibles (Autodesk, 2016).

En cuanto a animación 3D, cuenta con un sistema de secuencia de cámara que permite cortar entre varias cámaras, recortar y reordenar clips, manteniendo intactos los originales. Permite crear personajes con pieles muy realistas mediante el modificador *Skin*. Mediante la función *Populate* permite crear multitudes tanto dinámicas como estáticas, además de permitir generar comportamientos y animaciones como andar, correr, girar, sentarse, etc.

Contiene herramientas para la manipulación de los personajes, de forma que se puedan añadir esqueletos y articulaciones (*rigging*). Tiene mecanismos de animación y simulación de materiales como por ejemplo fluidos.

En relación a la texturización y el modelado, *3ds Max* tiene compatibilidad con *OpenSubdiv*, de forma que se puede obtener un mayor rendimiento en la ventana gráfica para mallas con muchas subdivisiones; cuenta con muchas opciones de sombreado e interoperabilidad de sombreadores con *Maya* y *Maya LT*; permite crear una amplia gama de materiales; se pueden utilizar nubes de puntos para crear modelos precisos a partir de referencias reales (incluso *Motion Capture*); tiene opciones para diseñar mapeados creativos de texturas (como mosaicos, materiales simétricos, estampados, …).

3ds Max permite cargar gráficos vectoriales como mapas de texturas y *renderizarlos* de modo que los gráficos siempre sean nítidos y claros; además contiene funciones de modelado basadas en polígonos, *splines* y NURBS (*non-uniform rational B-spline*) que consisten en modelos matemáticos para representaciones tridimensionales de objetos con superficies curvas (Manual Autodesk, 2016).

Respecto a la *renderización* en 3D, *3ds Max* incorpora la *renderización* d elementos mediante la computación en la nube, de forma que no se necesite emplear la capacidad de procesamiento del ordenador utilizado; su cámara física incorpora velocidad de obturación, apertura, profundidad de campo y exposición, de forma que tiene una mayor similitud con las cámaras reales; tiene compatibilidad con las mejoras de *Iray* y *mental ray*, que permite la *renderización* de imágenes fotorrealistas. Se puede mejorar la *renderización* con mediante técnicas de *shading ActiveShade*; se puede suavizar el efecto de las texturas para trabajar con menos ralentizaciones; utilizando la grabadora se pueden capturar, grabar y editar diferentes momentos de la escena, y se puede *renderizar* solo la parte modificada sin volver a hacerlo para toda la escena; cuenta con técnicas avanzadas de gestión de escenas y análisis multiproceso que agilizan el trabajo.

En cuanto a la dinámica y efectos, pueden ser creados efectos de partículas y fluidos como agua, fuego o nieve, entre otros. Tiene algunos componentes de uso directo como *mRigids* para simular cuerpos rígidos, *mCloth* para simulador tejidos (ropa, capas, etc.), y *mParticles* para crear los efectos de partículas. Además, es posible realizar simulaciones y análisis de iluminación con *Exposure (*Autodesk, 2016*).*

Por último, en lo relacionado con la interfaz de usuario, el flujo de trabajo y la producción, facilita la creación de nuevas herramientas que posteriormente pueden ser compartidas con el resto de usuarios. Su espacio de trabajo *Design* facilita el acceso a las herramientas y permite la configuración de plantillas para tener una configuración de inicio predeterminada.

Los datos se pueden organizarse en capas anidadas para una mayor facilidad en el trabajo. 3ds Max se puede ampliar y personalizar mediante el lenguaje de programación *Python*. Existe otras funciones más específicas como *Civil View* que permite fácilmente transformar el espacio civil en un modelo 3D o *DirectConnect* que permite intercambiar datos de diseño industrial con ingenieros que utilizan herramientas *CAD*.

En la figura 3.8 se puede ver el editor del software 3ds Max

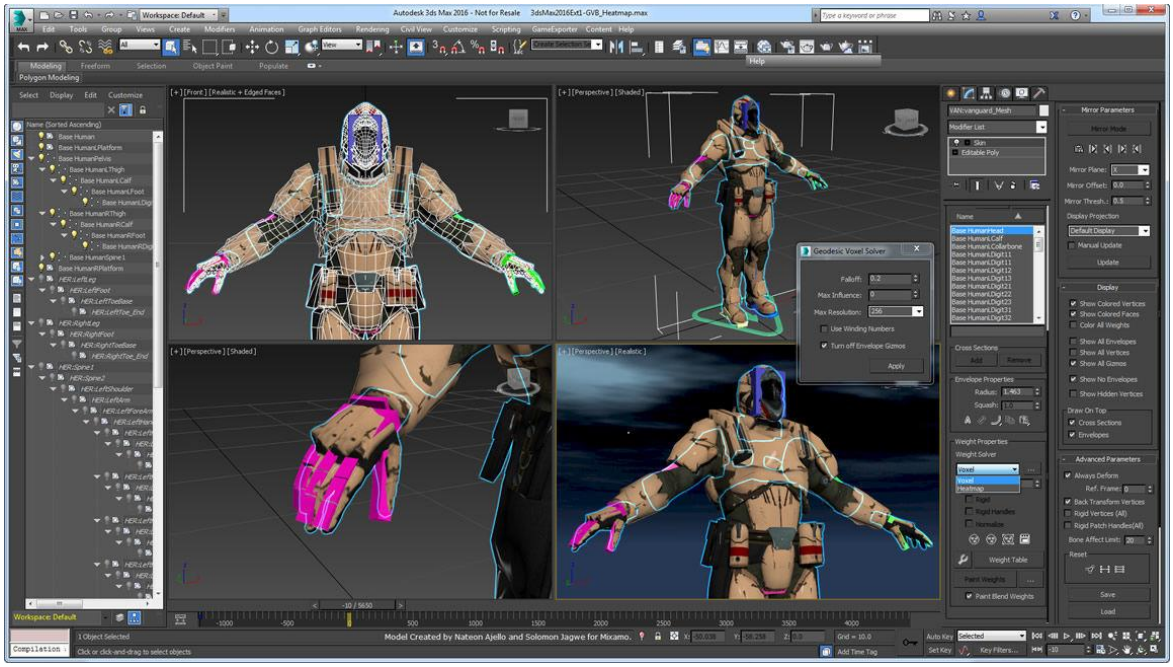


Figura 3.8 – Editor 3ds Max.

3ds Max cuenta con herramientas de aprendizaje tanto para comenzar a trabajar con este programa, como para personas más avanzadas en su manejo que pueden buscar respuestas a problemas que encuentren en el desarrollo de su proyecto.

3.1.2.2. Blender.

Blender es un software libre y de código abierto, para cualquier tipo de uso, que cuenta con un motor de *renderizado* que ofrece una representación muy realista, y con licencia permisiva para vincular con cualquier software externo (Blender, 2016). En la figura 3.9 observamos un proyecto en el entorno de trabajo de *Blender*.

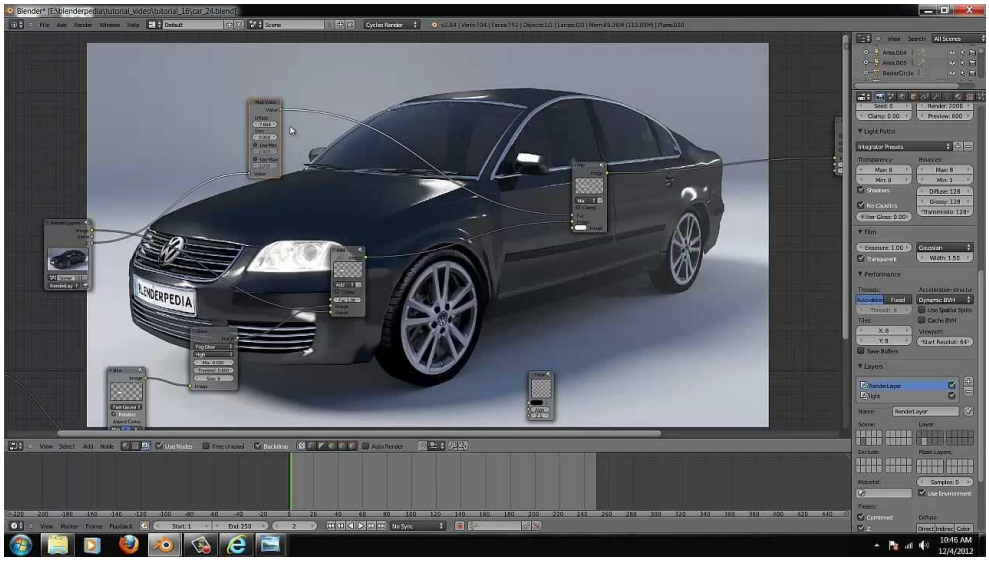


Figura 3.9 – Editor en Blender.

Cuenta con una amplia gama de herramientas para crear modelos, que pueden ser utilizadas de una manera muy rápida mediante atajos de teclado. Además, se pueden crear herramientas personalizadas y complementos, mediante el lenguaje de programación

Python. Su interfaz es flexible de forma que el diseño se podrá personalizar completamente.

Su motor de *renderizado* permite crear materiales realistas y simular objetos reales como puede ser el vidrio. En cuanto a los esqueletos, posee herramientas que permiten la modificación de los huesos de una manera sencilla. Además, contiene un conjunto de herramientas de animación con las que se pueden crear animaciones de personajes u otros elementos, incluso animaciones no lineales para movimientos independientes.

Para el modelado de los objetos, cuenta con diferentes pinceles para esculpir, e incluso herramientas para crear topologías variables. Sobre estos modelos se aplican texturas que pueden ser pintadas directamente sobre el modelo mediante pinceles o utilizar texturas UV (texturas que se usan como envoltorio o *wrapper* de un objeto tridimensional). Es posible realizar simulaciones de fluidos, humo, cabello, tela, objetos rígidos y partículas como podrían ser la lluvia, nieve, humo o chispas.

No es necesario exportar nada a programas ajenos, se puede hacer todo sin salir del programa. *Blender* es un motor gráfico completo, por lo que se puede crear y codificar la lógica de juego en él (Totten, C., 2012). Los *scripts* se hacen mediante *Python* y al igual que otros motores permite un entorno de *debug* dentro del propio *software*.

Este programa cuenta con una gran variedad de extensiones creadas por su comunidad de desarrolladores, que pueden ser activadas o desactivadas fácilmente: generación de árboles, esculpido de terreno, simulación de rotura objetos, herramientas para impresión 3D, etc.

También incluye un editor de video que permite cortar y juntar videos, mezcla de audio, sincronización, control de velocidad y transiciones, entre otras funciones más avanzadas.

Permite la importación/exportación de muchos formatos diferentes (dae, obj, etc.).

Blender cuenta con tutoriales *online*, con los que se pueden adquirir los conocimientos necesarios para trabajar con este programa.

3.2. Alternativas tecnológicas: Tecnologías web.

Las alternativas tecnológicas para el desarrollo de la plataforma web junto con la API REST son realmente enormes. Por ello solo se van a explicar de forma superficial algunas de las posibilidades, junto con sus funcionalidades o características.

Es posible desarrollar una plataforma web tanto usando un *framework* como partiendo de un gestor de contenidos (CMS – *Content Management System*). No se va a entrar en demasiados detalles de las diferencias entre ambos y de forma muy genérica se pueden referenciar indistintamente, pero hay que tener en cuenta que no son lo mismo.

Un *framework* (siempre en el entorno de las aplicaciones web) es un conjunto de librerías o componentes que permiten agilizar el desarrollo de aplicaciones mientras que un CMS es una aplicación ya funcional (generalmente construida usando un *framework*, propio o externo). Con un *framework* se dispone de mucha más flexibilidad y un mayor control sobre el proyecto, mientras que con un CMS uno tiene que adherirse a como esté estructurada la aplicación y "construir sobre ella". Esto puede limitar las posibilidades de un CMS, pero a su vez este nos proporciona un enorme número de funcionalidades ya programadas (*out-of-the-box)*, lo que puede repercutir en un enorme ahorro de tiempo.

Cuando hay que elegir una tecnología para realizar un proyecto de desarrollo, hay que sopesar muchas cosas: tiempo, esfuerzo, rendimiento, capacidades intrínsecas de la tecnología, etc. antes de tomar una decisión.

A continuación, se van a exponer *frameworks* y gestores de contenido que se consideraron opciones viables para el desarrollo de la plataforma y la API REST.

3.2.1. Los frameworks.

Un *framework* simplifica el desarrollo de las aplicaciones, ya que automatiza muchos de los patrones utilizados para resolver las tareas comunes. Además, un *framework* proporciona estructura al código fuente, forzando al desarrollador a crear código más legible y más fácil de mantener. Por último, un *framework* facilita la programación de aplicaciones, ya que encapsula operaciones complejas en instrucciones sencillas.

3.2.1.1. Symfony.

Symfony es un completo *framework* diseñado para optimizar, gracias a sus características, el desarrollo de las aplicaciones web. Para empezar, separa la lógica de negocio, la lógica de servidor y la presentación de la aplicación web, que es un patrón de arquitectura de software denominado MVC (Modelo-Vista-Controlador, figura 3.10).

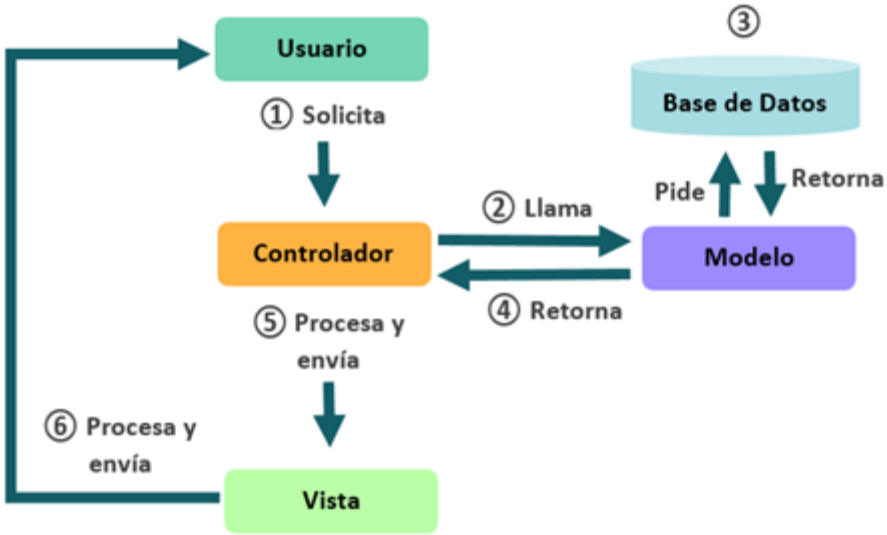


Figura 3.10 – Arquitectura Modelo Vista Controlador (Con Base de Datos como sistema de persistencia).

Proporciona varias herramientas y clases encaminadas a reducir el tiempo de desarrollo de una aplicación web compleja. Además, automatiza las tareas más comunes, permitiendo al desarrollador dedicarse por completo a los aspectos específicos de cada aplicación. El resultado de todas estas ventajas es que no se debe reinventar la rueda cada vez que se crea una nueva aplicación web.

Symfony está desarrollado completamente con PHP y ha sido probado con éxito en sitios como Yahoo! Answers, delicious, DailyMotion y muchos otros sitios web de primer nivel (Symfony, 2015). Es compatible con la mayoría de gestores de bases de datos, como MySQL, PostgreSQL, Oracle y SQL Server de Microsoft. Aunque no se compromete con ninguna tecnología de mapeo objeto-relacional (ORM – *Object-Relational Mapping*), el paquete *Symfony Standard Edition* viene con Doctrine como

sistema ORM (Doctrine, 2015). La combinación Symfony-Doctrine está enormemente extendida debido a la fabulosa simbiosis que existe entre ambas.

Symfony se diseñó para que se ajustara a los siguientes requisitos:

- Fácil de instalar y configurar en la mayoría de plataformas (y con la garantía de que funciona correctamente en los sistemas Windows y Unix estándares).
- Sistema de ruteo independiente, con la posibilidad de ofrecer servicios REST. El sistema de ruteo es un mecanismo encargado de simplificar las URLs mediante reglas para facilitar su visualización sin perder la funcionalidad (Symfony, 2016). Básicamente permite convertir URLs de la forma index.php?article_id=57 a favor de algo como /read/intro-to-symfony.
- Independiente del sistema gestor de bases de datos.
- Sencillo de usar en la mayoría de casos, pero lo suficientemente flexible como para adaptarse a los casos más complejos.
- Basado en la premisa de "convenir en vez de configurar", en la que el desarrollador solo debe configurar aquello que no es convencional.
- Sigue la mayoría de mejores prácticas y patrones de diseño para la web (Martin, 2009).
- Preparado para aplicaciones empresariales y adaptable a las políticas y arquitecturas propias de cada empresa, además de ser lo suficientemente estable como para desarrollar aplicaciones a largo plazo.
- Código fácil de leer que incluye comentarios de phpDocumentor y que permite un mantenimiento muy sencillo.
- Fácil de extender, lo que permite su integración con librerías desarrolladas por terceros.

3.2.1.2. Laravel.

Laravel es un *framework* de código abierto para desarrollar en PHP, con una filosofía muy clara enfocada a que el código sea lo más expresivo y elegante posible.

El que esté enfocado en el código, no le quita la versatilidad y potencia que cualquier *framework* actual pueda tener, además de tener una arquitectura interna siguiendo patrones de diseño como pueden ser el patrón de Fachada (*Facade*) o el principio de inyección de dependencias (*Dependency Injection*).

Laravel aprovecha las mejoras de las últimas versiones de PHP como son los *namespaces* y *closures*, al igual que utiliza el software **composer** para la instalación y la gestión de las librerías (Composer, 2016).

Algunas características de **Laravel:**

- Proporciona un sistema de ruteo basado en el uso de *closures*, con la posibilidad de que sea RESTful.
- Utiliza Blade como motor de plantillas.
- Consultas SQL creadas dinámicamente con Fluent.
- Utiliza su propio sistema ORM: Eloquent.
- Utiliza el gestor de dependencias Composer.
- Soporte para el caché.
- Arquitectura MVC.
- Usa componentes de Symfony, como *Routing* o *EventDispatcher*.
- Adopta las especificaciones PSR-2 y PSR-4 (PHP FIG, 2015b).

Laravel está anclado a **Symfony** por varios de sus componentes y por ello, al menos de momento, su evolución está ligada a la evolución de **Symfony**.

3.2.1.3. Zend Framework.

Es un framework de código abierto para desarrollar aplicaciones y servicios web con PHP 5.

Todos sus componentes están diseñados siguiendo el paradigma de orientación a objetos. En la estructura de los componentes de **Zend**, cada componente está construido con una baja dependencia de otros componentes siguiendo el principio de inversión de dependencias. Esta arquitectura débilmente acoplada permite a los desarrolladores utilizar los componentes por separado. A menudo se hace referencia a este tipo de diseño como "*use-at-will*" (uso a voluntad).

Aunque se pueden utilizar de forma individual, los componentes de la biblioteca estándar de **Zend Framework** conforman un *framework* de aplicaciones web al combinarse.

Entre las características a destacar de **Zend Framework** tenemos:

- Arquitectura MVC.
- Capa de abstracción a la base de datos.
- Modular y extensible mediante el uso de componentes.
- Creado mediante componentes independientes con un bajo nivel de acoplamiento entre ellos, por ejemplo, un componente de formularios que implementa la presentación de formularios HTML, validación y filtrado, componentes que proveen autentificación de usuarios, componentes para gestionar servicios web, etc.

Sea cual sea el problema, posiblemente exista un componente de Zend que permita reducir el tiempo de desarrollo.

3.2.2. Gestores de contenido (*Content Management System*).

Un gestor de contenidos es una aplicación (generalmente web) que proporciona la capacidad de que múltiples usuarios con diferentes niveles de permisos puedan gestionar de forma conjunta el contenido, datos o información de un sitio web, aplicación web o de una intranet.

Por gestión de contenidos se entiende la capacidad de crear, editar, archivar, publicar, colaborar, informar o distribuir contenido, archivos o información de una aplicación.

3.2.2.1. Concrete5.

El proyecto **Concrete5** se comenzó a gestar en 2003, aunque no fue hasta 2008 cuando entró en escena mediante versiones estables. Desde entonces ha ido dando pasos de gigante para hacerse un hueco en el mundo de los sistemas de gestión de contenidos.

Frente a los grandes CMSs del mercado como **Wordpress**, **Joomla**! o **Drupal**, Concrete5 se posiciona como una alternativa casi tan fácil de implementar por el experto y de utilizar por el usuario como WordPress, pero con una serie de funcionalidades que lo hacen algo más complejo de partida.

Funcionalidades de Concrete5 (Concrete5, 2015)

El sistema cuenta con todas aquellas funcionalidades propias de los sistemas de gestión de contenidos: creación/edición de contenidos, flujo de trabajo, integración de información externa, gestión de los usuarios, la creación del repositorio y su publicación.

En cuanto a la creación y edición de contenidos, Concrete5 permite trabajar directamente sobre las páginas que los albergan, sin necesidad de acceder a una interfaz de administración separada. Las áreas editables de un nodo, llamadas bloques (*blocks*), se encuentran definidas en la plantilla que se está utilizando. Cada bloque tiene capacidad para contener diferentes tipos de contenidos: texto, imágenes, carruseles de imágenes, comentarios, mapas, etc.

Paralelamente a los bloques existen los *stacks*. Un *stack* se forma a partir de la agregación de diferentes contenidos disponibles para los bloques, que posteriormente se podrán reutilizar en grupo en cualquier cantidad de páginas de nuestro portal.

Los puntos más fuertes de Concrete5 son la gestión de usuarios y el sistema de flujos de trabajo. Para la gestión de usuarios y permisos se tiene total flexibilidad para crear tipos de usuarios y asociarlos a un perfil y capacidades determinadas. También podemos crearlos con fecha de caducidad, es decir, determinar a partir de qué día ese grupo de usuarios dejará de tener esos privilegios. Desde el panel de control de permisos avanzados, es posible configurar los permisos relacionados con el acceso al sitio, la gestión de archivos, tareas, usuarios, mantenimiento del sitio, etc. El sistema de flujos de trabajo de Concrete es otro de sus puntos fuertes. Permite la creación y parametrización de flujos de trabajo personalizados por grupos de usuarios (administradores, usuarios registrados, invitados, …), usuarios individuales o grupos personalizados. A cada uno de estos grupos o usuarios individuales, se les puede asociar una fecha de inicio y otra de finalización tras la cual quedarán fuera del flujo de trabajo al que estaban asociados.

Las actualizaciones del sistema, así como las de los *plugins*, son totalmente automáticas y se pueden ejecutar desde la interfaz del sistema.

3.2.2.2. Liferay.

Liferay es un gestor de contenidos web de código abierto escrito en Java. Su origen se remonta al año 2000 y cuenta con un importantísimo equipo de desarrollo (Liferay, 2015b). Además, hay numerosas organizaciones y empresas que se dedican a crear módulos y ampliaciones de las especificaciones básicas del gestor **Liferay**. Es un CMS tan complejo como otros realizados en PHP (como **Wordpress** o **Drupal**)**,** pero programado en Java. Mientras que un CMS hecho con PHP necesita un servidor Apache, **Liferay** necesita un servidor adecuado para Java (Tomcat, Glassfish, etc.)

Aunque se puede hacer cualquier cosa que se necesite, su uso más habitual es la creación de portales web. Es especialmente utilizado por los Ayuntamientos, Diputaciones y webs de las Administraciones Públicas (Liferay, 2015a). Por ejemplo, el sistema de Espacios de Colaboración del Centro de Transferencia de Tecnología está desarrollado con Liferay (Administración Electrónica del Gobierno de España, 2014). Si no se necesita cambiar código o utilizar funcionalidades de alto nivel, su uso es muy sencillo con un sistema *drag & drop* que permite adjuntar contenido de forma dinámica. Estos contenidos o *Portlets* son módulos que ofrecen una funcionalidad en concreto.

Además de para portales públicos, **Liferay** también se utiliza para portales de clientes, aunque no siempre es la mejor opción debido al coste. Otro aspecto positivo es que están muy bien implementados los permisos y las zonas seguras. De esta forma, **Liferay** ofrece gran estabilidad en los portales que requieran de una gestión de permisos

de usuario muy granular. En definitiva, permite discriminar de forma muy sencilla quién tiene que ver qué información.

Por otra parte, las últimas versiones de **Liferay** utilizan **Bootstrap**, un *framework front-end* para gestionar el diseño de sitios web y aplicaciones, que permite crear plantillas interactivas y *responsive* (Bootstrap, 2016).

Liferay es uno de los mejores gestores de contenido del mercado, pero requiere de desarrolladores competentes en Java para poder añadir funcionalidades. Además, hay que tener en cuenta que requiere de un servidor Java como Tomcat o Glassfish. Estos servidores son más costosos de desplegar y de mantener que, por ejemplo, los servidores Apache para aplicaciones PHP, lo que muchas veces un factor importante que hace que se opte por otras tecnologías menos costosas para el desarrollo de las aplicaciones.

3.2.2.3. Drupal.

Drupal es un sistema de gestión de contenido modular y muy configurable.

Es un programa de código abierto, con licencia GNU/GPL, escrito en PHP, desarrollado y mantenido por una activa comunidad de usuarios. Destaca por la calidad de su código y de las páginas generadas, el respeto de los estándares de la web, y un énfasis especial en la usabilidad y consistencia de todo el sistema (Drupal, 2015a).

El diseño de Drupal es especialmente idóneo para construir y gestionar comunidades en Internet. No obstante, su flexibilidad y adaptabilidad, así como la gran cantidad de módulos adicionales disponibles, hace que sea adecuado para realizar muchos tipos diferentes de sitio web.

El sitio principal de desarrollo y coordinación de Drupal es drupal.org, en el que participan activamente varios miles de usuarios de todo el mundo.

Funcionalidades de Drupal (Drupal, 2015a)

- Contenido flexible.

Puede definir campos personalizados que podrán ser utilizados en tipos de contenido, usuarios, comentarios, términos y otras entidades, tanto si se almacenan los datos de esos campos en SQL, NoSQL o si se utiliza almacenamiento remoto.

- Mejor diseño de plantillas.

Permite controlar exactamente qué se muestra en pantalla con la nueva *Render* API (Drupal, 2015c) y algunos *hooks* (un *hook* en Drupal es una función que permite alterar el comportamiento de secciones de código sin necesidad de tener que modificar dicha sección directamente) drásticos para modificaciones. El nuevo módulo RDF (*Resource Description Framework*) (W3C, 2014) provee marcado semántico para la web.

- Accesible.

Las pantallas de administración son ahora mucho más accesibles. Las abundantes mejoras en la interfaz le facilitan la construcción de páginas web altamente accesibles.

- Imágenes y ficheros.

El soporte de imágenes en el contenido está ahora incorporado en el núcleo. Permite generar versiones diferentes para *thumbnails*, vistas previas y otros estilos de imágenes. También es posible utilizar las gestiones privada y pública de ficheros al mismo tiempo.

- *Testing* automático del código.

Un nuevo entorno de *testing* automatizado, con más de 30.000 *tests* incluidos por defecto, permite realizar *tests* para integración continua de todos los parches al núcleo de Drupal y a los módulos contribuidos.

- Soporte de base de datos mejorado.

Una nueva capa de abstracción de base de datos provee soporte para SQLite, MySQL/MariaDB y PostgreSQL *out-of-the-box*. Puede instalar módulos contribuidos para utilizar MS SQL Server, Oracle, y más.

- Mejor soporte para distribuciones.

Permite utilizar perfiles de instalación para distribuir un producto personalizado basado en Drupal. Una nueva API y configuración exportable permite capturar más opciones en código.

- Extensible.

Gracias a un enorme esfuerzo de la comunidad, más de 800 módulos están disponibles o bajo desarrollo activo para Drupal 7, incluyendo *Views*, *Pathauto*, y WYSIWYG, con muchos otros en el camino de actualizarse cada día.

3.3. Comparación de tecnologías empleadas.

En este apartado compararemos las diferentes opciones de tecnologías que podríamos utilizar para la realización del simulador, y finalmente se elegirán las empleadas.

3.3.1. Motores gráficos.

Suele ser común representar los rasgos o características de distintas opciones en una tabla que permita comparar rápidamente las características de un solo vistazo. En este proyecto hemos optado por evitar este tipo de tablas. Aunque sintetizar es bueno, en ocasiones no aporta lo necesario para tener una visión real de la situación. En ocasiones la herramienta con las mejores características no es la más adecuada para resolver nuestro problema.

Teniendo en cuenta nuestro problema, que consiste en crear un simulador en un tiempo limitado y de forma unipersonal, tenemos que descartar casi todos los motores, incluso aunque sean más potentes o más eficientes. Al necesitar un motor 3D tenemos que descartar todos los que no lo sean. Además, tienen que tener una curva de aprendizaje que permita realizar el proyecto en un número de horas limitado (150 h.). Esto descarta el motor *CryEngine* debido a su enorme curva de aprendizaje.

C++ es un lenguaje muy potente, pero de menos nivel que por ejemplo C#. Por ello hemos llegado a que Unity sea posiblemente la opción más obvia para nuestra situación.

En el mundo del desarrollo no existe una herramienta para todo, pero sí que hay una herramienta para cada problema, lo difícil es tomar la decisión.

En concreto, elegir Unity como motor solo se debe a una cuestión de comodidad. El lenguaje en el que se programa Unity, C#, tiene nociones muy similares a Java siendo un lenguaje tipado de alto nivel con unas funcionalidades muy diversas. Se dispone de un enorme número de componentes con los que extender su funcionamiento y un gestor de dependencias asombroso como es *Nuget*.

Por otro lado, *Unreal* es un motor muy utilizado en la industria del desarrollo de videojuegos, pero requiere de una curva de aprendizaje mayor, incluso para sus componentes más sencillos.

A fecha de 2016, *Unity* pretende ser un motor para todo (*jack-of-all-trades*) capaz de adaptarse a la mayoría de situaciones requeridas en un entorno que no requiera de gráficos hiperrealistas. En cualquier caso, la última versión de Unity, Unity 5 pretende enfrentarse a nivel tecnológico con los grandes del mercado incorporando *Enlighten* como sistema de iluminación, la creación del componente UI para las interfaces de usuario o el diseño de *Mecanim* para controlar las animaciones y los eventos. Ahora es un buen momento para darle una oportunidad a *Unity*.

3.3.2. Tecnologías web.

Este es un tema que por sí solo podría ocupar varios proyectos. En el punto 3.2.1 se han expuesto tres de los *frameworks* de PHP más utilizados en el mundo actual del desarrollo software y más en concreto en el desarrollo de aplicaciones web, con sus características y funcionalidades. Por otra parte, en el punto 3.2.2 se han mencionado unos gestores de contenido que también podrían ser viables para la solución a nuestro problema.

Symfony es un *framework* montado en PHP que sigue una programación orientada a objetos y que está estructurada por componentes, al igual que **Zend**. La filosofía de estructuración por componentes es a lo que tiende el futuro del desarrollo software (Lockahrats). Se pretende construir las aplicaciones utilizando componentes funcionales y ensamblados entre sí que permiten obtener lo que se necesita, pero siempre de una manera extensible. Frente a **Symfony** o **Zend**, **Laravel** se presenta como un *framework* con una curva de aprendizaje menor y más orientado a conseguir el código más elegante, lo que le hace uno de los *frameworks* más solicitados por las empresas de desarrollo. Su sistema de ruteo RESTful permitiría crear la API REST que se ha creado en este proyecto de forma ágil y rápida.

Cuando se iba a seleccionar la tecnología a utilizar, había que tener en cuenta que esto era un añadido al núcleo del proyecto y por lo tanto debía ser algo que proporcionara el mayor contenido directamente sin tener que desarrollarlo punto por punto. Esto nos llevó a descartar los *frameworks* y centrarnos en los gestores de contenido.

Hemos optado por **Drupal** por ser un *framework* conocido y con el que había trabajado a nivel profesional, lo que me permitiría mejorar los conocimientos y aptitudes frente al mismo. Sin embargo, varios de los otros expuestos habrían sido perfectamente válidos. **Liferay** es un proyecto gigantesco, pero al estar montado en Java requerirá de servidores más costosos de mantener que el típico *stack* LAMP (Linux-Apache-MySQL-PHP). Concrete5 quizás se quedara corto en cuanto a las funcionalidades que se pretendían conseguir a futuro.

A diferencia de CMSs como **Wordpress**, que ha estado siguiendo siempre un sistema de actualización creciente y que, en parte, ha llevado al enquistamiento de la evolución de núcleo, otros CMSs como **Drupal** han intentado evolucionar en cada una de sus nuevas versiones, de tal manera que en su versión 8 ha decidido utilizar componentes de **Symfony** para su desarrollo y pretende orientar su paradigma de programación de uno estructurado a uno orientado a objetos, siguiendo la línea por la que van las aplicaciones modernas. En cualquier caso, Drupal ha buscado siempre mantener una estructura modular que permitiera la creación de módulos funcionales razonablemente independiente entre sí con los que extender su núcleo para que pudiera satisfacer todo tipo de necesidades. Veremos en el apartado 5 cómo ha sido posible crear un módulo que sirva una API REST

para transmisión de datos entre aplicaciones remotas que utilizan distintas tecnologías y lenguajes de programación.

Como resumen, se ha optado por **Drupal** por comprensión de la tecnología, por poder resolver el problema en menos tiempo que con otras alternativas y porque nos permitía centrarnos solo en el desarrollo de nuestra funcionalidad dejando que el CMS gestione otros temas como los permisos de usuario o la maquetación. Si el núcleo del proyecto hubiera sido crear la aplicación web, posiblemente se habría optado por utilizar un *framework* y haber desarrollado todas las funcionalidades.

Personalmente, habría optado por usar **Laravel** por ser un *framework* que busca resolver problemas, pero haciéndolo con elegancia, buscando obtener aplicaciones con código limpio y legible. Si el proyecto se previera que fuera a ser de gran envergadura, entonces habría optado sin dudarlo por **Symfony** y habría usado **Twig** como motor de plantillas y **Doctrine** como ORM.

3.3.3. Programas de modelado 3D.

3ds Max es uno de los programas de modelado 3D más utilizado y conocido. Tiene unas características muy buenas, aunque su precio es bastante elevado. Además, cuenta con una interfaz intuitiva que permite que su curva de aprendizaje no sea demasiado alta.

La ventaja de *Blender* respecto a sus oponentes es que es un programa completamente gratuito. Aunque la curva de aprendizaje de *Blender* es alta, cada vez más empresas optan por aportar su granito de arena utilizando este software.

Teniendo en cuenta que el uso de estos programas va a ser mínimo se ha optado por *Blender* debido a ser gratuito. Posiblemente en otra situación se habría optado por 3dsMax o Maya ya que son programas con más salida en la industria, más fáciles de aprender y más utilizados en el mundo empresarial, además de tener una documentación de muy alto nivel.

4. Desarrollo del simulador de conducción.

En primer lugar, es necesario instalar Unity. Desde la página oficial de Unity se puede descargar un instalador que permite instalar, aparte del editor, cada una de las posibles plataformas a las que exportar los juegos.

En este caso, se ha trabajado con Unity 5 y se ha actualizado y adaptado hasta la versión 5.3.5. Además, se ha instalado el soporte para sacar aplicaciones *stand-alone* para Windows.

4.1. Flujo de trabajo en Unity.

Se va a presentar de forma somera cómo es el flujo de trabajo en el desarrollo con Unity (Blackman, 2011). Esto permitirá que sea más fácil entender cada uno de los puntos expuestos respecto al desarrollo del simulador en sí mismo.

4.1.1. Interfaz.

Interfaz del editor.

La ventana principal del editor está compuesta por varios paneles que contienen pestañas. Cada pestaña contiene una funcionalidad propia de Unity y pueden desplazarse libremente entre un panel o ventana y otro.

Por lo tanto, la apariencia visual del editor se puede adaptar al gusto de cada usuario.

A continuación, en la figura 4.1, se puede ver la disposición por defecto del editor.

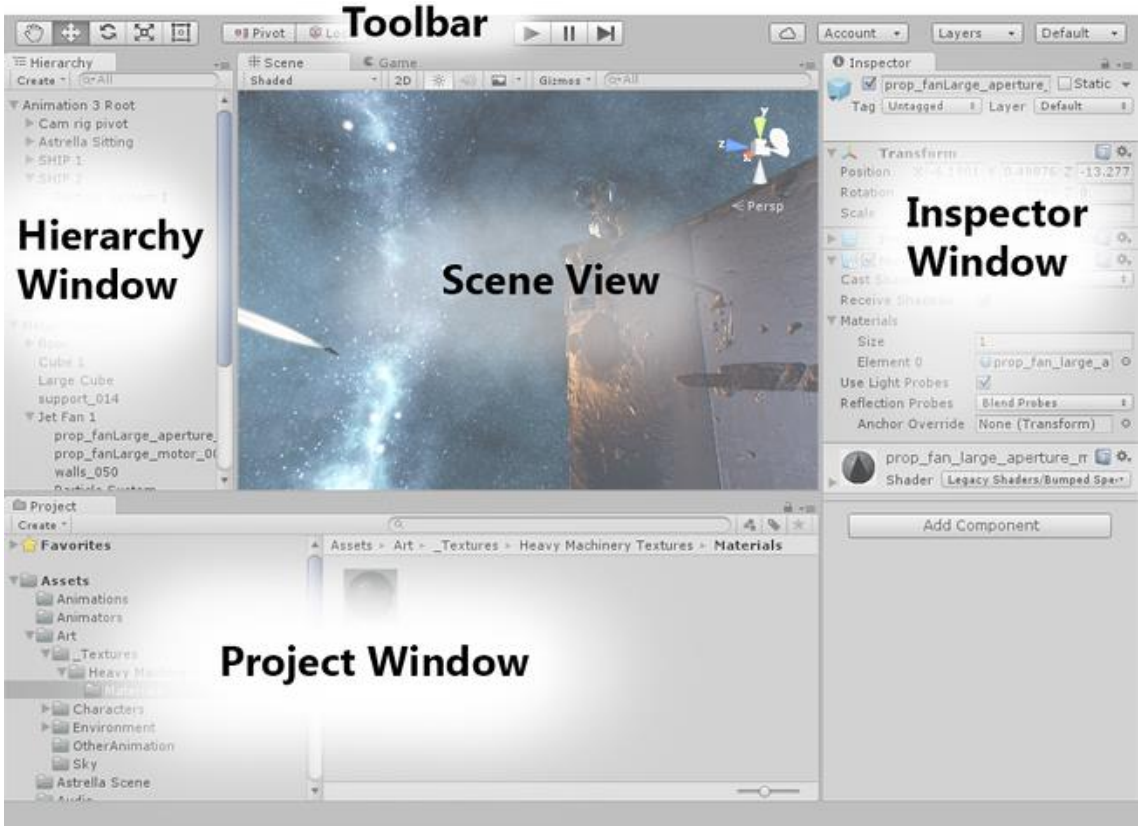


Figura 4.1 – Interfaz del editor de Unity.

La ventana del proyecto.

La ventana del proyecto muestra la librería de *assets* disponibles para el proyecto. Cada vez que se importan nuevos *assets* estos se muestran aquí. La organización de las carpetas es libre, aunque existen algunas carpetas con significados especiales como Editor o *Resources*. En la figura 4.2 se puede ver esta ventana:

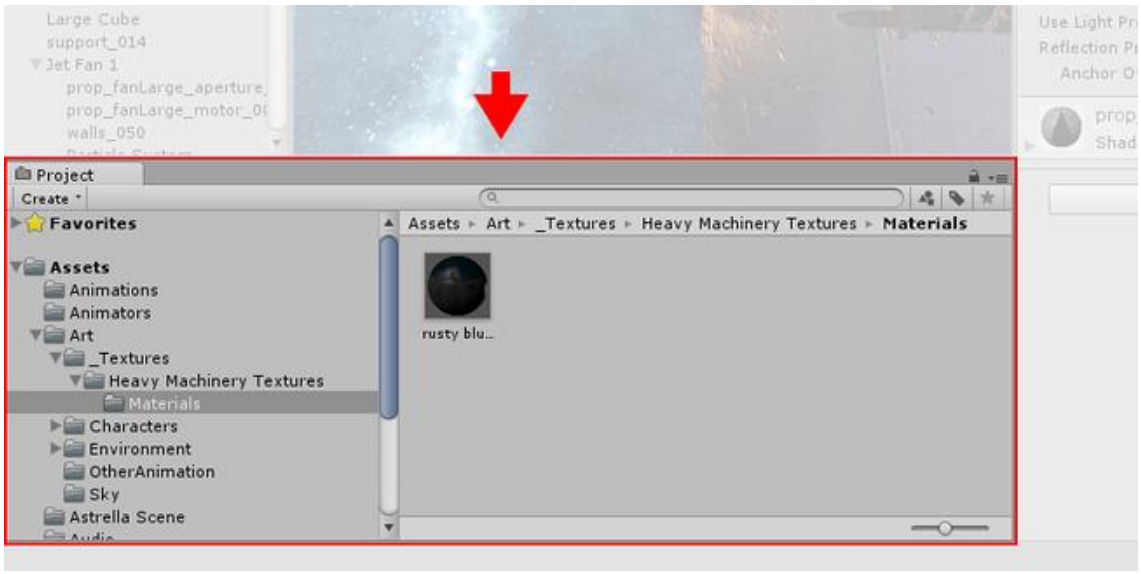


Figura 4.2 – Ventana de proyecto.

La vista de escena.

Esta vista permite navegar visualmente por una Escena (que es la representación de un nivel en Unity) y editar cada uno de los distintos *GameObjects* (esta palabra tiene sentido por sí misma al hacer referencia a una clase y no será traducida). Esta es una de las vistas más importantes del editor permitiendo seleccionar *GameObjects*, modificar su posición y rotación, etc.

Se puede ver un ejemplo de esta ventana en la figura 4.3:

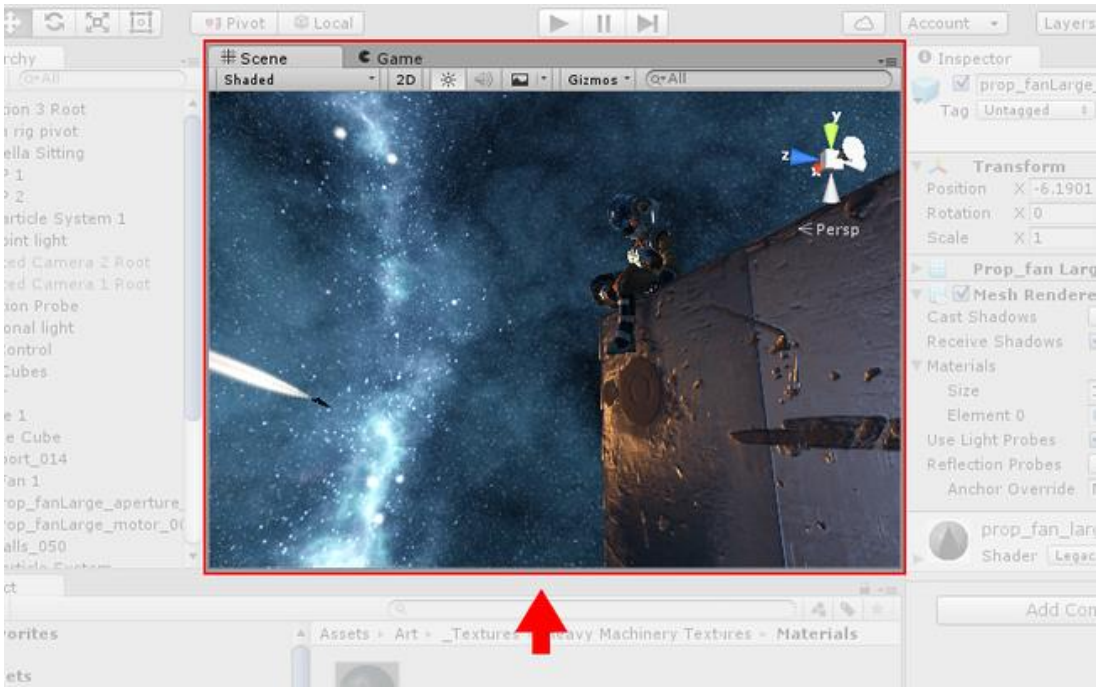


Figura 4.3 – La vista de escena.

La ventana de jerarquía.

Esta ventana es una representación textual jerárquica de cómo los *GameObjects* están organizados en la escena. En Unity todos los elementos de una escena son *GameObjects* y el sistema de jerarquías permite agruparlos a voluntad del programador. Como es una representación textual de la ventana de escena, ambas están relacionadas uno a uno.

Se puede ver un ejemplo en la figura 4.4.

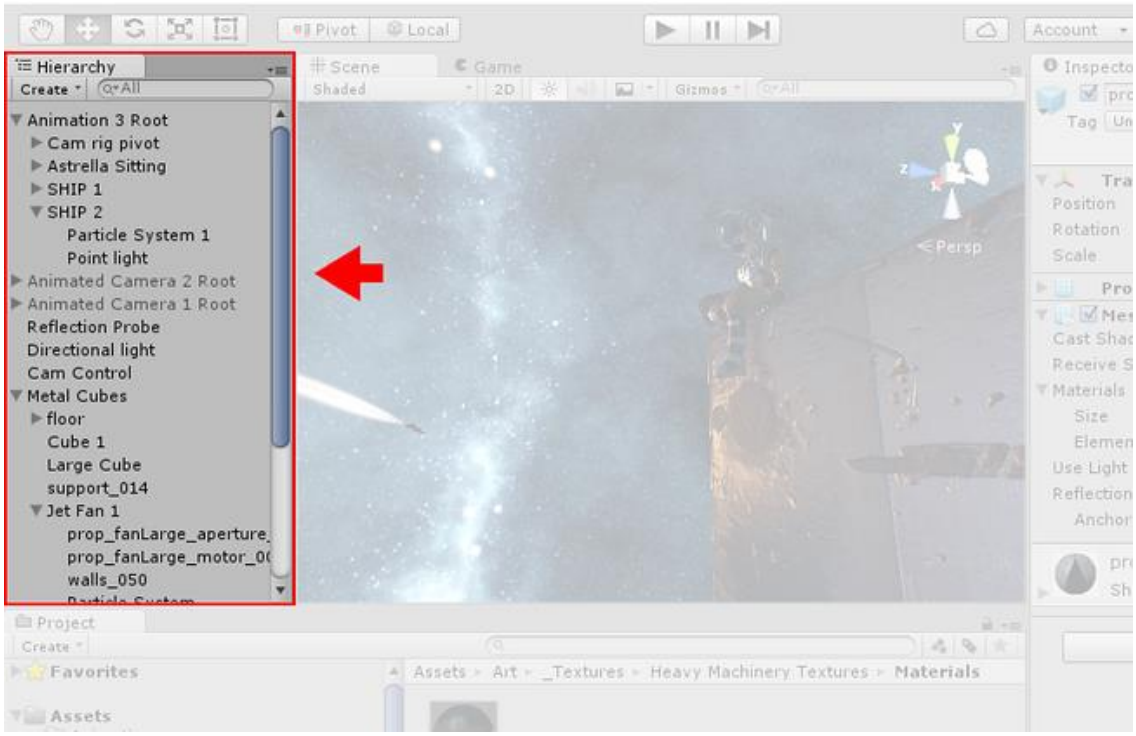


Figura 4.4 – Ventana de jerarquía.

La ventana del inspector.

El inspector permite identificar las propiedades de un elemento seleccionado en la jerarquía. Cada *GameObject* puede tener distintos componentes asociados y cada componente será representado de forma distinta en el editor. Desde el inspector se pueden añadir o modificar componentes y también modificar los parámetros expuestos por este componente al exterior (propiedades serializadas de los componentes).

En la ventana del inspector que se ve en la figura 4.5 se puede apreciar que el *GameObject* seleccionado tiene tres componentes asociados:

- Componente *Transform*: Inherente a los *GameObject*. Indica posición, rotación y escala del elemento en la escena.
- Componente *Mesh*: Permite enganchar una malla tridimensional a un elemento.
- Componente *MeshRenderer*: Indica cómo hay que renderizar la malla asociada. Principalmente permite asociar un material a una malla. Como aclaración, un material indica de qué forma se debe *renderizar* una superficie y hace referencia a texturas, tintes de color, etc. Las opciones disponibles de un material dependen del *Shader* que está utilizando.

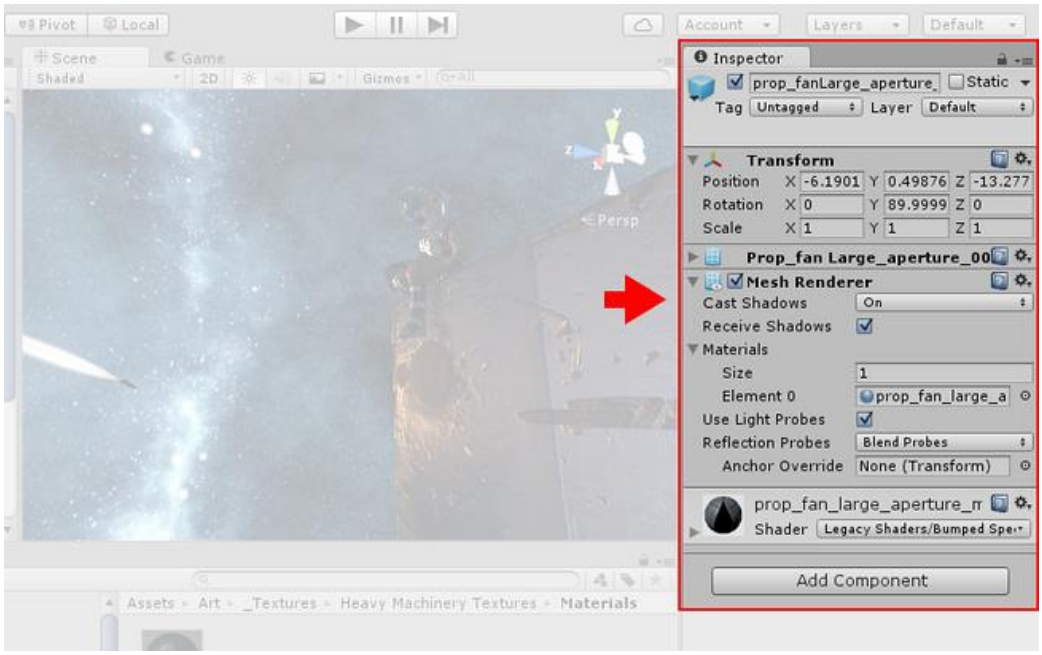


Figura 4.5 – La ventana del inspector.

La barra de herramientas.

La barra de herramientas contiene las funcionalidades básicas para trabajar con Unity. En la zona izquierda se tienen herramientas para manejar elementos en la escena. En la zona central se tienen los botones para compilar y entrar en modo *debug* en la escena, un botón de pausa y un botón para avanzar *frame* a *frame*. En la zona de la derecha se puede acceder a algunas funcionalidades como acceso a tu cuenta de Unity o tener acceso a las capas (*layers*) de tu aplicación. El diseño de la barra de herramientas se puede ver en la figura 4.6.

Figura 4.6 – Barra de herramientas.

4.1.2. Escena, GameObjects y Componentes

Escenas.

Las escenas contienen los objetos de la aplicación. Las escenas pueden ser usadas para crear menús o niveles individuales. En cada escena habrá que poner todos los elementos necesarios para construir un nivel y sus relaciones entre ellos, por ejemplo, elementos del entorno como árboles, o jugables como vehículos o personajes (Creighton, 2010). En las figuras 4.7 y 4.8 se pueden ver una escena recién creada y una escena finalizada del proyecto respectivamente.

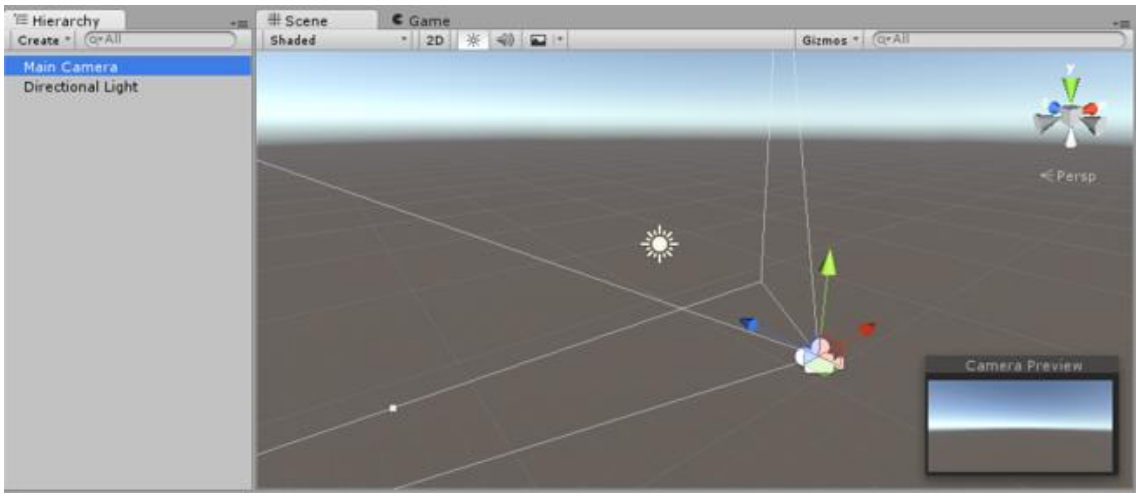


Figura 4.7 – Escena vacía.

Figura 4.8 – Escena con objetos del proyecto.

Para almacenar la escena con la que se está actualmente trabajando hay que ir a **File > Save Scene**. Las escenas se almacenan como *assets* en carpeta *Assets* del proyecto y por lo tanto aparecen en la ventana de proyecto, al igual que cualquier otro *asset*.

Para abrir una escena basta con hacer doble click sobre ella en la ventana de proyecto.

Como curiosidad, desde la versión 5 de Unity es posible trabajar con varias escenas a la vez sin necesidad de cargarlas de una en una. Para ello hay que cargar la escena en modo aditivo como se ve en la Figura 4.9:

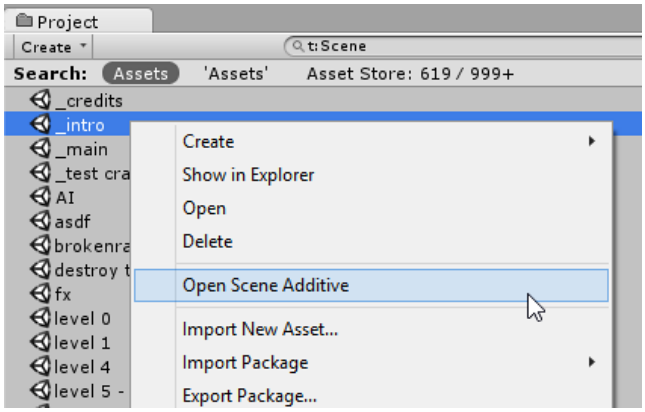


Figura 4.9 – Añadiendo múltiples escenas.

GameObjects.

Como se ha mencionado, los *GameObjects* son los objetos fundamentales en Unity que representan personajes, elementos o partes del escenario. Todo en una escena de Unity es un *GameObject* pero estos por sí mismos no consiguen nada, sino que actúan como contenedores de Componentes, que son los que implementan la funcionalidad real.

Si, por ejemplo, se quiere tener una luz, no vale con crear un *GameObject*, hay que asociarle un componente *Light* a este objeto, tal y como se puede ver en la figura 4.10.

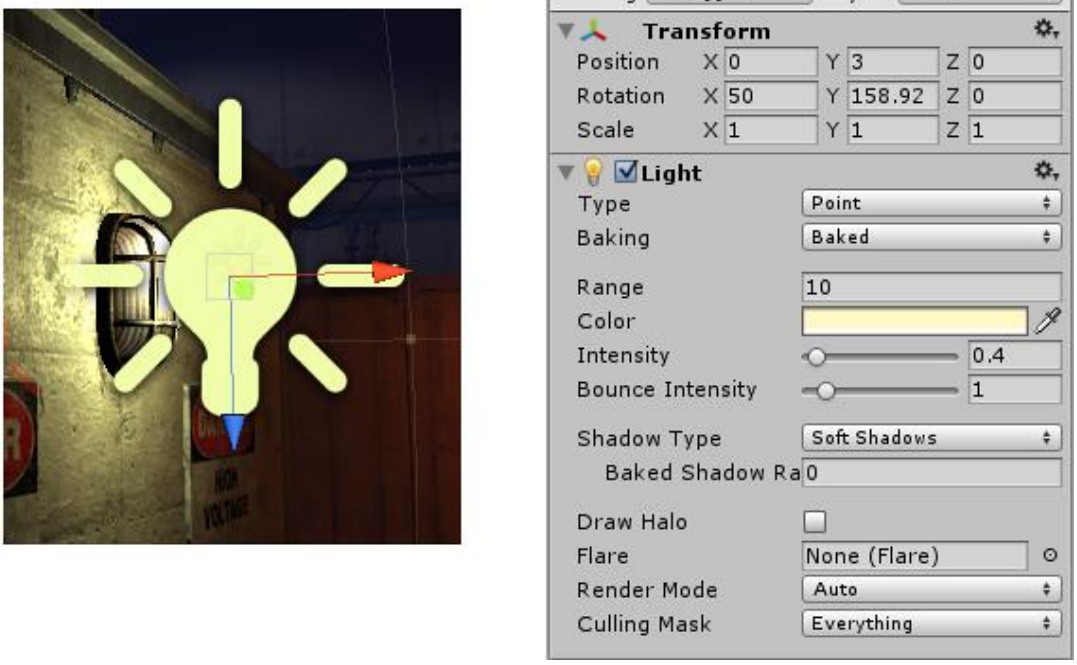


Figura 4.10 – Creando una luz.

Otro caso se puede ver cuando se genera un Cubo con Unity. Un cubo sólido en Unity es un *GameObject* que tiene asociados un *MeshFilter* y un *MeshRenderer* que se encargan de dibujar la superficie del cubo, y un componente *BoxCollider* que representa el volumen sólido de un objeto en términos de físicas. En la figura 4.11 se observa este caso del cubo con sus componentes (Mesh Filter, Mesh Renderer y Box Collider).

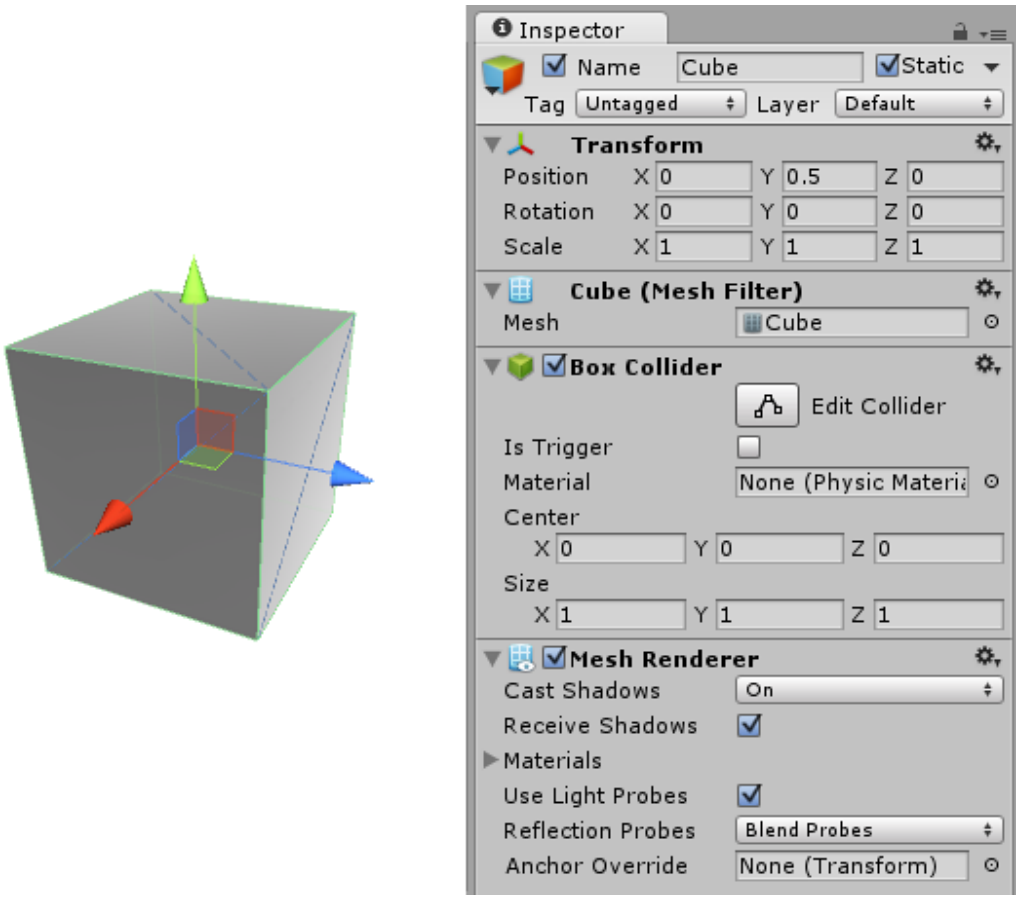


Figura 4.11 – GameObject genérico de un cubo.

Un *GameObject* tiene un nombre y puede tener asociada una *Tag* y pertenecer a una capa o *Layer*. Tanto *Tags* como *Layers* permiten referenciar objetos desde el código de un script de forma más eficiente que una búsqueda mediante el nombre.

Un *GameObject* siempre tiene un componente *Transform* para representar la posición y orientación del objeto en la escena y no puede eliminarse. El resto de componentes pueden ser añadidos o eliminados tanto desde el menú como desde el código.

Componentes.

Los componentes son el punto clave del funcionamiento de los objetos y de su comportamiento en las aplicaciones creadas con Unity. En definitiva, los componentes definen las funcionalidades de un objeto de forma modular. La creación de componentes independientes y modulares es una de las partes más complejas durante el desarrollo de un juego.

Como se ha mencionado en el apartado anterior, un *GameObject* vacío viene con un componente *Transform* asociado y que es imposible de eliminar. Para asociar, modificar o eliminar nuevos componentes hay que seleccionar el objeto, lo que nos permite ver todas las funcionalidades actualmente asociadas al mismo desde el inspector y actuar sobre cada una de ellas por separado.

Para añadir nuevos componentes a un objeto, se puede hacer desde el menú de componentes o desde el inspector seleccionando el botón *Add component* que muestra

una ventana por la que navegar por los distintos componentes del proyecto (Figura 4.12).

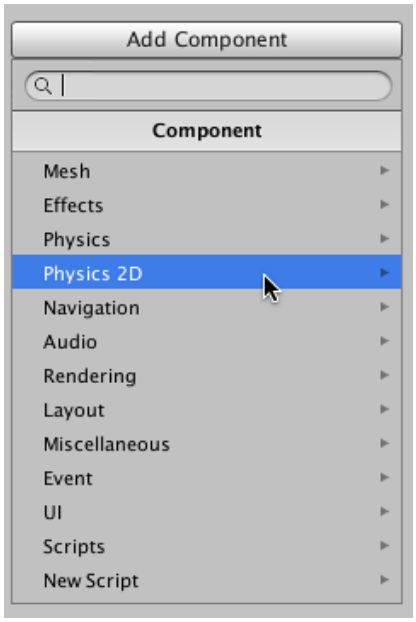


Figura 4.12 – Navegador de componentes.

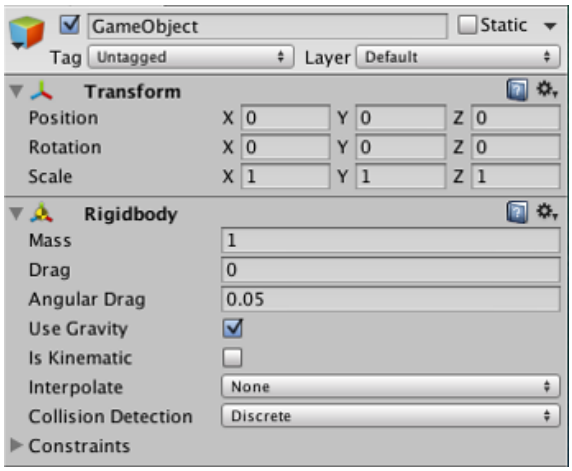


Figura 4.13 – *GameObject* con un componente Rigidbody asociado.

La figura 4.13 muestra el componente Rigidbody asociado un objeto. Dicho componente se encarga de otorgar masa al objeto dentro del motor de físicas.

Se puede añadir cualquier combinación de componentes a un GameObject, aunque algunos requieren de otros para funcionar.

Crear componentes propios – Los Scripts.

Se ha mencionado anteriormente la palabra *script*. Un *script* es una funcionalidad programada en C# o *UnityScript* y diseñada para funcionar en Unity.

Cuando creas un script y lo añades a un *GameObject*, este script aparece en el inspector de objetos como un componente más. Esto se debe a que los scripts se convierten a Componentes cuando son almacenados. Básicamente un script compila como un tipo de Componente y es tratado como tal por el motor de Unity.

4.1.3. Los *Prefabs.*

Es conveniente construir los objetos de la aplicación directamente en la escena añadiendo los componentes y modificando las propiedades asociadas a los mismos desde el inspector. Sin embargo, esto puede ocasionar problemas cuando, por ejemplo, se tienen objetos como un NPC (*No Player Character* – Personaje no jugable) o un objeto de entorno que va a ser reutilizado múltiples veces en una escena. Obviamente, se podrían realizar tantas copias del objeto como fuera necesario pero cada uno sería independiente del resto. Generalmente se desea que todas las instancias de un objeto en concreto tengan las mismas propiedades, de tal forma que lo deseable sería que cuando se modifica un objeto no haya que hacerlo en todos los demás manualmente (conllevaría tiempo y problemas).

Unity dispone de un tipo de *asset* denominado *Prefab* que permite almacenar un *GameObject* completo con sus relaciones, componentes y propiedades. Un *Prefab* actúa como una plantilla de la que se pueden crear nuevos objetos en la escena, de tal forma que cualquier modificación realizada en el *asset* del *Prefab* se ve reflejada en todas sus instancias. Eso no impide que se puedan alterar componentes y propiedades en cada instancia en concreto del *Prefab*.

Se puede crear un *Prefab* desde el menú **Asset > Create Prefab** y arrastrando el objeto deseado desde la escena al *Prefab* vacío. Arrastrar el *Prefab* desde la vista del proyecto a la escena creará una instancia del mismo. Los objetos que son instancias de *Prefabs* salen en la vista jerárquica como azules. También es posible crear instancias de *Prefab* desde el código.

Como se ha mencionado, es posible sobrescribir las propiedades de una instancia de un *Prefab*. Las propiedades sobrescritas de un objeto en concreto se pueden identificar porque salen en negrita en el inspector (Figura 4.14).

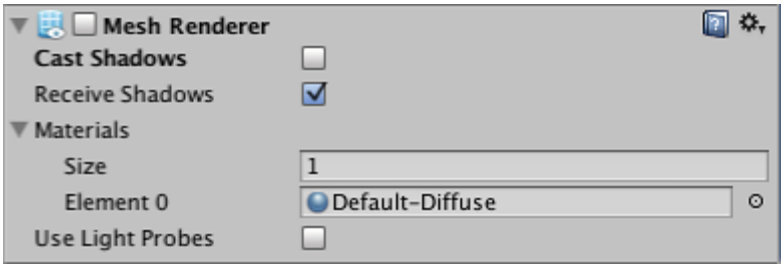


Figura 4.14 – Componente con una propiedad sobrescrita.

Es posible editar un *Prefab* desde una de sus instancias. En el inspector de una instancia aparecen tres botones nuevos: *Select*, *Revert* y *Apply*.

- *Select*: Permite seleccionar el *Prefab* del que se generó esta instancia.
- *Apply*: Permite aplicar los cambios sobrescritos en la instancia al *Prefab* y por lo tanto a todas las instancias del mismo (salvo aquellas que la tuvieran sobrescritas).
- *Revert*: Permite devolver una instancia modificada al estado base del *Prefab*.

A continuación, se van a presentar varias secciones que explican las dos escenas creadas y los elementos necesarios para crear cada escena.

4.2. Escena Main menu.

Esta escena gestiona el acceso a las distintas simulaciones realizadas para la aplicación y la conexión con el servidor remoto.

En la figura 4.15 se presenta cómo funciona el menú principal y las posibles transiciones entre las distintas pantallas:

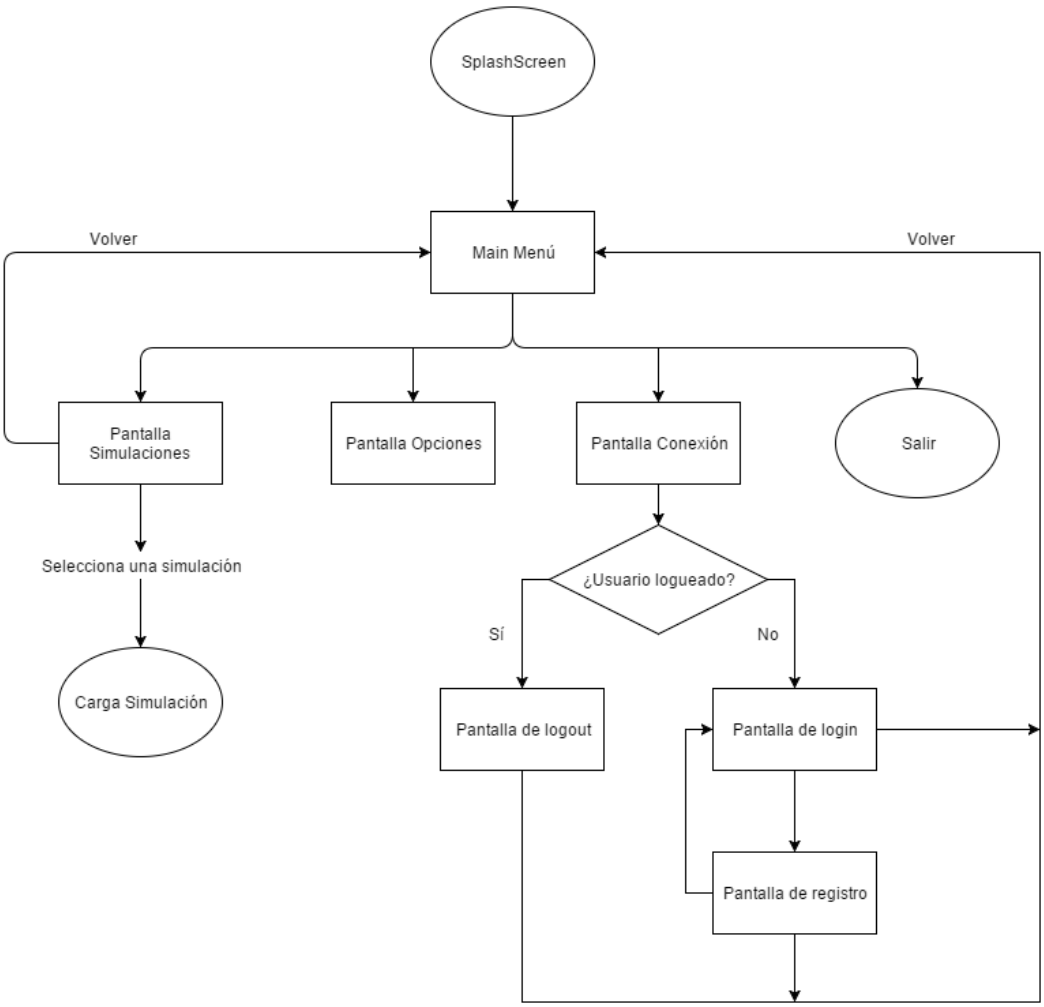


Figura 4.15 – Diagrama esquemático de transiciones.

Desde el menú principal se puede acceder a los siguientes botones:

- **Comenzar**: Lanza la pantalla de simulaciones.
- **Opciones**: Lanza la pantalla de opciones.
- **Conexión**: Lanza la pantalla de inicio de conexión. Esta dependerá de si el usuario está validado en el servido remoto o no.
- **Salir**: Cierra la aplicación.

El número de elementos necesarios para crear un menú es razonablemente pequeño. Para la parte visual solo se requiere de la herramienta de desarrollo de interfaces gráficas de Unity, denominado UI. Construir interfaces gráficas en Unity requiere del elemento *Canvas* y del sistema de gestión de eventos *EventSystem*, ambos creados cuando se intenta añadir un elemento perteneciente a la *suite* gráfica.

Se decidió crear cada una de las pantallas como paneles y gestionar las transiciones mediante scripts. Para ello se creó un objeto *GameController* que recoge todos los scripts utilizados para ello. En la fase inicial de desarrollo de menú principal se había creado la estructura de objetos pero no tenían funcionalidades añadidas, solo estaba creado el fondo mediante imágenes.

4.2.1. UI, Canvas, Panels.

UI (Manual Unity, 2015e) es el nombre que se le dio al nuevo sistema de creación de interfaces gráficas en Unity. En concreto este permite generar elementos de interacción como botones o desplegables como cualquier otro sistema similar, pero permite generarlo sobre un *Canvas* o Lienzo que puede ser *renderizado* de múltiples formas en la escena. Para el menú se ha utilizado el modo de *renderizado* Screen Space – Camera (Figura 4.16) que permite aplicar cierta perspectiva a nuestra interfaz de usuario (básicamente permite el giro de elementos dando mayor sensación de profundidad).

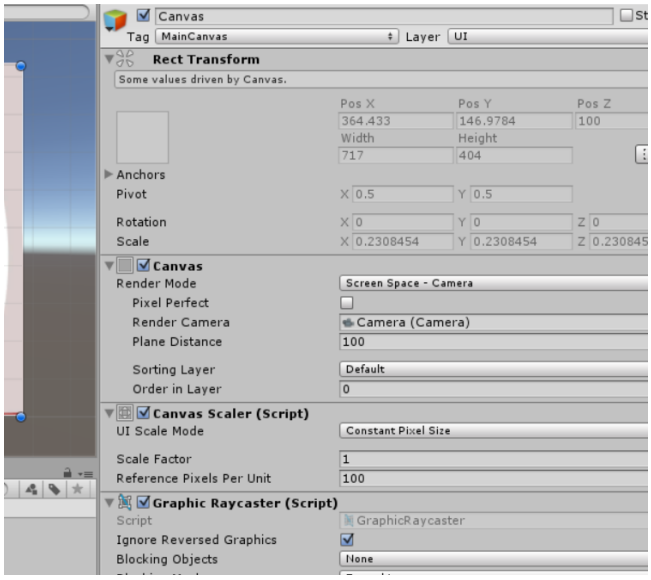


Figura 4.16 – Selección de modo de renderizado del *Canvas.*

Crear las distintas pantallas es muy simple. Solo hay que ir creando los elementos necesarios para construir la interfaz tal y como se desee.

Por ejemplo, para la pantalla principal del menú se ha creado un panel y dentro los cuatro botones generales. Los paneles (*Panels*) son agrupaciones de elementos dentro de un mismo *Canvas* y en nuestro caso los hemos usado para crear las distintas pantallas.

La creación de las transiciones entre las pantallas se realiza mediante el *script MenuTransitions* y dos animaciones, una de apertura y otra de cierre. Este *script* contiene dos métodos denominados *OpenMenuPanel()* y *CloseMenuPanel()* que reciben el nombre del elemento a abrir (como *Simulation* u *Options*). La llamada a estos métodos se hace desde el inspector de los botones de forma sencilla. El sistema UI permite realizar asociaciones de métodos a eventos (conocidos como *EventListener*) desde el propio editor. Obviamente también puede hacerse desde un script.

El sistema de transiciones es muy sencillo. Por defecto todas las pantallas están desactivadas y todas tienen asociadas las animaciones de apertura y de cierre. Cuando se pulsa un botón se lanza el método *OpenMenuPanel* o *CloseMenuPanel*. Estos

métodos se encargan de activar o desactivar los paneles adecuados y de lanzar las animaciones.

Tanto la pantalla de Simulación como la de Opciones no presentaron diferencias significativas en su desarrollo frente la del menú principal.

En la pantalla de simulaciones se presenta un listado de las 5 simulaciones disponibles a realizar y al pulsar en una de ellas se lanza la simulación adecuada mediante el método *SceneManager.LoadScene()*.

En concreto, estas simulaciones representan cinco trayectos distintos dentro de la misma escena. En la primera simulación se realiza un trayecto que empieza en ciudad y acaba en zona interurbana mientras que la segunda simulación hace el trayecto inverso. Estas dos simulaciones se realizan en horario diurno. Las simulaciones tercera y cuarta son similares a las dos anteriores salvo que se realizan de noche. La simulación cinco permite moverse libremente mientras el escenario va cambiando en un ciclo de día a noche.

La pantalla de opciones permite cambiar algunas configuraciones por defecto de las simulaciones. En concreto, se puede cambiar el volumen, el tipo de cambio de marchas, si se desea que aparezcan indicaciones en los cruces en la simulación libre y si se desea que aparezca información de las infracciones cometidas.

Toda la información se almacena mediante la clase *PlayerPrefs* que pertenece a Unity y que permite guardar datos simples de forma persistente.

4.2.2. Conexión con el servidor remoto.

La pantalla de Conexión será nuestro punto de relación con el servidor remoto. Para ello se han diseñado principalmente tres clases: *User*, *WebServices* y *ValidateForms*.

La clase *User* es la más importante y la encargada de gestionar los datos de un usuario en relación con el servicio web. Mantiene unas variables de control: nombre de usuario, *password*, *token* CSRF, entre otras. La clase se encarga de lanzar las peticiones para realizar el *Login*, *Logout* y Registro de usuarios.

Para gestionar las peticiones se ha recurrido a la librería UnityHTTP (https://github.com/andyburke/UnityHTTP). Esta librería encapsula las funcionalidades que posee Unity de envío y recepción de datos mediante peticiones HTTP. Básicamente proporciona una clase *Request* que permite crear las peticiones y gestionar las respuestas. Dispone de métodos orientados exclusivamente a manejar estas peticiones como *AddHeader()* para añadir cabeceras o *Send()* para enviar la petición.

El funcionamiento es el siguiente: cuando se lanza la validación de usuario desde la pantalla, se realiza una petición POST HTTP al servicio web de validación de usuario del servidor remoto pasándole los datos necesarios en el cuerpo de la petición en formato JSON. Por ejemplo, en el caso de la validación de usuario en la plataforma web (*login*) es necesario pasar en el JSON los campos “username” y “password”. Si la petición es correcta (estado 200), el servicio web de Drupal devuelve los datos del usuario como su UID o su *token* de identificación (que será necesario de ahora en adelante para demostrar que este usuario es el usuario validado en el sistema).

La pantalla de *logout* sigue una funcionalidad similar. Simplemente se lanza una petición POST al servicio web de *logout* con el *token* CSRF de identificación del usuario y si la petición es correcta se da al usuario por desconectado del servidor.

El registro es un poco más complejo. El registro de un usuario en Drupal necesita de una serie de campos que vienen definidos por los campos que tiene la entidad ***user*** en el servidor. Como mínimo un usuario va a necesitar los campos "name", "mail" y "pass" para ser dado de alta, pero puede necesitar más. En nuestro caso los usuarios del servicio tienen cuatro campos a mayores y, como son obligatorios es necesario pasarlos en el formulario de registro. El método que gestiona el registro de usuario es *RegisterUser()*, que se encarga de inicializar los valores y de realizar la petición.

Todas las direcciones de los recursos web están definidas en la clase *WebServices*. Si hubiera que cambiar el servidor donde están los servicios web activos bastaría con modificar la dirección aquí y recompilar la aplicación. En general, se han intentado sacar todas las variables posibles a clases externas como buena práctica para intentar que la realización de ciertos cambios en la aplicación sea lo más intuitivo posible.

Hay que tener en cuenta que las llamadas a los métodos *LoginUser()* y *RegisterUser()* no se hacen directamente sino que primero se llama a los métodos *ValidateLoginForm()* y *ValidateRegisterForm()*. Estos métodos básicamente se encargan de validar los formularios para que no se envíen datos erróneos a los servicios web.

4.3. MyVirtualCity

Este es el escenario principal de nuestra aplicación. Se ha intentado recrear una ciudad al completo que resulte, dentro los márgenes aceptables, "viva" con peatones moviéndose aleatoriamente y con un tráfico dinámico.

La idea inicial era generar cinco escenarios, uno para cada simulación, pero realmente se estaban creando cinco escenarios iguales en los que cambiaban pocas cosas más allá de la hora del día de la simulación o de la posición de inicio y fin.

Por ello, se redujo la inicialización de datos a una clase *InitSimulationController* que se encarga de modificar todos los datos de la escena en tiempo de ejecución, permitiendo que la misma escena tenga comportamientos distintos. Por supuesto, en situaciones más complejas tal vez fuera necesario replicar y modificar la escena, pero en este caso esto nos permitía no tener que rehacer los cambios hechos en una escena en todas las demás si queríamos que se mantuvieran iguales, además de reducir el tamaño de la aplicación compilada final.

En las secciones siguientes se presentan los distintos elementos y su orden de desarrollo, además de las técnicas utilizadas tanto para llevarlos a cabo a nivel funcional como aquellas aplicadas para intentar mejorar el rendimiento.

4.3.1. Creación del escenario. *Terrain engine* de Unity.

El primer paso para crear el escenario es crear un terreno usando el *Terrain engine* de Unity. Esto se puede realizar desde la opción de menú **GameObject > 3D Object > Terrain,** tal y como muestra la figura 4.17**:**

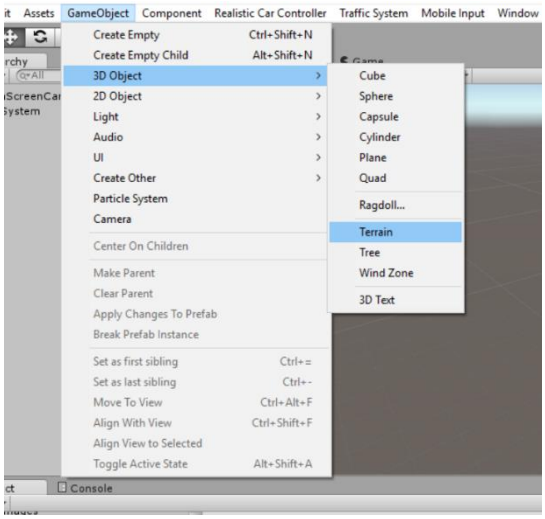


Figura 4.17 – Como añadir un terreno.

Mediante el sistema de esculpido del terreno de Unity se han ido creando las montañas, valles, etc., de nuestro escenario como se ve en la figura 4.18:

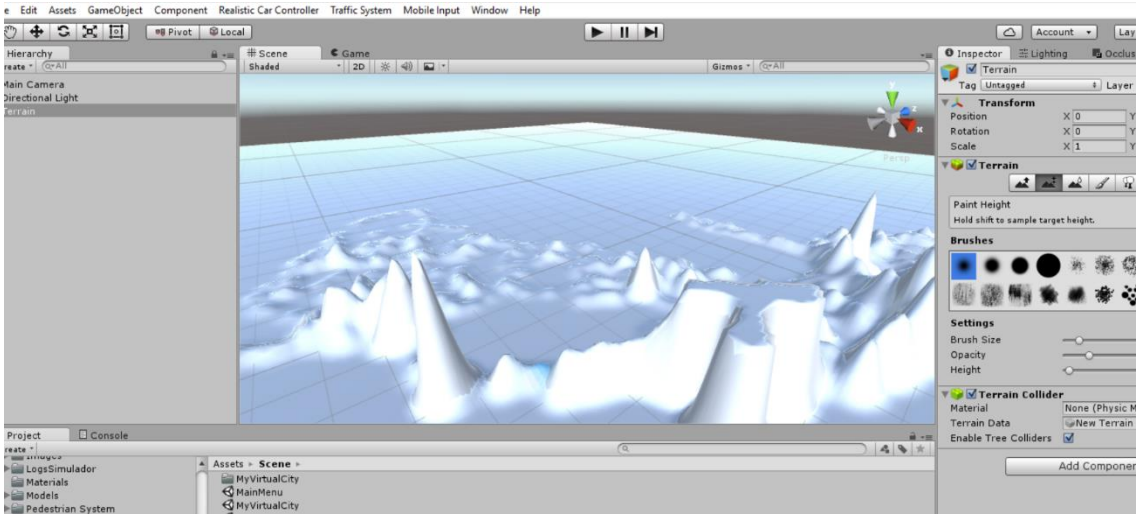


Figura 4.18 – Esculpido del terreno.

En el inspector de un terreno se tiene acceso a varias herramientas de esculpido, que permiten generar el terreno de distintas maneras. En la Figura 4.19 se puede ver cómo es la ventana del inspector de un terreno en Unity.

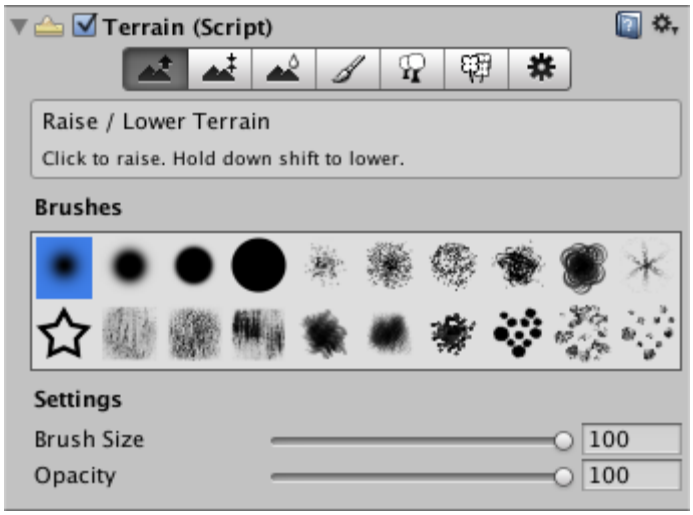


Figura 4.19 – Inspector de un terreno.

Los primeros tres botones permiten modificar la altura y forma del terreno mediante una serie de pinceles que se usan directamente sobre el terreno en la escena. Estos tres botones permiten modificar la altura del terreno de forma libre, modificar la altura hasta un valor fijo (permite crear planicies) y suavizar los contornos del terreno respectivamente. Los pinceles tienen ciertas propiedades como tamaño u opacidad que modifican el área del pincel o la potencia de su efecto respectivamente.

Luego se dispone de un botón que permite pintar texturas sobre nuestro terreno.

También se tienen dos botones que permiten la colocación en masa de árboles, hierba, flores, etc. Todo esto combinado permite generar terreno de cierta complejidad visual, como se puede ver en la figura 4.20.

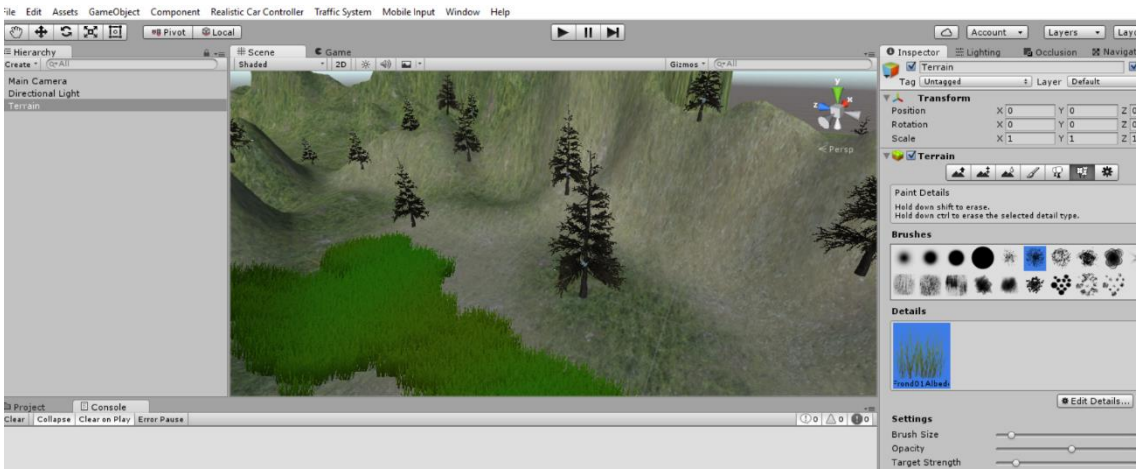


Figura 4.20 – Terreno con árboles, hierba y texturas.

Por último, existe una sección de Configuración del terreno desde donde se pueden gestionar ciertas propiedades de los terrenos que permiten modificar la manera en que estos son *renderizados*. Se pueden establecer varias propiedades como *Pixel Error*, *Material*, *Cast* Shadows y muchas otras. En la figura 4.21 se pueden observar cuáles son las que se han puesto en los terrenos que hemos generado.

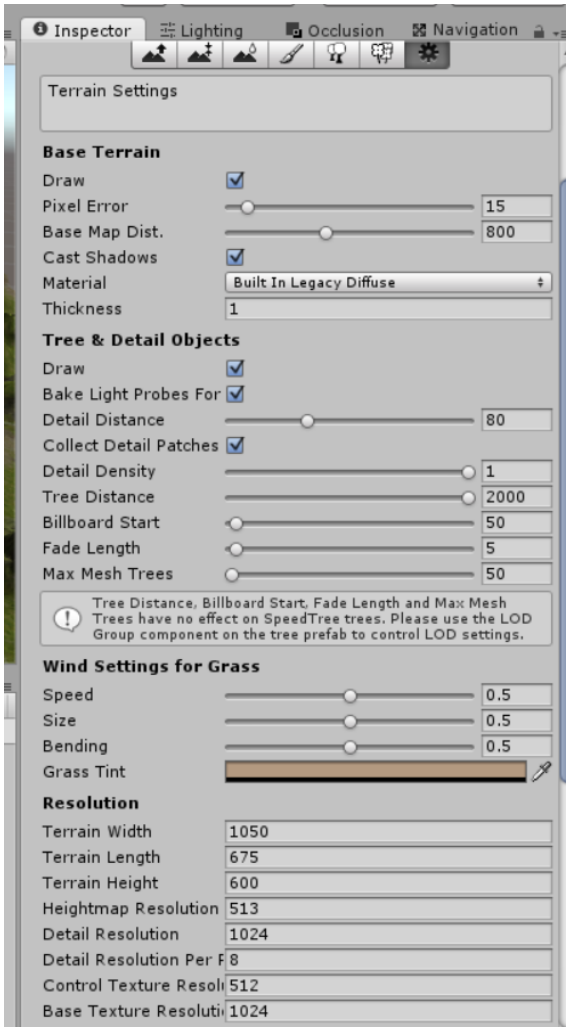


Figura 4.21 – Propiedades de configuración de un terreno.

En nuestro caso, el valor principal modificado ha sido el *Base Map Dist*, que define a partir de qué distancia (en unidades de Unity) se empieza a *renderizar* un escenario con un nivel de detalle menor. Disminuir este valor mejora el rendimiento de forma genérica. Obviamente, este valor solo tiene sentido si está por debajo de la distancia máxima de dibujado de la cámara. La modificación de estos valores permite balancear la carga computacional y la representación visual. Toda la información de estos parámetros está disponible en http://docs.unity3d.com/Manual/terrain-OtherSettings.html.

Para crear nuestro escenario se ha recurrido al uso de cuatro terrenos en lugar de uno. Esto permite gestionar los terrenos como bloques independientes y así se pueden tener distintas características para cada terreno, permitiendo más flexibilidad.

Sin embargo, cuando se utilizan varios terrenos es necesario indicar a cada terreno cuáles son sus "vecinos" de forma que se mantenga el LOD (*Level of Detail*) de forma continua entre los terrenos. Para ello se ha creado un script *TerrainManager* que, asignado a un terreno, permite desde el inspector asignar los terrenos colindantes.

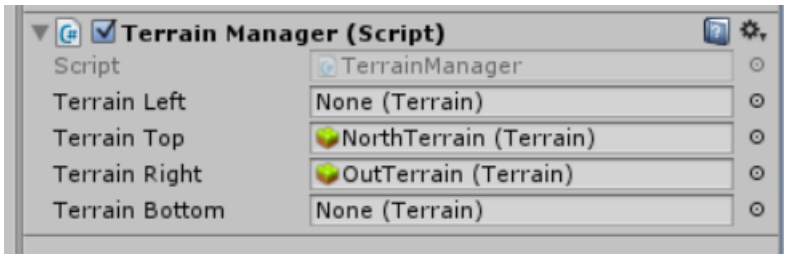


Figura 4.22 – Script Terrain Manager.

La figura 4.22 muestra el inspector de elementos del script *TerrainManager*. Este script se añade a un terreno y mediante el inspector se asignan los terrenos colindantes. El script utiliza el método *SetNeighbours* para establecer cuáles son estos terrenos (Scripting API Unity, 2015).

4.3.2. Pantalla de opciones, de pausa y de finalización.

Una vez diseñado el terreno, era necesario crear las pantallas de inicio, pausa y fin de una simulación. Estas tres pantallas forman parte del *Canvas* principal del escenario. Cada una de estas pantallas tiene un controlador asociado con todos los métodos y propiedades necesarias para su funcionamiento.

Al entrar en una nueva simulación aparece de nuevo la ventana de opciones como en el menú principal, con los datos que almacenamos en dicha pantalla. Sin embargo, aquí se nos vuelve a permitir modificar estas opciones, pero solo para esta simulación en concreto.

Estas opciones son:

- Selección del tipo de cambio de marchas. Permite modificar cómo se cambian las marchas en el vehículo.
- Simulación con indicaciones: permite seleccionar si aparecen o no las indicaciones al entrar en una intersección o glorieta.
- Información en la simulación: muestra un panel de información cada vez que se comete una infracción durante la simulación.

Cuando se han seleccionado las opciones y se pulsa en Comenzar, se lanza el método *StartSimulation*() que gestiona la inicialización de todas la configuraciones asociadas a cada simulación, como las salidas a tomar en los cruces, el cielo utilizado o el color ambiental. Como se ha mencionado al comienzo del apartado 4.3, al realizar la configuración mediante código podíamos reutilizar la misma escena para todas las simulaciones. Esto ha conllevado una reducción del tamaño de la aplicación compilada en al menos 600 Mb, lo que supone un ahorro de espacio muy grande sin pérdida de funcionalidad.

La pantalla de pausa es muy simple y su única función es detener el juego para poder volver al menú principal o salir directamente de la aplicación. El controlador, *PauseController*, se encarga de abrir o cerrar la ventana, de pausar el tiempo de juego usando *Time*.timeScale = 0 y de reactivarlo usando *Time*.timeScale = 1.

Este controlador además se encarga de almacenar la partida en un fichero, aunque no se haya finalizado la simulación completamente. El fichero se marca con un IMCO para dar a entender que es una partida incompleta.

La pantalla de finalización muestra un breve resumen de los datos recopilados durante la simulación y permite volver al menú principal. Cuando la simulación finaliza se almacena siempre un fichero de log con toda la información recopilada en formato

JSON. Este es el mismo formato que se envía al servidor y siempre es posible cargarlo manualmente desde el servidor si así se desea.

En cualquier caso es posible, si el usuario está validado, enviar directamente la información al servidor remoto pulsando en el botón grabar. Este botón solo está disponible para usuarios validados en la aplicación. La gestión del lanzamiento de la pantalla, codificación de datos a JSON y el envío al servidor remoto se hace desde la clase *FinishSimulationController*. Esta clase contiene algunos métodos públicos de utilidad, siendo el principal el método *FinishSimulation*(), que lanza la pantalla y almacena el fichero de log. Además existe el método *SendPartidaToDatabase*(), que permite mandar el fichero JSON al servidor remoto. Este método se enlaza al botón grabar desde la interfaz de usuario.

4.3.3. Road & Traffic System.

Este es un *plugin* de la ***Asset Store*** que permite incluir tráfico con cierto realismo en nuestra simulación.

El funcionamiento en concreto consiste en crear bloques de carreteras que se unen entre sí mediante nodos. El *plugin* permite a los usuarios crear y simular redes de tráfico dinámicas para cualquier clase de vehículo de una manera rápida. Es posible personalizar las piezas utilizadas, y cuenta con glorietas, intersecciones con semáforos, etc.

Este escenario incluye carreteras de 1, 2 y 3 carriles por sentido, además de intersecciones en 'T', intersecciones de 4 carriles reguladas mediante semáforos y glorietas.

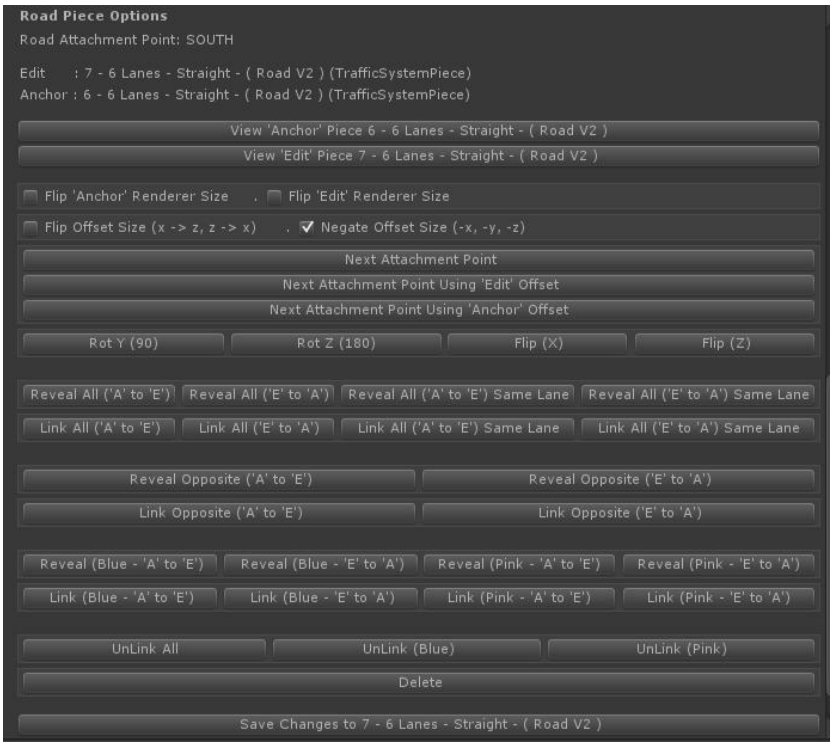


Figura 4.23 – Inspector del objeto Traffic System.

Para utilizar el *plugin* hay que crear un componente *TrafficSystem* que es un *Singleton* (sólo puede haber uno por escena). Desde el inspector del *TrafficSystem* (Figura 4.23) se puede gestionar la creación, conexión y borrado de las piezas o *TrafficSystemPieces,* que son las que representan los bloques de carretera. Cuando se va

a crear un nuevo bloque, el ya existente es el ancla (*Anchor*) mientras que el nuevo es el objeto editable (*Edit*). Cuando creamos una nueva pieza hay que conectarla para que se creen las líneas que luego seguirán los vehículos.

Desgraciadamente el *Road & Traffic System* ha demostrado ser un *plugin* tosco en sus funcionalidades. Los vehículos no responden bien a las colisiones y es fácil que estos pierdan el próximo nodo al que ir lo que hace que se queden bloqueados y tenga que saltar el temporizador para destruir el vehículo. Algunos de estos problemas se han pulido, pero en general habría que modificar el diseño completo para poder arreglar todos estos errores y eso se escapa del propósito del proyecto.

En cualquier caso, se han añadido dos funcionalidades principales a los vehículos del *TrafficSystem* (clase *TrafficSystemVehicle*).

La primera consiste en adaptar los vehículos para que detecten a los peatones (apartado 4.3.4) que tienen delante y paren. Para ello se ha creado un método *CanSeePedestrian*() que detecta si un peatón está delante del vehículo mediante un *CapsuleCast*. Desde el *Udpate* del script se ha añadido el siguiente código que se encarga de detener el vehículo si se puede ver un peatón y el vehículo no está en modo locura (*crazy mode*):

```
// Si se cruza con un peatón parará salvo que esté en modo locura
if (this.CanSeePedestrian() && !this.crazyMode)
{
    StopMoving = true;

}
```

Respecto al modo locura se ha añadido para dotar de situaciones inesperadas a la simulación. Cuando un vehículo está en este modo puede modificar su velocidad de forma más violenta a lo esperado y si se encuentra a un peatón colisiona con él en lugar de detener el vehículo. Para detectar si se está en este modo se realiza la invocación de un método cada 5 segundos que se encarga de activar o no dicho modo. Este método, *ChangeCrazyMode*(), activa el modo locura en un vehículo con una probabilidad establecida en el script y modificable desde el inspector.

4.3.4. Sistema de peatones.

El sistema de peatones está desarrollado desde cero y debía consistir en desarrollar un controlador que creara y moviera los peatones de forma aleatoria por la ciudad de la simulación. Los peatones debían poder detenerse ante los vehículos y deberían poder acelerar, por ejemplo, cuando cruzaran una carretera por un lugar que no fueran los pasos de peatones.

Para llevar esto a cabo se ha utilizado el sistema de navegación de Unity (Manual Unity, 2015b). Este sistema permite ejecutar cálculos de caminos entre dos puntos mediante algoritmos A*. Estos algoritmos buscan el camino más corto entre dos puntos sujetos a unas condiciones. En Unity estas condiciones son los pesos de las áreas.

Lo primero que había que hacer era generar un área nueva que serían las carreteras. Estas áreas tendrían un coste mayor de forma que los peatones solo decidan cruzar por ellas cuando el total computado cruzando estas áreas sea menor que no cruzándolas.

Todas las operaciones para trabajar con la navegación en Unity se realizan desde la pantalla *Navigation*, que se puede abrir desde la opción de menú **Window** > **Navigation.**

En nuestro caso, queríamos tener tres áreas diferenciadas. Unity proporciona tres zonas por defecto: *Walkable*, *Not Walkable* y *Jump*. Tanto las áreas *Walkable* como *Not Walkable* servían a nuestro propósito de crear dos zonas diferenciadas, una por la que los peatones puedan andar y otra por la que no. Pero, además, queríamos una zona que simulara las zonas de carretera y esta debería tener un peso mayor que el área estándar (área *Walkable*). Desde la sección Areas de la ventana de navegación se puede acceder a áreas ya existenten y añadir áreas nuevas.

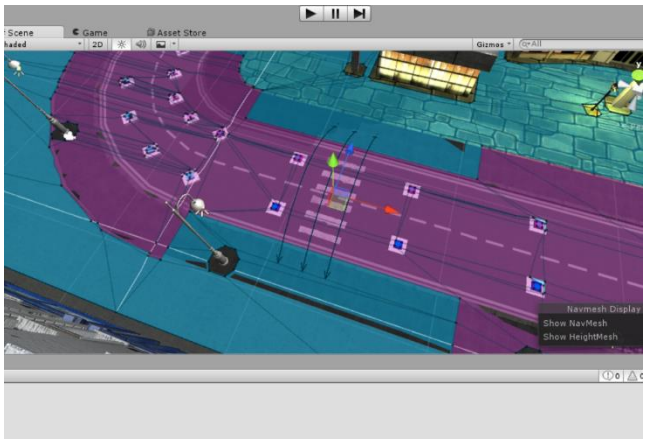
Figura 4.24 – Áreas de navegación.

En la figura 4.24 se puede ver que se ha creado un área nueva a la que se ha llamado Road y que tiene un peso de 4. Esto indica que cada unidad de camino computada sobre esta área será multiplicada por 4 dentro del cálculo global. Una vez definida la nueva área, hay que asignar a los distintos elementos el área al que pertenecen. Esto puede hacerse pulsando el botón *Object* dentro de la ventana de navegación. Desde ahí podremos navegar los objetos en la jerarquía o en la escena y desde esta ventana podremos asignar el área deseada.

En la ventana de selección tenemos tres opciones: si el objeto es estático, si se desea que genere *OffMeshLink* de forma automática (se hablará de los *OffMeshLink* más adelante) y el área del objeto. Cuando marcamos un objeto como estático con respecto a la navegación, le estamos indicando a Unity que este objeto no se va a mover nunca. De esta forma, Unity puede optimizar los algoritmos de cálculo.

Una vez hemos asignado áreas a nuestros objetos, lo que hay que hacer es precomputar (*bake*) la zona de navegación y una vez realizado se podrá observar de forma visual el área de navegación en la escena.

Dos párrafos más arriba se ha mencionado la palabra *OffMeshLink*. Un *OffMeshLink* es un componente que proporciona el motor de navegación de Unity para unir zonas que no son directamente navegables. Por ejemplo, en nuestro caso, queremos que los peatones crucen por los pasos de cebra como si estuvieran en una zona segura. Pero los pasos de cebra están en la carretera y pertenecen al área Road. Para solucionar esto se han creado *OffMeshLink* en estas zonas y se les ha asignado el área *Walkable* tal y como se puede ver en las figuras 4.25 y 4.26:



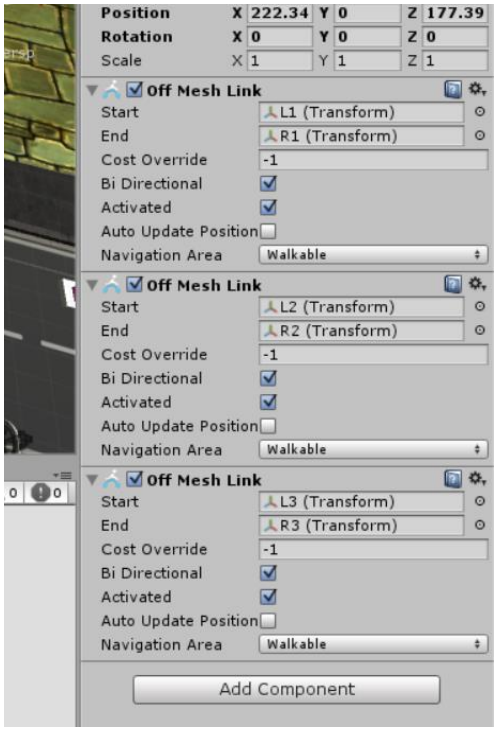


Figura 4.25 y Figura 4.26 – Representación y propiedades de un OffMeshLink.

Ahora ya disponemos de una zona navegable, pero necesitamos objetos que se muevan por ella.

El componente necesario para que un objeto pueda moverse por una zona navegable es el componente *NavMeshAgent*. Este componente proporciona una serie de funcionalidades que permiten asignar puntos de destino, velocidad, etc. y el propio componente se encarga de calcular la ruta a seguir.

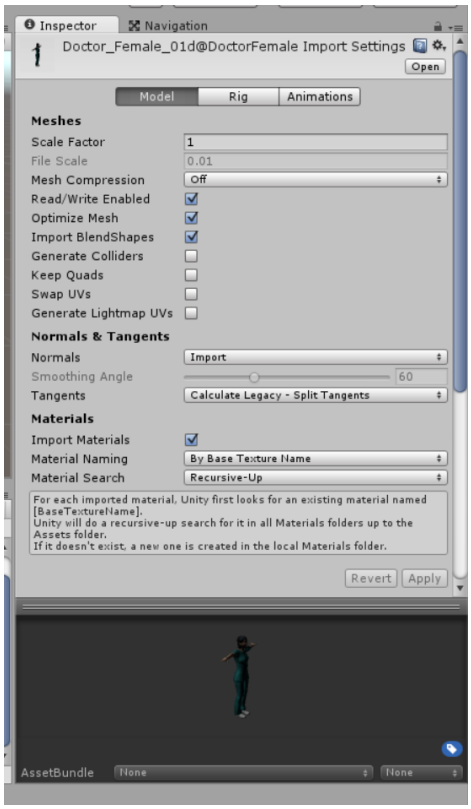


Figura 4.27 – Modelo humanoide.

Para crear nuestro peatón genérico es necesaria una serie de preparaciones.

En primer lugar, necesitamos un modelo tridimensional de un ser humano. Además, debe estar *riggeado*, es decir, tener asignados huesos y articulaciones que permitan mover el modelo y por lo tanto ser animado. El modelo utilizado pertenece a un pack de Mixamo gratuito. Cuando importas un modelo en Unity, se pueden modificar y adaptar varias de sus características. Además, permite importar los materiales o animaciones del modelo para ser usados en Unity. La figura 4.27 muestra la pantalla de importación de nuestro peatón por defecto:

Desde esta ventana de configuración del modelo se puede comprobar y modificar el *rigging* del modelo importado (Figura 4.28). Las animaciones se pueden crear en un programa de modelado como Blender, sin embargo, en nuestro caso se ha optado por usar las animaciones proporcionadas en los *Standard Assets* de Unity. Estas animaciones permiten animar de forma genérica los modelos humanoides (dos piernas, dos brazos y una cabeza). En definitiva, se ha utilizado el controlador de animaciones y el script de *ThirdPersonController* de los *Standard Assets* para mover y animar a nuestro peatón.

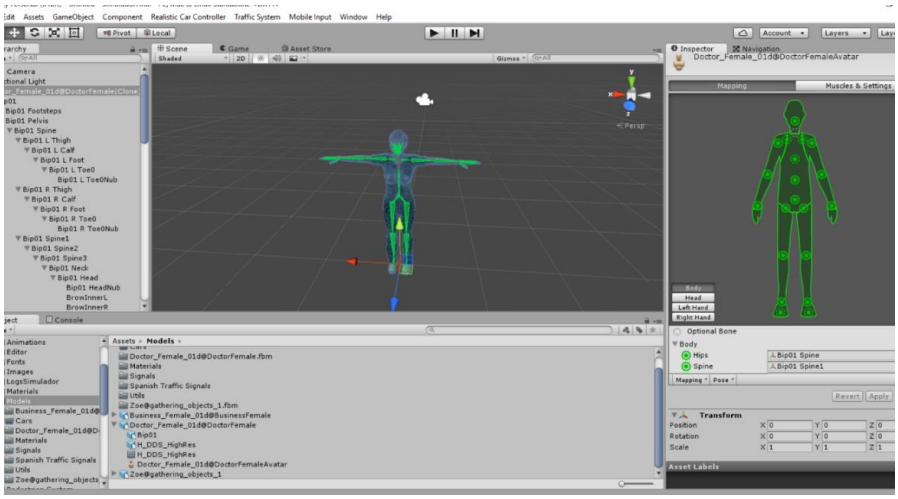


Figura 4.28 – Rigging de un modelo tridimensional.

Para gestionar las animaciones, Unity ha creado un nuevo sistema denominado *Mecanim*. Este sistema permite representar las animaciones como transiciones entre estados, tales como andando, parado, saltar, etc. El controlador de animación de los *Standard Assets* utiliza tres estados genéricos (Figura 4.29): En el suelo (*Grounded*), en el aire (*Airborne*) y agachado (*Crouching*).

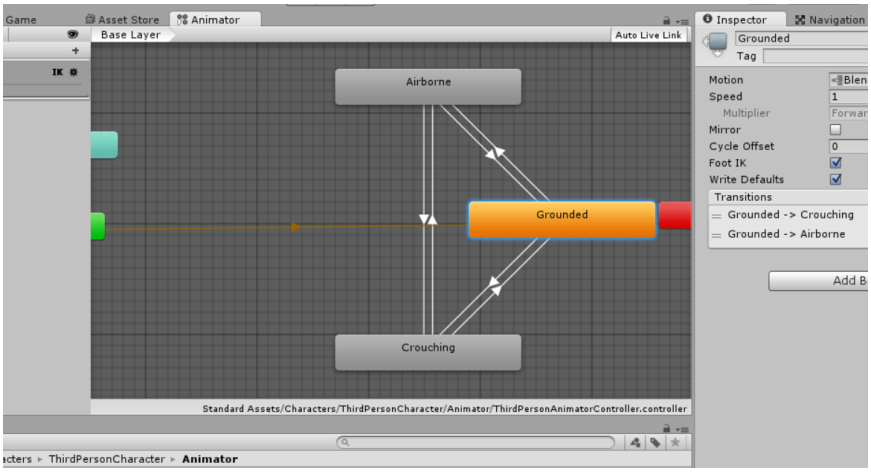


Figura 4.29 – Animator Controller de un personaje.

Se puede pasar de cada uno de estos estados a otro siguiendo las líneas de transición. Estas transiciones se realizan a nivel de código mediante variables. Cada uno de estos estados tiene asociado un *BlendTree*. Los *BlendTrees* permiten definir animaciones variables según los datos de entrada. Supongamos que queremos variar las animaciones según los ejes de entrada de manera progresiva, de forma que la animación se adapte, entonces podemos usar un *BlendTree*. Estos mapean las animaciones a datos de entrada analógicos, generalmente los ejes de un dispositivo. De esta manera, se puede modificar, por ejemplo, la animación de moverse hacia delante de forma que varíe progresivamente según forzamos más el eje de entrada, pasando de andar a correr.

En la Figura 4.30 se pueden ver el *Prefab* de nuestro peatón con los componentes necesarios para su funcionamiento dentro de nuestro sistema de peatones.

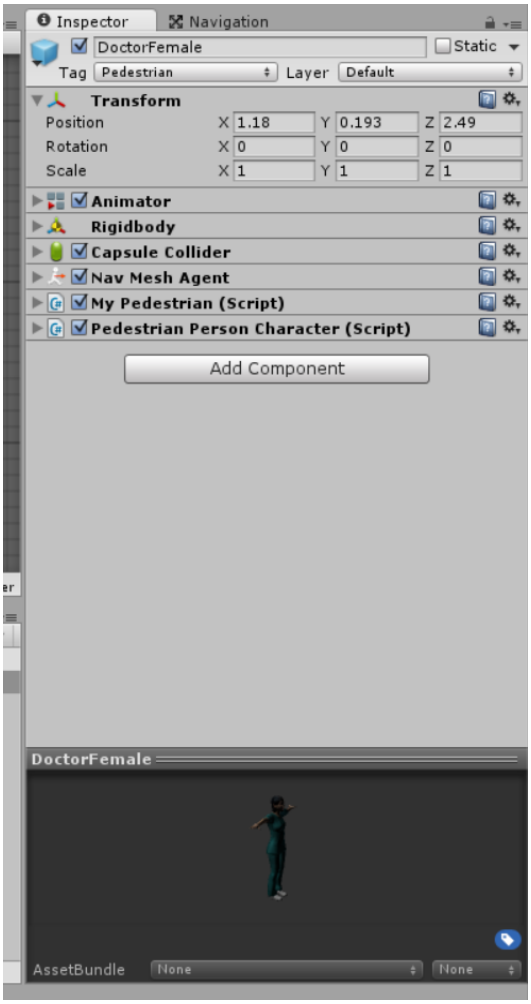


Figura 4.30 – *Prefab* de un peatón

Listado de componentes:

- **Animator**: Es el componente encargado de usar el sistema *Mecanim* para animar nuestro modelo.
- **Rigidbody**: Otorga masa a nuestro modelo.
- **Capsule collider**: Establece un volumen físico que representa nuestro objeto dentro del motor de físicas.
- **NavMeshAgent**: Es el componente que se encarga de moverse por el área de navegación.
- **PedestrianPersonCharacter**: Este script es una copia del script *ThirdPersonCharacter* de los *Standard Assets* y se encarga de la animación y movimiento del peatón.

• **MyPedestrian**: Este script se encargará, junto con el script *PedestrianController*, de gestionar la Inteligencia Artificial o IA de los peatones. La IA de nuestros peatones pretende establecer comportamientos sencillos pero aleatorios mediante la selección de puntos de destino de forma dinámica en conjunción con el uso del sistema de navegación de Unity. El diagrama de flujo de la misma se muestra más adelante en la figura 4.31.

Ahora ya se ha creado el peatón, pero necesitamos instanciarlos y asignarlos dentro de nuestra escena. Para ello se crea una clase, *PedestrianController*, que va encargarse de crear los peatones al inicio de la simulación. Esta clase recupera los puntos de inicio y los puntos de destino establecidos en la simulación. Los peatones se crean aleatoriamente en los puntos de inicio establecidos y se les asigna, también aleatoriamente, un punto de destino.

Durante la simulación, la clase *MyPedestrian* se encarga de modificar los puntos de destino cuando se llega a ellos. En la figura 4.42 podemos ver de forma general cómo funciona este script:

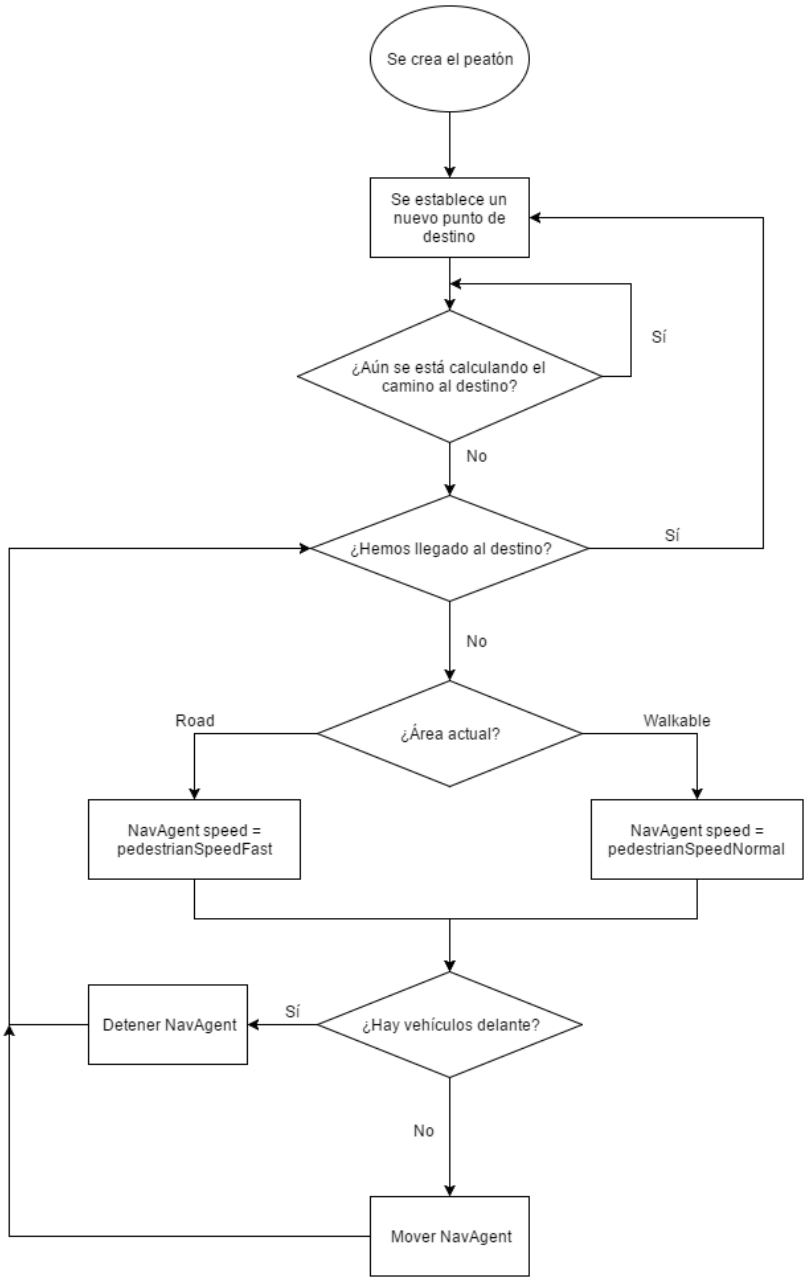


Figura 4.31 – Diagrama de flujo del comportamiento de un peatón.

Este script recurre al componente *NavAgent* para calcular el camino hasta el destino. Luego realiza dos comprobaciones básicas para detectar si el peatón debe andar o no. Estas comprobaciones son:

• *CheckActualArea*(): Comprueba si nuestro peatón se encuentra en un área *Walkable* o *Road*. En función de ello se establece una velocidad distinta para el componente *NavAgent*.
• *CheckVehiclesInFront*(): Comprueba mediante un sencillo *Raycast* la existencia de vehículos delante del peatón.

Al final se mueve el peatón mediante el método *Move*() de la clase *PedestrianPersonCharacter*.

4.3.5. Sistema de recuperación de datos.

La clase *DataController* se encarga de la recuperación de datos de forma regular. En realidad, es un sistema muy simple. Esta clase contiene otra clase interna llamada *DatoInstantaneo* que almacena información relativa a un instante dado como velocidad, consumo, etc.

Mediante el método *InvokeRepeating*() se lanza otro método de forma repetitiva en un intervalo de tiempo. Resumiendo, mediante *InvokeRepeating* lanzamos nuestro método *AddNewDatoInstantaneo* que va creando una nueva instancia de nuestra clase *DatoInstantaneo* y la va almacenando en un listado de objetos *DatoInstantaneo*.

La figura 4.32 muestra el inspector del script *DataController* y los parámetros modificables desde el mismo:

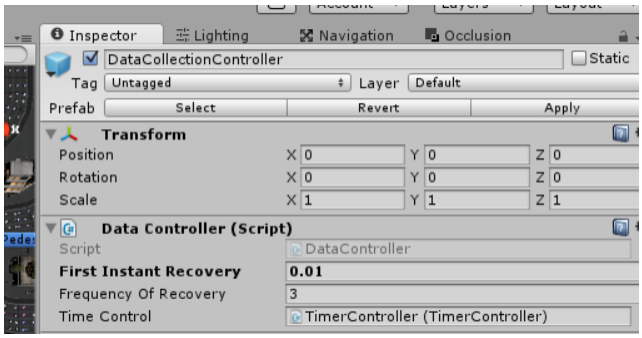


Figura 4.32 – Inspector del script DataController

Los parámetros modificables permiten cambiar el primer instante de recuperación de datos (*First Instant Recovery*) y la frecuencia de recuperación de datos (*Frequency of recovery*).

4.3.6. Sistema de infracciones.

Esta es una de las partes centrales de nuestro proyecto. La idea era crear un controlador que gestionara un cierto número de infracciones y permitiera almacenarlas de la misma forma en que se almacenan datos. De forma genérica podemos definir este controlador como el sistema que define las infracciones posibles (mediante un *enum* numérico donde los valores de las infracciones son los IDs de las mismas en nuestra base de datos de la plataforma web) y que permite a otras clases añadir infracciones mediante el método *AddNewInfraccion*.

Listado de infracciones gestionadas mediante el controlador:

```
public enum TiposInfraccion {
```

```
        AtropelloPeaton = 1,
        ColisionCoche = 2,
        PisarAcera = 3,
        ExcesoVelocidad = 4,
        SalidaRotondaIncorrecta = 5,
        RutaRotondaIncorrecta = 6,
        RotondaRealizadaEnSentidoInverso = 7,
        SalidaInterseccionIncorrecta = 8,
        StopRealizadoIncorrectamente = 9,
        CedaRealizadoIncorrectamente = 10,
        CruzadoSemaforoEnRojo = 11,
        ColisionObjeto = 12,
        CruceLineaContinua = 13,
        LucesEnTunel = 14,
        LucesDeNoche = 15,
        GiroInterseccionIncorrecto = 16,
        NoRespetadaDistanciaSeguridad = 17,
        IntermitenteNoUtilizado = 18,
        AdelantamientoSinIntermitentes = 19,
        VelocidadDemasiadoBaja = 20
    }
```

La mayoría de estas infracciones están consideradas como tal en el Manual del Conductor de la DGT (DGT, 2015a) y se ha pretendido llevarlas a cabo de la manera más fiel posible en el simulador. Algunas de las infracciones están relacionadas con las indicaciones que se muestran en las intersecciones y glorietas y que pretenden simular las direcciones que podría indicar un examinador del carnet de conducir.

Como funcionalidad extra, permite mostrar cierta información por pantalla cada vez que el conductor comete una infracción de forma que pueda ver qué ha hecho mal.

A continuación, se presenta cómo se han implementado la mayoría de estas infracciones.

4.3.6.1. Infracciones generales relativas a glorietas e intersecciones

Se han desarrollado dos infracciones relativas a las indicaciones: una para intersecciones (en “T” y de cuatro carriles) y otra para las glorietas. Estas infracciones utilizan *triggers* en cada uno de las entradas para detectar por qué sitio entra y sale el vehículo.

Cuando un vehículo entra por uno de los *triggers* más pequeños dentro de las intersecciones o glorietas se selecciona la salida a tomar y se activa la imagen que representa esa salida. La infracción se lanza cuando el vehículo no sale por la salida marcada en la imagen.

Esto permite detectar cuándo el conductor no está cumpliendo las indicaciones.

Para gestionar las intersecciones y glorietas se crearon varias clases abstractas y se utilizó la herencia para adaptar los casos concretos.

La clase *IndicationControl* estableces unas propiedades y métodos que deben implementar todas las clases que hereden de ellas, en concreto las clases *RoundaboutIndicationControl* e *IntersectionIndicationControl*. Estas clases controlan el flujo genérico dentro de las intersecciones y se encargan de activar o desactivar objetos específicos propios, como los encargados de comprobar en una glorieta si esta se ha realizado correctamente o si se ha realizado en sentido contrario.

Luego, tanto para rotondas como intersecciones, existe una clase *IndicationAbstract* de la que deben heredar las clases que controlan las salidas de las intersecciones. Dependiendo del número de salidas de una glorieta o intersección, las implementaciones de ciertos métodos son distintas. Hacerlo de esta manera permite ceder la ejecución de la funcionalidad a la clase abstracta y delegar la implementación de los métodos concretos a las clases hijo. El código en concreto se puede ver inspeccionando las clases *IntersectionIndicationAbstract* y sus herencias *IntersectionIndication4Exits* e *IntersectionIndication3Exits*. Es cierto que al tener solo dos casos sería viable haber programado todo mediante condiciones, pero eso evitaría flexibilidad y escalabilidad al código. En definitiva, las clases acabadas *Exit* controlan cada una de las salidas y su funcionalidad principal es detectar cuándo un vehículo entra (o sale) por ella y de esa manera comprobar a posteriori si se ha tomado la salida marcada al entrar en la glorieta o no.

Aparte de las infracciones de las indicaciones, se ha establecido una infracción relativa a no utilizar los intermitentes al realizar una intersección o una glorieta. La norma de circulación indica que se debe utilizar el intermitente de la dirección adecuada cuando se gira en una intersección o se toma una salida de una glorieta. Esta implementación requería de una clase que permitiera controlas los intermitentes de nuestro vehículo. Dicha funcionalidad ha sido implementada mediante la clase *BlinkingCarControl*. Esta clase permite activar los intermitentes mediante botones de entrada y también leer el estado de los mismos. De esta forma es muy sencillo implementar una funcionalidad en nuestras clases para controlar la salida de las glorietas para que comprueben si se el intermitente está activo al salir de la glorieta.

En las intersecciones es un poco más complicado porque el intermitente a presionar depende de la salida. Por ello se mantiene un control durante la realización de la intersección que detecta si se ha pulsado alguno de los intermitentes. Cuando se sale de la intersección se tiene un método que devuelve si la salida tomada respecto a la entrada es la primera, la frontal o la izquierda y mediante esto se comprueba si se ha girado a izquierda a derecha y si se ha activado el intermitente adecuado.

4.3.6.2. Glorietas

Las glorietas suelen ser un punto clave en la conducción debido a la falta de un reglamento específico sobre ellas como se ha mencionado en apartado 2.2.5.

Aunque no son todas las viables, se han establecido dos infracciones relacionadas con las glorietas.

La primera consiste en comprobar que se ha realizado la salida correctamente. Para ello se comprueba a posteriori, en función de la salida tomada, si el conductor se posicionó en el carril externo antes de tomar la salida. Esto se realiza mediante un *Prefab* de *triggers* que se colocan en la posición y orientación adecuadas al entrar en la glorieta.

Cuando el conductor pasa por estos *triggers* se marcan parámetros a TRUE de forma que se puede comprobar el curso del vehículo por la glorieta a posteriori. Esto es un poco mayor en complejidad que solo comprobar si el vehículo se posicionó en el carril exterior. De hecho, permitiría comprobar si el conductor realizó una ruta en concreto por dentro de la glorieta. Esta infracción nos permite establecer, al menos, cuando un conductor ha realizado una glorieta de forma no incorrecta.

La segunda infracción comprueba si se intenta hacer la glorieta en sentido contrario. La norma indica que las glorietas deben hacerse siempre en sentido antihorario, por lo tanto, tomarla en sentido horario no solo es una infracción muy grave

sino también muy peligrosa. Para gestionar esta infracción se coloca otro bloque de dos *triggers* adecuadamente.

Si el conductor entra en el *trigger* de la izquierda (nada más entrar en la glorieta) se añade la infracción, mientras que si entra en el de la derecha se desactiva el bloque de *triggers*.

4.3.6.3. Stop

Para realizar un Stop correctamente, hay que detener el vehículo por completo antes de la intersección señalizada. Luego solo se puede invadir la intersección cuando no haya vehículos en la misma. Para gestionar esta infracción se requiere de un bloque de dos *triggers* (uno a la entrada de la intersseción y otro sobre la propia intersección) y de dos scripts para gestionarla: *StopTrafficControl* y *StopIntersectionDetection*.

Cuando un vehículo entra en el *trigger* de la entrada (la zona de parada), el script *StopTrafficControl* controla si el vehículo se detiene por completo o no y lo almacena como una variable. Cuando el vehículo abandona la zona de parada se activa el *trigger* de la intersección y se comprueba si existe algún vehículo en la intersección que no sea el conductor. Si alguna de estas dos comprobaciones devuelve TRUE, se considera que el conductor ha realizado incorrectamente el Stop y se añade una nueva infracción.

4.3.6.4. Ceda el paso

Las infracciones de ceda el paso son algo más complejas que las de stop, aunque siguen una lógica similar.

También se utilizan dos *triggers* aunque ahora no basta con comprobar que existan vehículos en la intersección sino también qué salida tomaron los mismos. Las normas de circulación definen el ceda el paso como una señal de tránsito que se utiliza en intersecciones o zonas de conflicto en la infraestructura de transporte que indica al conductor, en caso de que no alcance a cruzar o incorporase de forma segura sin interferir en la maniobra de los vehículos de la otra corriente, la obligación de ceder el paso. La clave de la dificultad de implementar esta infracción está en la frase "*en caso de que no alcance a cruzar o incorporarse de forma segura sin interferir en la maniobra de los vehículos de la otra corriente*". Esta frase conlleva que si, por ejemplo, se da la situación de la figura 4.33, debes cederle el paso a un vehículo que viene por tu izquierda. Pero si este señaliza un giro a su derecha como lo mostrado en el caso de la figura, entonces podríamos invadir la intersección y realizar un giro a la izquierda porque no interferiríamos en la maniobra del vehículo.

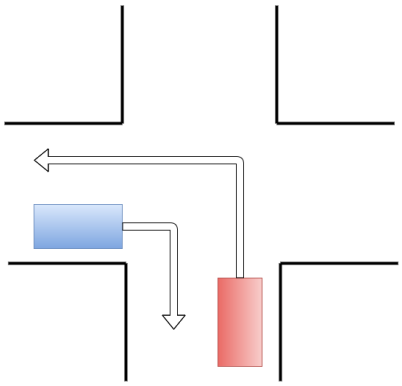


Figura 4.33 – Ejemplo de ceda el paso correcto

Esto conllevaba realizar la comprobación a posteriori. El bloque usado es muy similar al del Stop y también consta de un *trigger* a la entrada y otro sobre la intersección.

La idea consiste en que cuando el conductor entra en el *trigger* pequeño, se activa el *trigger* más grande que actúa como detector para almacenar una referencia de todos los vehículos que están en ese momento en la intersección. Para realizar la comprobación a posteriori, se lanza un método un número de segundos después para comprobar, por cada vehículo, si el nuestro ha interferido en su trayectoria. Para ello se comprueba la relación entrada-salida de cada vehículo y se compara con la relación entrada salida del vehículo del conductor. Toda esta operación se realiza mediante una Corrutina (*Coroutine*) que son métodos que permiten realizar su funcionalidad en distintos ciclos. En concreto, mediante la palabra reservada *yield* una corrutina devuelve el control a Unity, manteniendo su contexto actual. Esto permite, por ejemplo, realizar una comprobación sobre un enorme número de objetos que se recorren mediante un bucle en distintos frames, distribuyendo la carga computacional de forma más eficiente sin afectar a la funcionalidad.

4.3.6.5. Colisiones, velocidad, etc.

En esta sección se va a mostrar cómo se han desarrollado otras infracciones.

- Colisión coche y Colisión objeto.

Estas infracciones se implementan en la clase *RCCCarControllerV2* y utilizan el método *OnCollisionEnter*, que es un método lanzado por el motor de Unity cuando se detecta una colisión entre dos objetos. El sistema es tan sencillo como comprobar si el objeto que colisiona con el vehículo del conductor es otro vehículo comparando su Tag con la de los vehículos del *TrafficSystem* o es un objeto (edificios, bancos, farolas, etc.) comparando que la capa (*Layer*) a la que pertenece el *GameObject* sea *Buildings* o *SmallObject*.

- Exceso de velocidad y Velocidad demasiado baja.

La clase *VelocityControl* se encarga de gestionar la infracción de exceso de velocidad. Esta clase controla la velocidad a la que va el vehículo y la compara con una velocidad máxima establecida en esta misma clase. Esta velocidad máxima se modifica mediante el Prefab *VelocityControlSection* como se puede ver en la figura 4.34:

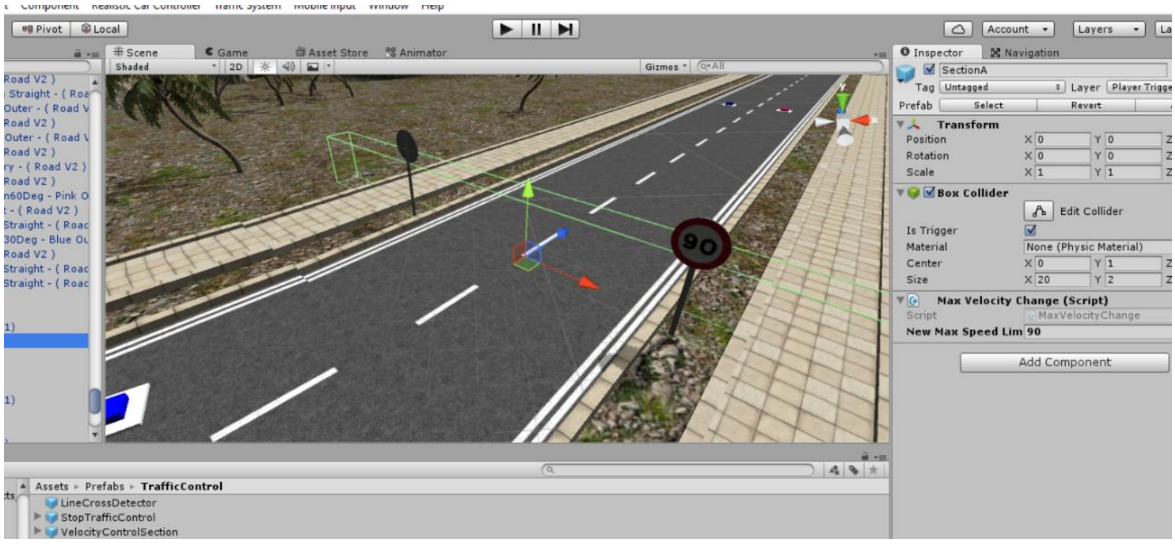


Figura 4.34 – Prefab de control de velocidad

Este *Prefab* contiene un script que permite modificar la velocidad máxima del script *VelocityControl*. De esta manera se pueden establecer puntos de cambio de velocidad para zonas con curvas, rectas, etc.

- Luces de noche y luces en el túnel

Esta infracción controla que se hayan encendido las luces en las simulaciones nocturnas. Básicamente lanza un método mediante *InvokeRepeating* que realiza la comprobación de que el vehículo del conductor lleva las luces encendidas.

En el caso de las luces del túnel, la norma indica que siempre deben encenderse las luces cortas cuando se entra en un túnel. Para gestionarlo, la clase *TunnelContext* se encargada de almacenar una variable que se pone a TRUE si durante el tiempo que el vehículo está en el túnel activa las luces. Si al salir del túnel esta variable no está a TRUE, se inserta una infracción.

- No respeta la distancia de seguridad

En la zona de carretera externa a la ciudad, se ha establecido una infracción que determina si el conductor está respetando la distancia de seguridad con el vehículo que tiene delante. El diagrama que representa el flujo del script *CarDistanceDetector* que es el encargado de realizar las comprobaciones es el mostrado en la figura 4.35:

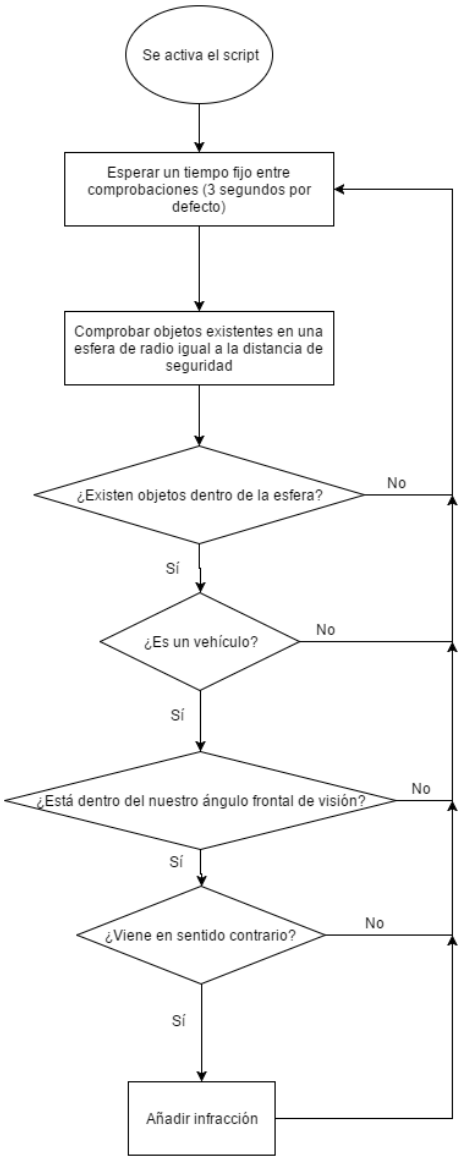


Figura 4.35 – Diagrama de flujo del script de control de distancia

Este script solo está funcional en las zonas fuera de la ciudad y va comprobando en intervalos regulares si existen vehículos por delante del conductor dentro de la distancia de seguridad. Esta distancia se suele establecer como 2 segundos por la velocidad de nuestro vehículo. Básicamente se utiliza esta distancia como el radio de una esfera de detección que devuelve todos los objetos dentro de ella. Luego se hace proceso para descartar aquellos que no podemos ver (por ejemplo, si están detrás de un edificio), los que están fuera del ángulo frontal de visión y lo los que vienen en sentido contrario. Si un vehículo no es eliminado tras estas comprobaciones es porque es un vehículo que está delante y dentro de la distancia de seguridad, que obviamente no estamos respetando y por lo tanto se mete una infracción.

4.3.7. El coche. Asset RCCController.

Para crear el funcionamiento general del vehículo se ha recurrido al *asset Realistic Car Controller* (BoneCracker Games, 2016). Este *asset* proporciona un script que gestiona todo el funcionamiento respecto al control del vehículo. Cambio de marchas, dirección o posición del centro de masas. Todo está gestionado mediante el script *RCCControllerV2*.

Sin embargo, este vehículo está orientado hacia una simulación más *arcade* y menos realista. La palabra *arcade* (Macmillan Dictionary, 2015) hace referencia a las máquinas recreativas que se jugaban en salones de videojuegos. Simulador *arcade* es un término que se les da a aquellos juegos que representan situaciones de simulación (conducir un vehículo, un avión o montar a caballo) pero centrando sus esfuerzos en la diversión y no en la fidelidad. Por ejemplo, *Need for Speed* (desarrollado por Electronic Arts) es una saga de videojuegos de conducción considerados simuladores *arcade* frente a, por ejemplo, Gran Turismo (Polyphony Digital, 2015) o iRacing (iRacing, 2015) que están considerados simuladores. Resumiendo, la diferencia está en cómo se orienta el desarrollo: en un simulador se busca un comportamiento fiel y realista del vehículo frente a un *arcade* que pretende dar experiencias más ficticias, como derrapes o giros imposibles.

Aunque no fue fácil dar con los puntos que modificar respecto a la física para dar un comportamiento más realista al vehículo, se ha conseguido obtener un comportamiento más adecuado a un simulador modificando algunos parámetros.

En concreto se han realizado cuatro cambios significativos:

- Modificar la conversión de la velocidad en el script. El script convertía la velocidad a kilómetros por hora multiplicando por 3 en el método *Engine*() y esto en nuestro caso es incorrecto. La velocidad en Unity vienen dada por unidades de Unity por segundo. En nuestro caso se ha intentado mantener una coherencia tal que una unidad de Unity equivalga a 1 metro. Por lo tanto, para convertir a kilómetros por hora hay que multiplicar por 3.6, no por 3. Esto hacia que la sensación de velocidad fuera incorrecta puesto que se mostraba una velocidad en Km/h menor de la realmente simulada.
- También se ha añadido el sistema de cálculo de consumo como lo hizo Paniagua (2015) en su trabajo de fin de grado. Sin embargo se ha modificado la variable que usaba en el intervalo de tiempo, de la constante Time.deltaTime al valor Time.fixedDeltaTime, debido a que nosotros realizamos el cálculo de consumo en la función *FixedUpdate*() en lugar de en la función *Update*().

- Modificar los ángulos de giro permitidos como se muestra en la figura 4.36:

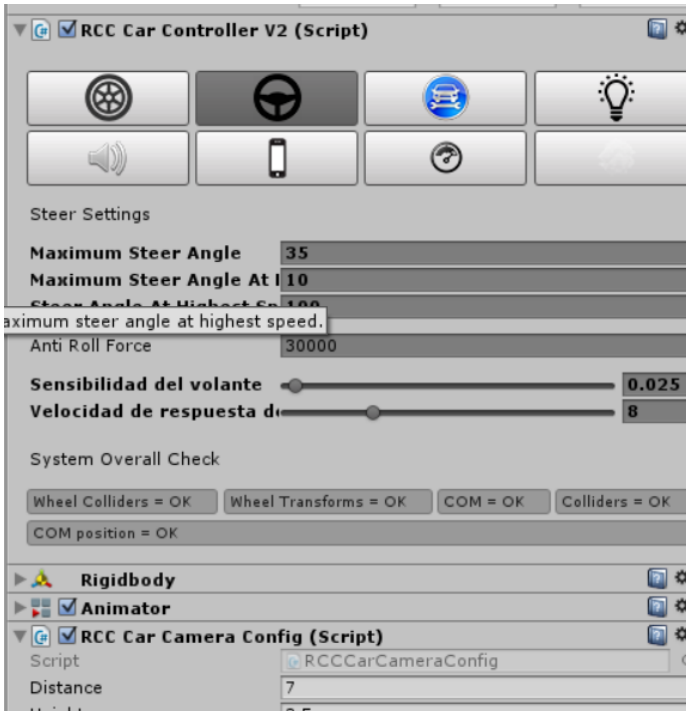


Figura 4.36 – Configuración relativa a los giros.

Por defecto, el script tiene como máximo ángulo de giro de las ruedas 40º. Esto es, el rango de entrada del volante se mapea linealmente entre -1 y 1 y -40º y 40º. Sin embargo, estos rangos, en combinación con materiales físicos (Manual Unity, 2015) con poca fricción hacían que el coche derrapara fácilmente. Primero se ha reducido el valor máximo a 35º. Luego para mejorar la sensibilidad del volante se extrajeron dos variables del script para que pudieran cambiarse desde el editor. La sensibilidad del volante establece a partir de qué valor entre 0 y 1 comienza a detectarse un giro del volante como un giro de las ruedas. Esto se debe a que un eje de entrada analógico tiene un rango donde se tiene que considerar neutro. Cuanto más reduzcamos ese rango antes el giro de nuestro volante se verá reflejado en un giro real del vehículo. El otro parámetro es la velocidad de respuesta que establece como de rápido se interpola entre dos entradas distintas de ángulos dentro del vehículo.

- Modificar los parámetros de deslizamiento de las ruedas para evitar que el coche derrape con tanta facilidad. Para ello se ha modificado el parámetro de *Stiffness* como se indica a en la figura 4.37.

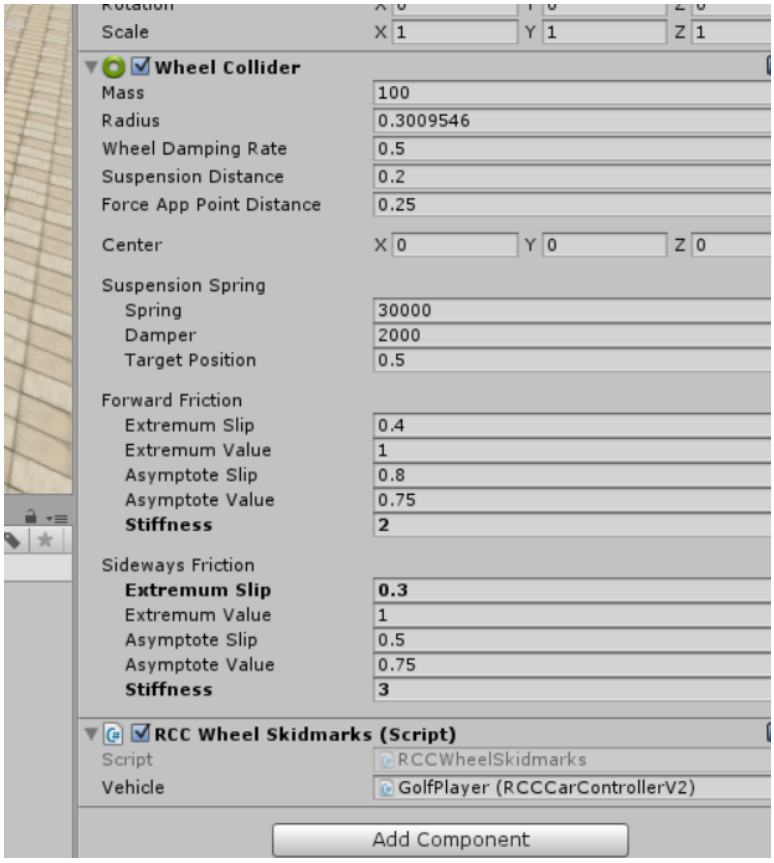


Figura 4.37 – Parámetros del Wheel Collider.

El parámetro *stiffness* es un multiplicador que actúa sobre el *Extremum Value* y el *Asymptote Value* (por defecto es 1). Permite cambiar la "dureza" de la fricción. Este es el parámetro que hay que modificar en tiempo de ejecución para simular distintos suelos (Manual Unity, 2015a).

Para saber qué fuerza se aplica en el punto de contacto el motor de físicas simula una curva como la mostrada en la figura 4.38.

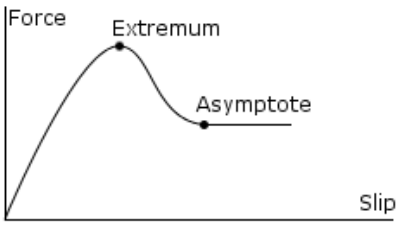


Figura 4.38 – Representación típica e la curva de fricción de una rueda (Manual Unity, 2015a)

El motor PhysX de físicas determina el deslizamiento del neumático basado en la diferencia de velocidad entre el neumático y la carretera. Entonces se usa este valor de deslizamiento para hallar la fuerza ejercida en el punto de contacto. En cualquier caso, se recomienda no alterar estos valores directamente y modificar el parámetro de *stiffness* para modificar la curva.

Para entender cómo actuaba este parámetro se realizaron dos pruebas:

- Poner el valor de *stiffness* a 0.1: Esto hace que la curva se haga menos alta y más plana y por lo tanto la fuerza ejercida era menor. El coche deslizaba al mínimo giro, como si fuera en hielo.
- Poner el valor de *stiffness* a un valor muy alto como 10: Esto conllevaba que el coche se volviera loco y comenzara a "botar". La fuerza ejercida era demasiado grande.

Moverse en el rango de valores entre 0.5 y 5 proporciona distintas respuestas que pueden ser realistas dependiendo del tipo de entorno que estemos simulando. En nuestro caso, un valor de 2 hacía que el coche deslizara muy pronto, mientras que un valor de 4 hacía que no se deslizara ni aunque se condujera a más de 100 km/h y se girara todo el volante. El valor final de 3 parecía otorgar un buen compromiso entre la velocidad y los deslizamientos de las ruedas.

4.4. Técnicas de mejora del rendimiento en aplicaciones realizadas con Unity.

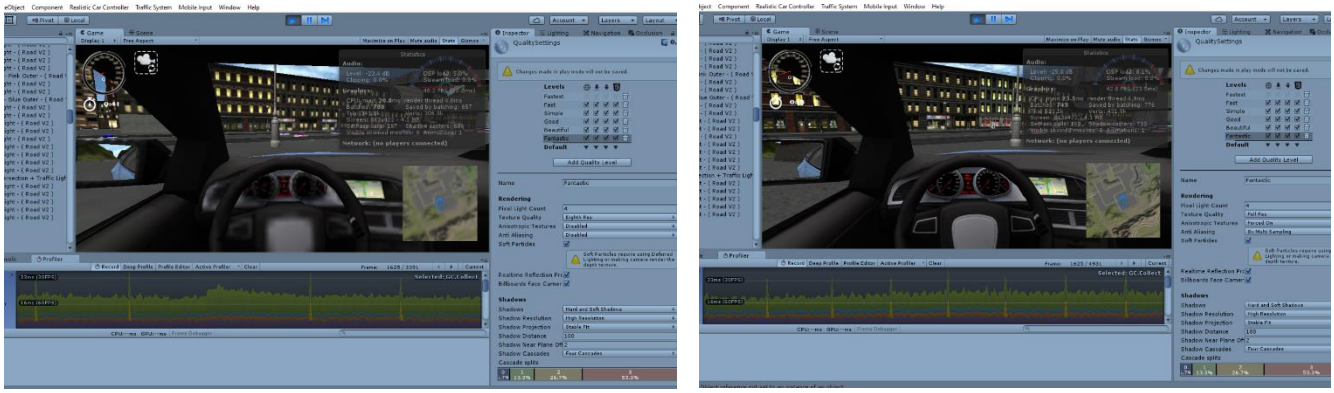
Crear aplicaciones con Unity es, dentro de lo que cabe, razonablemente simple. Gracias a su *Asset Store* es posible conseguir un enorme número de funcionalidades ya construidas y de esa manera solo tener que ir montando la aplicación. Dependiendo de la complejidad podría no ser necesario crear una sola línea de código. Sin embargo, esto puede provocar problemas de rendimiento que influyen directamente en el resultado final de una aplicación. Esta sección pretende ser un breve resumen del artículo "*How to plan optimizations with Unity*" (Wesolowski, 2013) que da a conocer algunas de las herramientas de las que dispone Unity para mejora el rendimiento de sus aplicaciones y que merece la pena conocer para saber cuándo y cómo usarlas.

Gestor de calidad (*Quality Manager*)

Unity dispone de un sistema para modificar el acabado final de la aplicación de forma que puedan ofrecerse distintas calidades finales. Se puede acceder a este menú en **Edit >Project Settings->Quality**. Este menú ofrece ciertas características que se pueden alterar para mejorar el rendimiento, generalmente a costa de una pérdida de calidad visual.

Algunas de las características modificables:

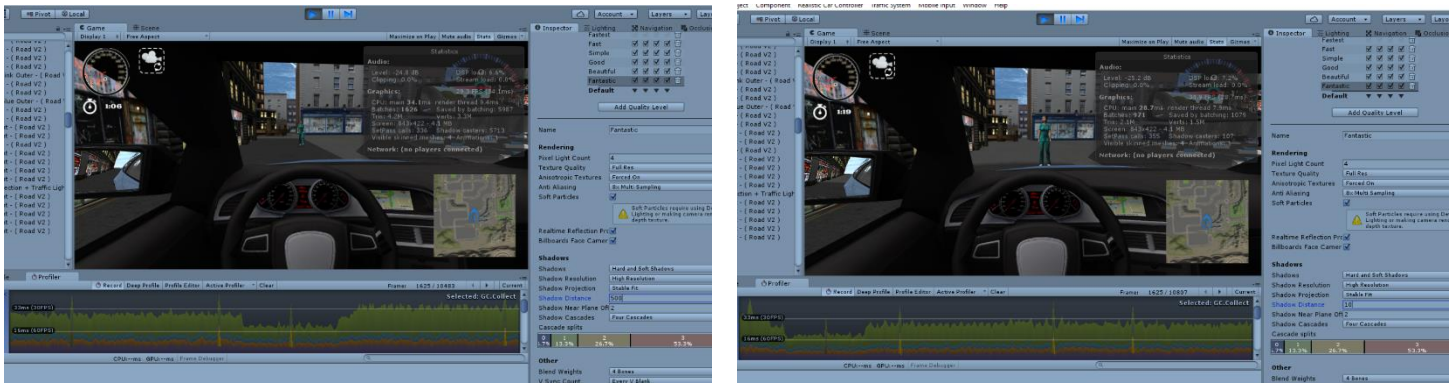
Calidad de textura (Texture Quality): Uno de los parámetros modificables consiste en cambiar la resolución de *renderizado* de las texturas usadas entre los valores 1/8, 1/4,1/2, o Resolución total (*Full Resolution*). En las figuras 4.39 y 4.40 se establece una comparación entre el juego con texturas a máxima resolución y a 1/8.



Figuras 4.39 y 4.40 – Ejemplos de una escena con resolución de textruras a 1/8 (izuierda) y a resolución completa (derecha).

Se aprecia una mejora tanto en el *framerate* como en la carga computacional del renderizado, sección verde de la gráfica del *Profiler*.

Distancia de dibujado de sombras (*Shadow Distance*): La distancia de sombreado es una configuración que modifica la distancia de *culling* de la cámara a la que se muestran las sombras de los objetos. En concreto, los *GameObjects* dentro de la distancia de sombreado tendrán sus sombras renderizadas mientras que el resto no. Las sombras tienen un enorme impacto en el rendimiento debido a la cantidad de procesamiento que requieren. Por ello, es importante balancear esta distancia en función de los resultados visuales. La distancia de sombreado y otras configuraciones de las sombras se pueden modificar en tiempo de ejecución mediante código, lo que permitiría adaptarlas a varios tipos de situaciones. En las figuras 4.41 y 4.42 se observa cómo la distancia de dibujado de las sombras puede suponer una mejora significativa del rendimiento, aunque a costa de una pérdida de inmersión similar a reducir la calidad de las texturas.



Figuras 4.43 y 4.44 – Escena con una distancia de dibujado de 500 unidades (izquierda) y con una distancia de 10 unidades (derecha).

Se observa cómo la sombra está recortada en la figura 4.44 pero también una mejora significativa del *framerate* (prácticamente 5 fps más).

Capas (*Layers*): Ya se ha mencionado con anterioridad la existencia de capas. Las capas permiten agrupar objetos con funcionalidades similares de forma que se pueden actuar sobre ellos en grupo.

Distancia de sacrificio de capas (*Layer Cull Distances*): En Unity las cámaras no renderizan objetos más allá de su valor de *Far clipping plane*. Sin embargo, existe una manera mediante código de establecer una distancia menor para ciertas capas. En la figura 4.45 se muestra un script sacado de la documentación de Unity de cómo modificar las *Layer Cull Distances*.

```
function Start () {
    var distances = new float[32];
    // Set up layer 10 to cull at 15 meters distance.
    // All other layers use the far clip plane distance.
    distances[10] = 15;
    camera.layerCullDistances = distances;
}
```

Figura 4.45 – Establecer las *Layer Cull Distances*

Utilizar las *Layer Cull Distances* adecuadamente demuestra ser muy beneficioso para el rendimiento de la aplicación. Por ejemplo, en nuestro caso los objetos pequeños que están lejos de la cámara tienen una distancia menor que el *Far Clipping* de la cámara. Esto permite que estos objetos no sean *renderizados* a distancias que no son apreciables a nivel visual dentro del juego. Esto es algo que habría que tener en cuenta previamente al diseño de los niveles de un juego o aplicación.

Cámara (*Camera*): la cámara en Unity son los ojos y oídos del jugador. Existen varias propiedades que pueden modificarse para mejorar el rendimiento.

Far Clipping Plane: Determina la distancia de dibujado máxima de la cámara. Obviamente este es un valor clave en el rendimiento. Todos los objetos fuera del rango de la cámara no se *renderizan*, lo que implica que cuanto menor sea este valor, menos objetos se *renderizan* y por lo tanto menor es la carga computacional.

La distancia de dibujado máxima se puede modificar fácilmente en tiempo de ejecución y vamos a ver cómo afecta al tiempo renderizado en las figuras 4.46, 4.47, 4.48 y 4.49.

Figuras 4.46, 4.47, 4.48 y 4.49 – Efecto de la distancia máxima de dibujado de la cámara. Valores establecidos: 500 (arriba izquierda), 750 (arriba derecha), 1000 (abajo izquierda) y 1500 (abajo derecha).

En los datos se observa que el *framerate* es mejor cuanto menor es la distancia máxima de la cámara puesto que tiene que renderizar menos objetos. Sin embargo, existe un punto a partir del que aumentar este valor deja de influir. Este valor va a ser la diagonal de nuestra escena, que es el valor más grande en el que pueden entrar objetos en el rango de visión de la cámara. Sin embargo, poner este valor suele ser inviable en la mayoría de simulaciones. La clave consiste en crear los niveles con coherencia de forma que se pueda obtener una distancia máxima lo menor posible que optimice el rendimiento al máximo.

Render Path: El *Render Path* indica cómo Unity maneja el *renderizado* de luces y sombras. Unity ofrece varios tipos de renderizado y cada uno tiene distintos costes respecto a CPU y GPU. Los tipos soportados (del más costoso al menos) son *Deferred*, *Forward*, y *Vertex Lit*. Comprender bien cómo funcionan estos *Render Path* permite elegir el más adecuado para la aplicación a crear (Manual Unity, 2015d).

Occlusion Culling: La técnica de *Occlusion Culling* deshabilita el renderizado de los objetos, no solo fuera del *Far Clipping Plane* de la cámara sino también aquellos objetos que están ocultos por otros. Esto puede ser enormemente beneficioso en términos de rendimiento porque disminuye el número de objetos a *renderizar* por la tarjeta gráfica, pero esto no es simple de implementar. Implica diseñar los niveles de forma adecuada para poder aprovecharse de esta característica.

Tengamos presenta la terminología relacionada con esta técnica:

Occluder – Un objeto marcado como *occluder* actúa como una barrera que previene objetos marcados como *ocludee* de ser renderizados.

Occludee – Marcar un objeto como *occludee* indica a Unity que no debe *renderizar* ese objeto si está tapado por un *occluder*.

Por ejemplo, todos los objetos dentro una casa podrían ser marcados como *occludees* y la casa como *occluder*. Si el jugador está fuera de la casa todos los objetos interiores marcados como *occludee* no serían renderizados. Esta técnica requiere que los desarrolladores hagan mucho trabajo manual. Hay que marcar los objetos de forma adecuada y además requiere *testear* bien que ningún objeto desaparezca de forma inesperada. Esta técnica no es algo a comparar de forma sencilla, ya que depende de la existencia de objetos que puedan ocultar a otros. Por ejemplo, en una carretera muy larga con edificios a los dos lados, esta técnica no aportaría prácticamente nada, puesto que ningún edificio ocultaría al resto completamente. Como caso opuesto, si estuviéramos de frente a un enorme edificio que oculte todo lo que hay detrás nos estaríamos beneficiando en gran medida de esta técnica. Es una solución que solo puede estudiarse con pruebas intensivas de la aplicación por un grupo amplio de usuarios.

Level of Detail (LOD): El *Level of Detail (LOD)* permite adjuntar a un objeto múltiples mallas y cambiar entre ellas según la distancia del objeto a la cámara. Supongamos por ejemplo un edificio de gran tamaño. Cuando está cerca de la cámara podemos querer que se muestre a pleno detalle, pero cuando esté lejos, aunque no queramos que desaparezca, posiblemente no nos importe que se muestre una malla menos compleja. Esto se puede obtener mediante el componente LOD de Unity.

4.5. Documentación de la API del Simulador.

Es inviable mostrar todo el código del simulador en esta memoria, por ello se ha optado por dar explicaciones funcionales evitando entrar en el código. Aun así, para comprender el funcionamiento completo de ciertas clases o eventos explicados se puede requerir tener el código delante para entenderlo y visualizarlo de una forma rápida y sencilla. Se ha intentado documentar el código siguiendo ciertas convenciones de forma que se pudiera generar un documento de la API del mismo.

Este documento está hecho con fines educativos y por ello se ha optado por sacar los elementos con visibilidad privada. La documentación solo se refiere a los scripts creado para este proyecto, no a todos los utilizados. Los documentos son accesibles en la carpeta Documentation/RJSimuladorUnityDocs/Doxygen.

5. Desarrollo del módulo de Drupal para gestión, almacenamiento y procesado de datos.

Para crear la plataforma web que se deseaba se ha optado por recurrir a un *CMS* muy extendido para el desarrollo web como es Drupal.

Drupal es un sistema de gestión de contenido modular, muy configurable y fácilmente extensible.

Es un programa de código abierto, con licencia GNU/GPL, escrito en PHP, desarrollado y mantenido por una activa comunidad de usuarios.

5.1. Funcionalidades de Drupal.

Recordemos algunas funcionalidades más características de Drupal:

- **Contenido flexible:** Fácil creación de contenido con técnicas WYSIWIG.
- **Mejor diseño de plantillas:** Una capa que permite separar la lógica de los temas y el diseño.
- **Accesible.** Implementa sistemas de control de acceso y gestión por roles.
- **Gestión de ficheros**: Permite la gestión visual de ficheros, permisos de acceso, control sobre los tamaños de ficheros y de imágenes.
- ***Testing:*** Testeado de código en el núcleo y los módulos contribuidos por la comunidad.
- **Soporte de base de datos mejorado.**
- Una **comunidad activa** y fabulosa que proporciona de forma gratuita un enorme número de módulo para extender las funcionalidades básica del núcleo.
- **Extensibilidad**: Su estructura modular y su sistema de *hooks* permiten abarcar todo tipo de proyectos desde pequeños blogs hasta enormes plataformas web, incluyendo tiendas *eCommerce*.

5.2. Requisitos e instalación previa básica.

Como Drupal está escrito sobre PHP y requiere de base de datos, para instalar Drupal 7 lo más cómodo es partir de un *stack* LAMP (Linux-Apache-MySQL-PHP) que permita la ejecución del código.

XAMPP (XAMPP, 2015) es un programa que viene con un instalador que monta un servidor PHP en nuestra máquina Windows. Para Linux es aún más sencillo pudiendo recurrir directamente a su gestor de paquetes *apt* si estamos en Debian o *yum* en RedHat, aunque no se tratarán aquí.

Una vez instalado XAMPP, podremos meter nuestras aplicaciones PHP en su carpeta htdocs, generalmente C:\xampp\htdocs. Luego basta con activar los *daemon* del servidor Apache y del servidor MySQL y ya tenemos un entorno local en el poder realizar nuestra instalación.

Desde la web del proyecto Drupal (Drupal, 2015a) podemos descargar la última versión de Drupal 7, descomprimir la carpeta en el directorio anterior y acceder desde un navegador usando la dirección http://localhost. La instalación de Drupal es simple y guiada, basta con seguir los pasos. Sin embargo, Drupal requiere de una base de datos

para la persistencia. Aunque Drupal soporta varias bases de datos diferentes, es común el uso de MySQL que además se nos habrá instalado con XAMPP.

Desde http://localhost/phpmyadmin podemos crear una base de datos de forma sencilla como se ve en la figura 5.1.

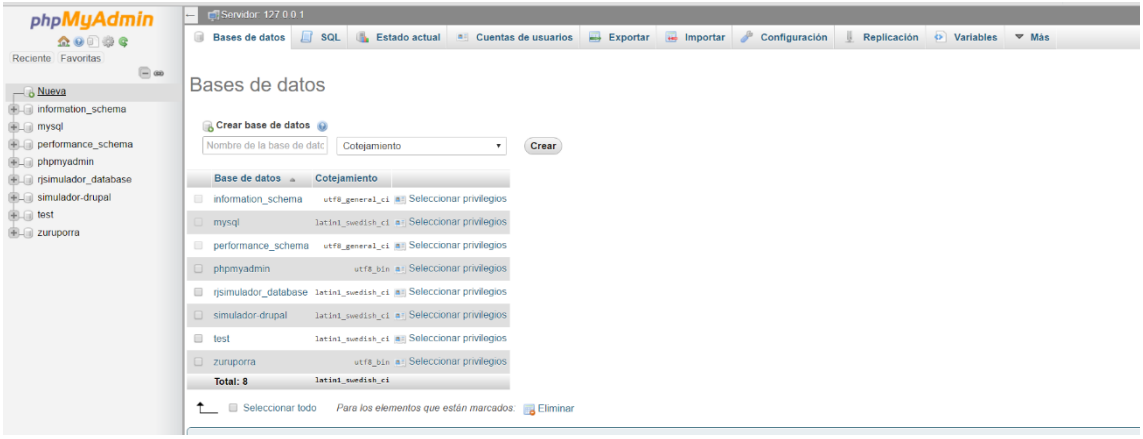


Figura 5.1 – Crear una BBDD en PHPMyAdmin

Una vez instalado Drupal procederemos a la instalación de los módulos que serán necesario para la ejecución de nuestro propio módulo. Estos serán lo que comúnmente se denominan dependencias.

Los módulos en concreto (y donde encontrarlos) son:

- Libraries API: https://www.drupal.org/project/libraries
- Services: https://www.drupal.org/project/services
- Charts: https://www.drupal.org/project/charts
- XAutoload: https://www.drupal.org/project/xautoload

Es posible que estos módulos requieran de otros a su vez. Es muy sencillo encontrar todos estos módulos en la web de Drupal (Drupal, 2015a).

Instalar un módulo en Drupal es tan sencillo como acceder a la dirección http://{entorno_local}/admin/modules/install y desde ahí sólo hay que ir añadiendo uno a uno los módulos descargados de la web.

Una vez subidos hay que acceder a la sección de *Modules* como administradores y activamos los módulos antes mencionados si no se han activado ya.

Respecto al módulo *Charts*, este sirve como un puente, una interfaz entre ciertas librerías de gráficos y el sistema de renderizado de Drupal. Entre otras cosas, permite trabajar con la librería *Highcharts*, que es la que se ha utilizado en este proyecto. Podemos descargarla desde http://www.highcharts.com/download. En Drupal las librerías externas se gestionan con el módulo *Libraries API* y se meten en la carpeta <root>/sites/all/libraries.

En concreto la librería Highcharts debe ir en <root>/sites/all/libraries/highcharts/js/highcharts.js.

Una vez seguidos los pasos anteriores ya se dispondría de un entorno Drupal con los módulos necesarios para el desarrollo del módulo de gestión de datos de simulador.

5.3. Crear un módulo

Esta sección nos permitirá entender que es un módulo Drupal y cómo es su estructura básica.

En Drupal, un módulo es un conjunto de archivos de programa, imágenes, datos, hojas de estilo, etc., organizados en un directorio, cuyo propósito es realizar tareas ligadas con la solución de una necesidad o de un grupo de necesidades afines.

Estructura general de un módulo

- **Nombre de un módulo**

Todo módulo tiene dos nombres, que eventualmente pueden ser iguales, pero que deben ser considerados como diferentes: Un nombre amigable y un nombre técnico.

El nombre amigable es un nombre con sentido social y cuyo propósito es presentar al módulo ante la comunidad para que sea identificado y entendido por sus usuarios. Eg: "Mi módulo"

El nombre técnico está dirigido a la máquina, es humanamente legible, pero se orienta a ser entendido por "el motor de Drupal" y debe ser único en cada sitio en que sea instalado. Debe ser escrito usando únicamente letras minúsculas (sin eñes ni tildes), números y guiones bajos. Eg: "mi_modulo".

- **Contenido de un módulo**

Todo módulo debe tener como mínimo dos archivos (*.*info* y *.*module*) alojados en un mismo directorio y, los tres (directorio y archivos) deben tener en común el nombre técnico del módulo (Drupal, 2015b).

Ejemplo: Supongamos que acabamos de escribir e instalar un módulo. La estructura de archivos resultante se parecerá a la siguiente:

```
/directorio_del_sitio/sites/all/modules/mi_primer_modulo
mi_primer_modulo.info
mi_primer_modulo.module
```

El módulo, dependiendo de su complejidad podrá contener otros archivos y subdirectorios anidados, como por ejemplo: archivo de instalación (*.*install* que es opcional pero también tiene que tener el nombre técnico del módulo), hojas de estilo, temas por defecto, scripts, etc.

Y, salvo los tres archivos señalados (*info*, *module* e *install*) el resto puede ser organizado a gusto del desarrollador del módulo. Sin embargo, es conveniente mantener presente que los nombres de archivo respeten la sintaxis de nombre técnico, puesto que un archivo es ofrecido al sistema para que sea abierto y "mostrado" por un controlador. Los nombres técnicos nunca deben contener caracteres especiales, pues pueden provocar fallos en algunos sistemas operativos, entre los que el menor error es no responder correctamente a los enlaces web.

- **El archivo de declaración de un módulo**

El archivo "nombre_tecnico_del_modulo.info" es conocido como archivo de declaración del módulo. Y su propósito es "presentar" el módulo ante el motor de Drupal. Basta con que se escriba y ubique en la posición recomendada para que Drupal sepa de la existencia del módulo y pueda ofrecerlo en el menú de construcción del sitio para ser activado.

Contenido del archivo de declaración

El archivo de declaración *.info*, es un archivo de texto plano debe contener como mínimo los siguientes parámetros: nombre amigable, descripción y versión de Drupal a la que está dirigido el módulo. No se necesita el nombre técnico pues este es ofrecido por el nombre del archivo. Existen otros parámetros que pueden declararse en el archivo de declaración.

Descripción de los parámetros del archivo:

- name (Obligatorio): Nombre amigable. Se recomienda que sólo su primera letra sea mayúscula, y en caso de requerir tildes debe usar entidades HTML y ser escrito entre comillas.
- description (Obligatorio): Descripción breve del propósito del módulo (para qué sirve, qué hace, etc.)
- core (Obligatorio): Familia de versiones de Drupal a la que está dirigido puede ser 6.x, 7.x, 8.x, etc. Establece restricciones de compatibilidad.
- dependencies: Establece las dependencias del módulo.
- configure: Establece el enlace desde donde se puede acceder a la configuración del módulo en caso de que exista.

Ejemplo de fichero *.info*:

```
name = Mi modulo
description = Mi modulo para Drupal
core = 7.x
dependencies[] = ctools
configure = admin/config/content/firstmod
```

- **El archivo de programa principal de un módulo**

El archivo "nombre_tecnico_del_modulo.module" es un archivo de programa escrito en lenguaje PHP típico, que ofrece las funciones y objetos necesarios para que el módulo sea operativo.

Tiene tres características fundamentales:

1. Nunca usa la marca terminadora ?> típica de los archivos php. Pero si debe usar la inicial: <?php
2. Implementa mediante puntos de enganche *hooks* las funciones necesarias para que Drupal pueda activar y ejecutar sus servicios. Antes de seguir hay que recordar que es un *hook* en Drupal. Los *hooks* son personalizaciones de acciones. O en lenguaje llano, los *hooks* permiten modificar el comportamiento de un método o función sin modificar su código, siempre que esté preparado para ello.
3. Puede incluir funciones y objetos adicionales para ser llamados y empleados mediante código PHP desde otros módulos o desde páginas, cuando el módulo PHP ha sido habilitado para ser usado como formato de entrada al escribir las páginas del sitio.

- **El archivo de instalación de un módulo**

Al igual que el archivo de programa principal, se trata de un archivo de programa PHP sin cierre de etiqueta en la línea terminadora. Su función es realizar tareas que deben ejecutarse sólo una vez, necesarias para la correcta operación del

módulo, presentar tareas de actualización, definir la estructura de base de datos requerida, crear e iniciar las tablas necesarias y eventualmente ofrecer mecanismos de desinstalación. En una sección posterior se comentará un poco más acerca de este fichero.

5.4. Desarrollo del módulo: rjsimulador.

Las pretensiones de este módulo consisten en crear un punto conexión remota entre nuestra aplicación de simulación y la plataforma web. Para ello se va a utilizar el módulo *Services* para ofrecer una API REST que permita al simulador enviar datos en formato JSON para ser almacenados en el servidor.

En primer lugar es necesario configurar el módulo *Services*. Esta configuración es bastante simple y consiste crear un *endpoint*, decir el tipo de datos que va a soportar el servidor, establecer la autenticación y luego activar los recursos que se desea que estén funcionales.

Desde structure/services/add se va añadir el *endpoint* (Figura 5.2). Para ello es necesario proporcionar un nombre, un tipo de servidor y activar la autenticación:

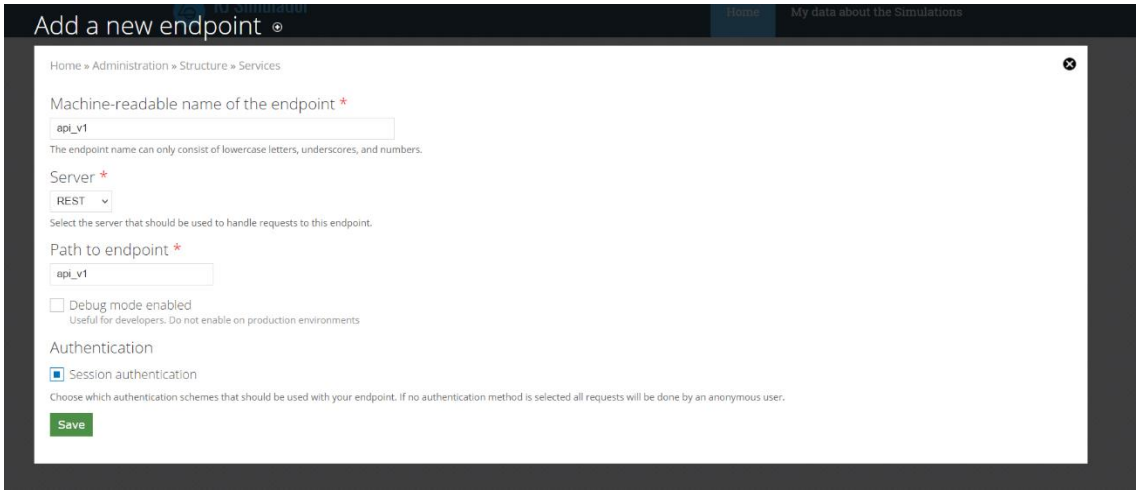


Figura 5.2 – Pantalla de creación de un nuevo *endpoint*

Si en el desplegable de servidor no aparece nada, es porque al instalar el módulo *Services* no se activó el módulo que proporciona este tipo de servidor. Basta con ir a la sección de módulos y activarlo.

Luego desde la pestaña *Server* hay que seleccionar los tipos de peticiones y respuestas soportadas (Figura 5.3). En el caso que nos ocupa se ha optado por JSON por ser un tipo de dato manejable y posiblemente de los más usados en la comunicación web a día de hoy junto con XML.

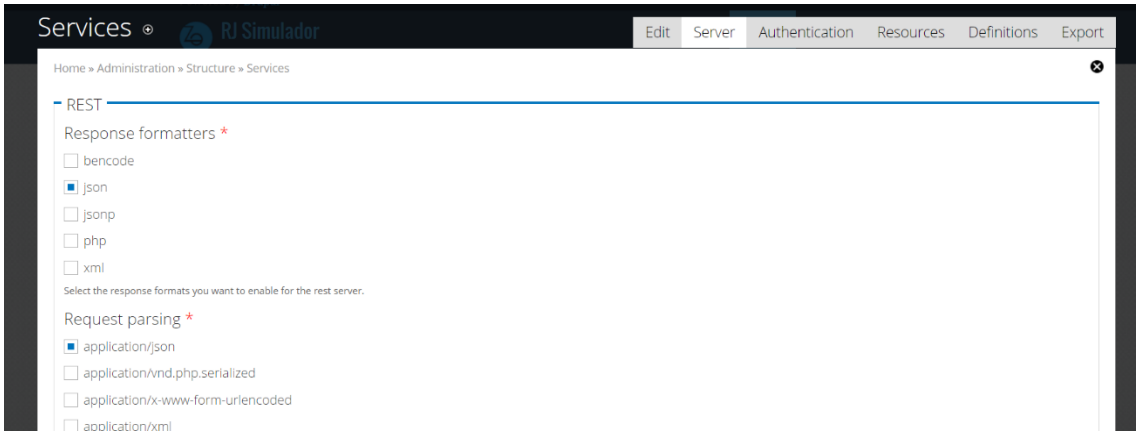


Figura 5.3 – Establecimiento de los tipos de datos soportados

Luego se van a activar los recursos deseados, que en nuestro caso serán los de recursos asociados al usuario (recursos *user*), que nos permitirán hacer *login* (validarnos en la plataforma), *update*, *create*, etc. Se pueden activar los recursos como se aprecia en la figura 5.4.

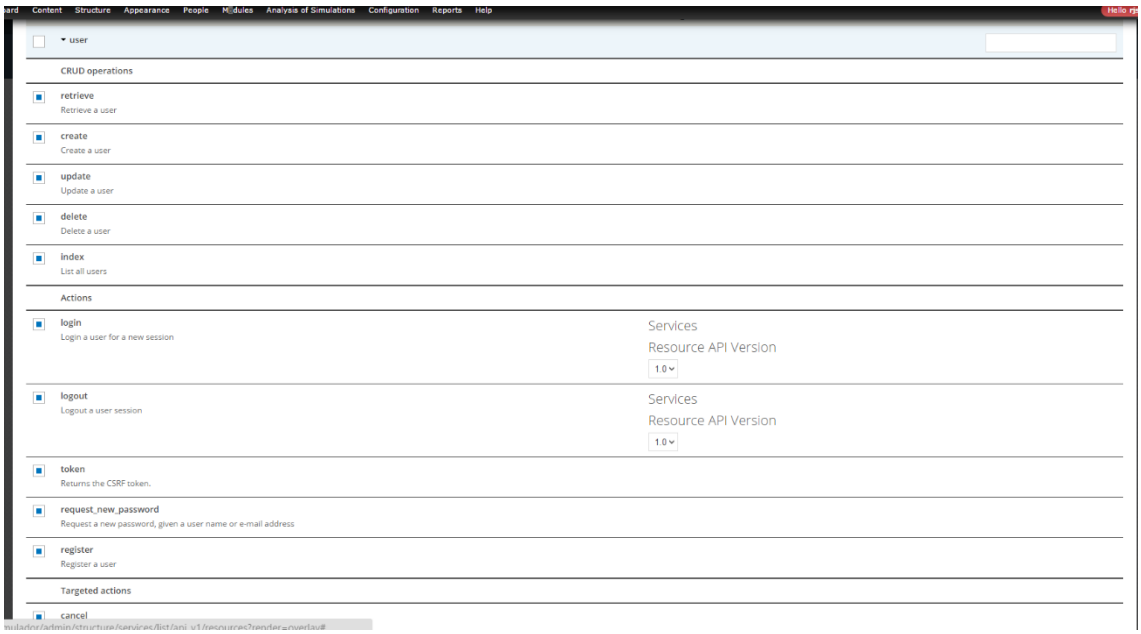


Figura 5.4 – Recurso activos

Como se ha mencionado anteriormente, un módulo en Drupal requiere de al menos dos ficheros que son el fichero de declaración del módulo (*.info*) y el fichero de programa principal del módulo (*.module*).

Se va a proceder a ver cómo están implementados esos ficheros en nuestro módulo.

Fichero rjsimulador.info

```
name = Análisis de datos de simulación
description = Gestión de los resultados recibidos desde la aplicación
de conducción de Unity.
package = UVa Simulador
core = 7.x
version = 7.x-1.1
; Configuration
configure = admin/config/media/rjsimulador
; Dependencies
dependencies[] = libraries
dependencies[] = services
```

```
dependencies[] = charts
dependencies[] = xautoload
```

Se observa que el fichero contiene los parámetros esperados como name, description o core. Además establece un enlace de configuración y unas dependencias.

Fichero rjsimulador.module:

```
      <?php
use Drupal\rjsimulador\Factory\FactoryDataManager;
use Drupal\rjsimulador\Partida;

/**
 * Implementation of hook_permission().
 */
function rjsimulador_permission() {
  return array(
    'crear partidas' => array(
      'title' => t('Create new Partidas'),
      'description' => t('Allows users with this permission to store
data of a game.'),
    ),

      […]

  );
}

/**
 * Implementation of hook_menu().
 */
function rjsimulador_menu() {
  $items = array();
  $items['simulaciones'] = array(
    'title' => 'My data about the Simulations',
    'file' => 'rjsimulador.pages.inc',
    'page callback' => 'rjsimulador_simulaciones_page',
    'access callback' => 'check_user_access_has_saved_partidas',
    'access arguments' => array('ver info partidas'),
    'menu_name' => 'main-menu',
    'type' => MENU_NORMAL_ITEM,
  );

      […]

  $items['admin/simulaciones_analysis/%/simulaciones/%/partidas/%'] =
array(
    'title' => 'User with @uid: Partida Data',
    'title arguments' => array('@uid' => 2),
    'file' => 'rjsimulador.pages.inc',
    'page callback' => 'rjsimulador_partida_page',
    'page arguments' => array(6, 2),
    'access callback' => 'check_user_access_to_partida',
    'access arguments' => array('comparar info partidas', 6, 4, 2),
    'type' => MENU_NORMAL_ITEM,
  );

      […]
```

```
  return $items;
}

function check_user_access_has_saved_partidas($permission, $uid =
NULL) {

      […]

}

function check_user_access_to_partida($permission, $id_partida,
$id_simulacion, $uid = NULL) {

      […]

}

/**
 *  Implements hook_services_resources().
 */
function rjsimulador_services_resources() {
  // Include resources definitions
  module_load_include('inc', 'rjsimulador',
'resources/rjsimulador_partida.resource');
  $resources = array();
  $resources += partida_resource_definition();
  return $resources;
}

      […]


/**
 * Implements hook_user_delete().
 */
function rjsimulador_user_delete($account) {
  $provider = FactoryDataManager::createDataProvider();
  try {
    $usuario = $provider->loadSimulatorUser($account->uid);

    if ($usuario->countPartidas() == 0) {
      throw new LogicException("El usuario borrado no tiene Partidas
almacenadas que eliminar.");
    }

    // Creamos una transacción para eliminar la partida; si algo falla
hacemos rollback
    $transaction = db_transaction();
    try {
      foreach($usuario->getListaSimulaciones() as $simulacion) {
        foreach($simulacion->getListaPartidas() as $partida) {
          $partida->remove();
        }
      }
    } catch (Exception $e) {
      $transaction->rollback();
      throw $e;
    }
  } catch (LogicException $le) {
    watchdog_exception('rjsimulador', $le, 'There are no Partidas to
delete for user with UID @uid and username @username.', array(
```

```
      '@uid' => $account->uid,
      '@username' => $account->name
    ), WATCHDOG_INFO);
  } catch (Exception $e) {
    watchdog_exception('rjsimulador', $e, 'Error removing Partidas
from user with UID @uid and username @username.',
      array('@uid' => $account->uid, '@username' => $account->name),
WATCHDOG_ERROR);
  }
}
```

Se van a presentar algunas nociones relativas a la estructura del fichero principal del módulo. Sin embargo no se puede hacer una descripción detallada de todas y cada una de las funciones, para ello se proporciona una documentación completa de funciones, clases y todos los ficheros fuente.

Las funciones que empiezan por el nombre del módulo rjsimulador son implementaciones de diversos *hooks*. Como se ha mencionado antes, un *hook* nos permite modificar una funcionalidad en concreto de Drupal.

Aquí se pueden apreciar al menos 4 implementaciones.

1. **hook_permission**: Permite establecer nuevos permisos para el módulo. Estos permisos serán gestionables por roles desde la sección de *Permissions* del *Backend* de la plataforma (Figura 5.5).

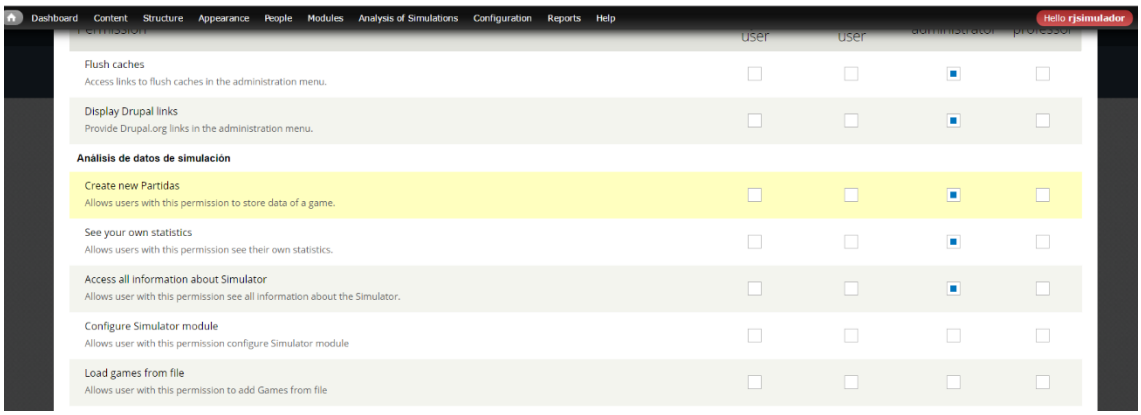


Figura 5.5 – Página de permisos del *Backend*

2. **hook_menu**: A pesar del nombre, este *hook* permite la creación de rutas. Establece una ruta y debe devolver un array asociativo donde la nomenclatura de las claves es vinculante a una funcionalidad. Podemos mencionar entre ellas:
   - "title": Obligatoria. Cadena sin traducir con el nombre de la página.
   - "title callback": Función para generar el nombre del título.
   - "title arguments": Argumentos enviados a la función *title_callback*.
   - "description": Descripción del ítem de menú sin traducir.
   - "page callback": Función que devolverá un array de renderizado con la página a mostrar.
   - "page arguments": Array de argumentos a pasar a la función anterior.
   - "access callback": Una función que debe devolver TRUE si el usuario puede acceder a la página o FALSE en caso contrario.
   - "access arguments": Un array de argumentos a pasar a la función superior.

3. **hook_services_resource**: Será el encargado de crear los recursos REST como los que se han visto anteriormente. Devuelve un array muy similar a hook_menu. Sin embargo, tiene algunas peculiaridades que permiten establecer ciertas

configuraciones relativas a servicios web en concreto, como indicar el tipo de verbo HTTP soportado por el recurso, de donde leer los parámetros recibidos, etc. Veremos el fichero donde está la función *partida_resource_definition*() más adelante.

4. **hook_user_delete**: Permite realizar opciones sobre un usuario cuando va a ser eliminado. En nuestro caso se ha usado para que cuando se elimine un usuario por completo del sistema, se elimine las partidas asociadas al mismo de forma que no queden datos huérfanos en la base de datos.

Lo más interesante cuando creamos un módulo de Drupal es que solo los ficheros **.module* son los escaneados por Drupal. Esto fuerza a que las implementaciones que requieren del núcleo de Drupal como los *hooks* estén declarados ahí. Sin embargo, esto también permite que el resto del módulo pueda ser programado con bastante libertad.

Por ello se decidió optar por el paradigma de orientación a objetos para estructurar nuestro módulo, sobre todo como preparación para el nuevo diseño de Drupal 8 que pretende seguir este paradigma en la medida de lo posible. Actualmente el equipo PHP FIG (Framework Interoperability Group) pretende establecer unas convenciones que permitan al lenguaje evolucionar y adaptarse a los nuevos tiempos (PHP FIG, 2015a). Por ello, en el desarrollo de este proyecto se ha optado por seguir estas reglas de cara a entender por qué se ha optado por ellas a nivel global y poder aplicarlas en el futuro si fuera necesario.

Para finalizar se va presentar como se ha estructurado el resto de nuestro proyecto (Figura 5.6):

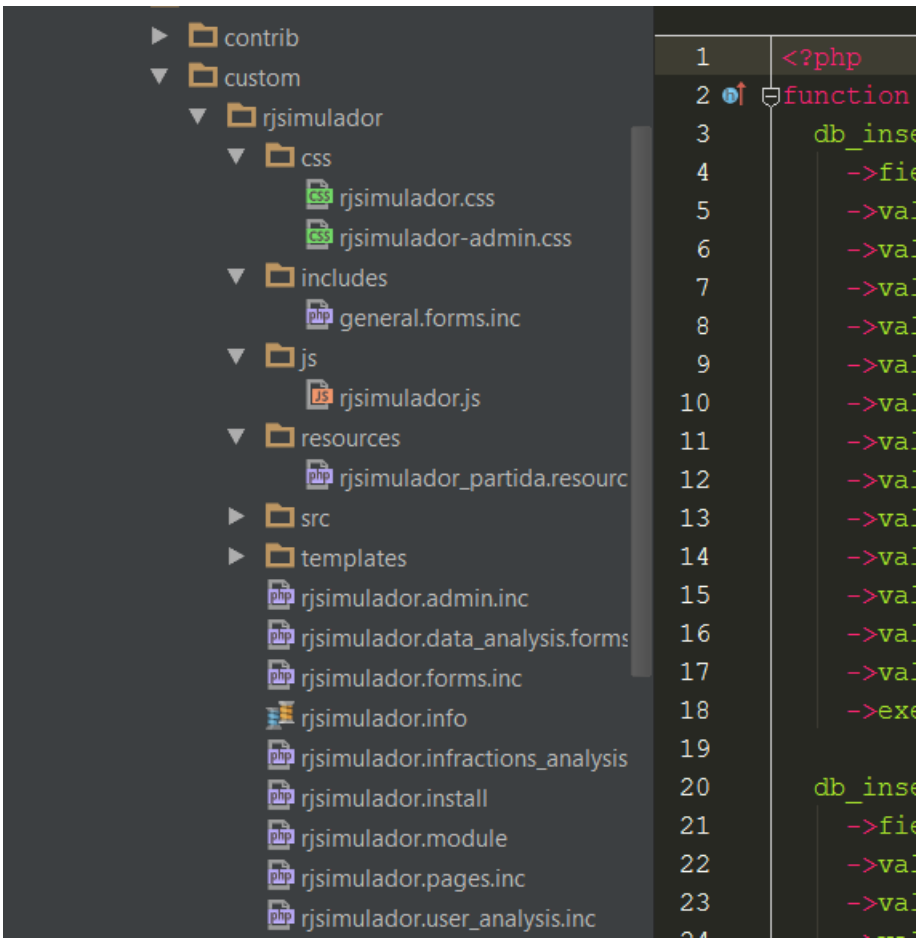


Figura 5.6 – Estructura de ficheros del módulo rjsimulador

Se observan varios fichero aparte del fichero *.info* y el *.module*.

En concreto se puede ver que está el fichero *.install*, reconocido por Drupal (en el caso de que exista) y que permite realizar tareas que deben ejecutarse sólo una vez. Este fichero permite implementar los *hooks* relacionados con las acciones de instalación (y desinstalación) del módulo. Principalmente se pueden nombrar tres *hooks* (aunque existen otros):

- **hook_schema**: Permite utilizar la *Schema* API (Drupal, 2015c) para crear las tablas cuando se instala un módulo y borrarlas cuando se desinstala.
- **hook_install**: Permite realizar operaciones de inicialización como insertar datos en las tablas, crear variables, etc.
- **hook_uninstall**: Permite realizar operaciones durante el desinstalado de un módulo.

Vamos revisar brevemente como es nuestro fichero de instalación.

Fichero rjsimulador.install

```
      <?php
function rjsimulador_install() {
  db_insert('rjsim_infracciones')
    ->fields(array('nombre_infraccion'))
    ->values(array(1, 'Atropello a un peatón'))
    ->values(array(2, 'Colisión con un vehículo'))

      […]

    ->values(array(13, 'Se ha cruzado la línea continua de la
carretera'))
    ->execute();

  db_insert('rjsim_simulacion')
    ->fields(array('nombre_simulacion'))
    ->values(array(1, 'Simulación 1: Recorrido por ciudad'))
    ->values(array(2, 'Simulación 2: Recorrido por el exterior'))

[…]

    ->execute();
}

function rjsimulador_schema() {
  $schema['rjsim_partida'] = array(
    'description' => t('Listado de partidas jugadas'),
    'fields' => array(
      'id_partida' => array(
        'description' => 'ID de la partida',
        'type' => 'serial',
        'unsigned' => TRUE,
        'not null' => TRUE
      ),
      'uid' => array(
        'description' => 'UID del usuario que hizo al partida',
        'type' => 'int',
        'unsigned' => TRUE,
        'not null' => TRUE,
        'default' => 0
      ),
      'fecha' => array(
        'description' => 'Timestamp de la fecha en la que se hizo la
```

```
partida',
        'type' => 'int',
        'unsigned' => TRUE,
        'not null' => TRUE,
        'default' => 0
      ),
      'id_simulacion' => array(
        'description' => 'ID de la simulacion que se hizo en la
partida',
        'type' => 'int',
        'unsigned' => TRUE,
        'not null' => TRUE,
        'default' => 1
      ),
      'consumo_medio' => array(
        'description' => 'Consumo medio del vehículo durante la
partida',
        'type' => 'float',
        'precision' => 12,
        'scale' => 6,
        'not null' => TRUE,
        'default' => 0
      ),
      'consumo_total' => array(
        'description' => 'Consumo total del vehículo al finalizar la
partida',
        'type' => 'float',
        'precision' => 12,
        'scale' => 6,
        'not null' => TRUE,
        'default' => 0
      ),
      'tiempo_total' => array(
        'description' => 'Duración total de la partida',
        'type' => 'float',
        'precision' => 12,
        'scale' => 6,
        'not null' => TRUE,
        'default' => 0
      ),
    ),
    'primary key' => array('id_partida'),
  );

  […]

  return $schema;
}

function rjsimulador_uninstall() {
  variable_del('rjsimulador_grupo_default');
  variable_del('rjsimulador_grupos_edad');
  variable_del('rjsimulador_grupos_experiencia');
  variable_del('rjsimulador_grupos_kilometraje');
}
```

En este fichero se aprecia la implementación de esos tres *hooks*. La implementación de *hook_install* y *hook_uninstall* simplemente se encarga de insertar datos en las tablas y crear variables genéricas o eliminarlas.

La implementación del *hook_schema* es la encargada de crear las tablas en la base de datos que utilizará nuestro módulo para el almacenamiento de datos.

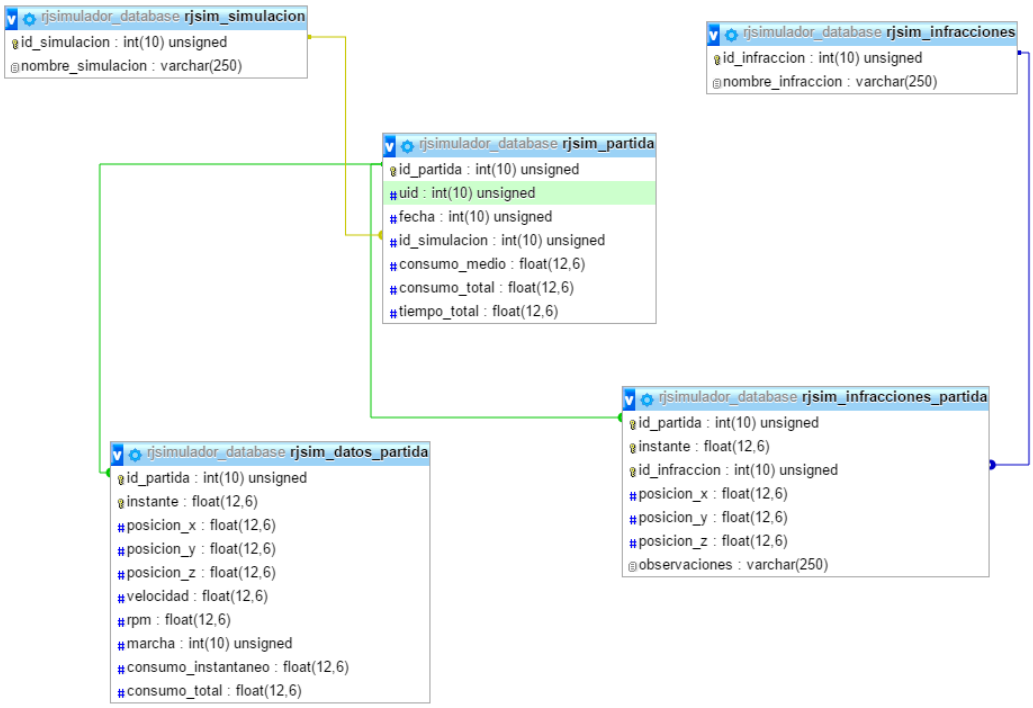


Figura 5.7 – Estructura de las tablas del módulo

En la Figura 5.7 se muestra la estructura de tablas creadas por el módulo. En la función *rjsimulador_schema()* se muestra como es el array que generaría la tabla central de la imagen, la rjsim_partida. Hay que establecer los campos (*fields*) y cada clave del array se corresponderá con los nombres de las columnas de las tablas. Luego basta con seguir la estructura de datos necesaria para cada tipo de campo. Es posible establecer atributos en función del tipo de campo como "not null" o "default". Para más información de los campos soportados y las funcionalidades, se puede acceder a la *Schema* API de Drupal (Drupal, 2015c).

Existen otros ficheros con la terminación .inc que contienen funciones *callback* que permiten generar las páginas de Drupal. Estas funciones actúan como controladores en nuestro módulo. Cuando se accede a un enlace se realiza una de estas llamadas y a partir de ahí podemos decidir cómo gestionar y procesar la salida.

Es inviable ver todos y cada uno de ellos, aun así más adelante se mencionarán algunas de estas funciones cuando se explique cómo se han desarrollado otras partes del módulo.

5.5. Programación orientada a objetos (OOP) y estructura de clases.

Vamos a presentar la estructura de las clases creadas para el módulo.

En la figura 5.8 se puede ver cómo están estructuradas las clases en el módulo **rjsimulador**:

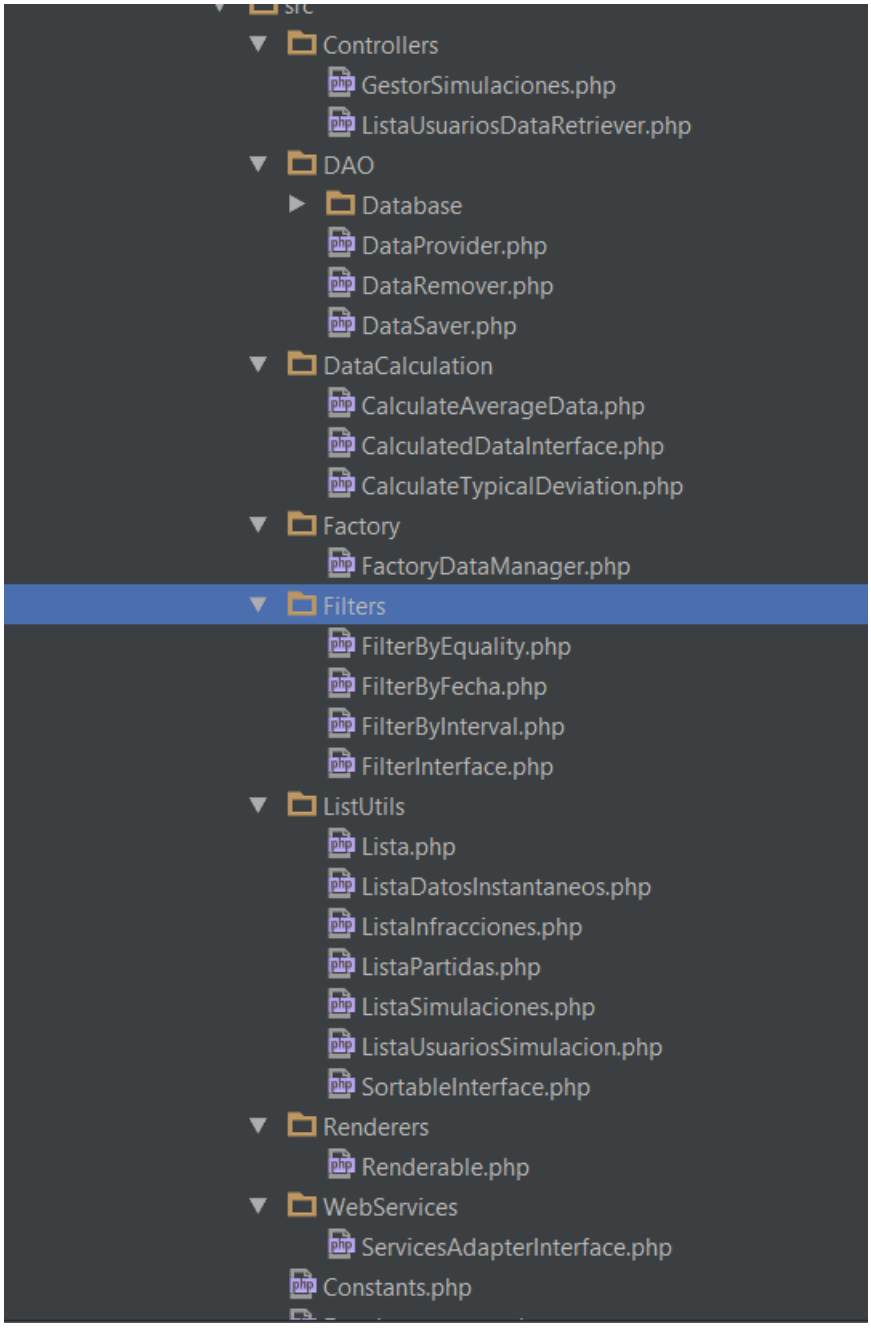


Figura 5.8 – Estructura de clases del módulo

Se ha decidido seguir la convención PSR4 (PHP FIG, 2015b) que indica cómo deben estructurarse los ficheros en una aplicación PHP para que un *autoloader* genérico pueda cargar las clases cuando estas se quieran instanciar y no estén en memoria (recordemos que PHP es un lenguaje interpretado y no compilado).

Para ello se va a presentar un sencillo diagrama de clases (Figura 5.9) que representa la herencia entre las clases e interfaces que hemos creado en nuestro módulo.

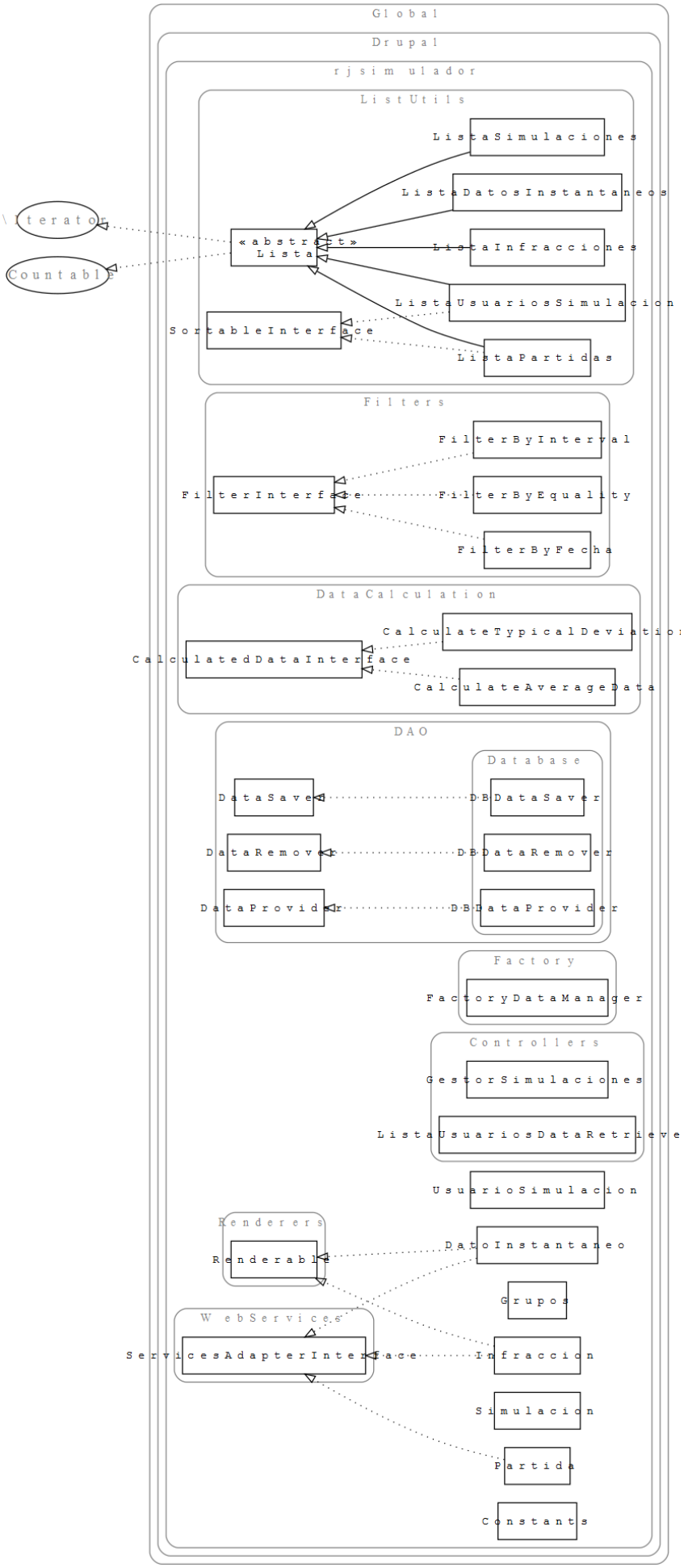


Figura 5.9 – Diagrama de clases

Este diagrama presenta tanto la herencia entre clases como la implementación de interfaces. Los recuadros que rodean las clases representan los *namespaces*, que son agrupaciones virtuales del código de los ficheros PHP. Los *namespaces* se implementaron con la versión 5.3 de PHP y surgen como método de organización de código (Lockhart, 2015).

En este módulo se han definido los siguientes *namespaces*:

- **ListUtils.**

Agrupa funciones que sirven como envoltorio de listas de objetos. Hacen la función de un ArrayList en Java.

- **Filters**.

Clases que implementan lógica de filtrado. Mediante el uso del Patrón Estrategia (Gamma et al., 1994), que se explicará con detalle más adelante en este apartado, derivamos la complejidad del filtrado de una lista de objetos a una clase externa.

- **DataCalculation**.

Igual que con los filtros, en este *namespace* existen clase para hacer cálculos de datos sobre una lista. Al usar el Patrón Estrategia estamos desacoplando la lógica del cálculo del listado de objetos obre el que calcular.

- **DAO**.

Clases que implementan el patrón DAO (*Data Access Object*). Este patrón permite crear una abstracción entre la persistencia de datos y la lógica de la aplicación (Gamma et al., 1994). En el *namespace* DAO están las interfaces que establecen los métodos que luego tendrán que ser implementados por otras clases. Dentro del *namespace* DAO hay un *subspace* Database donde están las clases que implementan esos métodos de forma orientada hacia la persistencia en una base de datos.

- **Controllers**.

Contiene clases que gestionan la comunicación entre las demás.

- **Renderers**.

Contiene las clases (la clase) relativas a como renderizar un objeto como HTML.

- **Factory**.

Contiene las clases usadas para implementar el Patrón Factoría que se explicará con detalle más adelante en este mismo apartado.

- **WebServices**.

Contiene interfaces que establecen unos métodos que deben implementar los objetos para poder usarlos como recursos de un servicio REST.

Se van a detallar algunas de las clases creadas en el módulo, pero sobretodo se van a exponer las razones por las que se ha optado recurrir a esta estructura para llevar a cabo lo que se pretendía. La referencia a las clases se va a realizar de forma relativa a su *namespace* de la forma \Namespace\Subspace\Clase. Para exponer la forma de desarrollo del módulo se va a seguir una estructura **Objetivo-Implementación-Razón**. De esta forma se pretende tener una visión general de cuales eran algunos de los objetivos a conseguir con el módulo y como se han intentado solucionar de la mejor forma posible.

Objetivo: Tener listados de objetos filtrables.

Algunas clases creadas para lograr este objetivo.

- **\ListUtils\Lista**

Esta es una clase abstracta de la que heredaran otras listas de objetos. Es un envoltorio (*wrapper*) alrededor de un array de objetos e implementa las interfaces *Iterable* y *Countable* que permiten recorrer la lista para acceder a los objetos que contiene y contar el número de objetos de la lista respectivamente.

- **\Filters\FilterableInterface**.

Esta interfaz establece un solo método *filter($item)* que recibe un ítem y devuelve TRUE o FALSE si el ítem cumple las condiciones del filtro.

- **\Filters\FilterByFecha**

Esta clase recibe dos fechas y comprueba si el atributo fecha del ítem recibido está entre esos valores.

Razones: En un principio el módulo no estaba definido completamente. Se deseaba poder filtrar las listas de partidas, infracciones, etc, pero aún no se sabía cómo o si iba a ser necesario extender las formas de filtrado. Para solucionar este problema se optó por implementar un patrón de diseño denominado Patrón Estrategia o *Strategy Pattern* (Gamma et al., 1994)

En este patrón, los algoritmos se extraen de las clases complejas de modo que puedan ser reemplazados fácilmente. Por ejemplo, el Patrón Estrategia es una opción si se desea cambiar como se clasifican páginas en un motor de búsqueda. Pensemos en un motor de búsqueda en varias partes - una que itera a través de las páginas, otra que clasifica cada página, y otra que ordena los resultados en función de la clasificación. En un ejemplo complejo, todas esas partes estarían en la misma clase. Utilizando el Patrón Estrategia, se toma la parte de clasificación y se pone en otra clase para que se pueda cambiar como se clasifican las páginas sin afectar al resto del motor.

El patrón de estrategia es ideal para sistemas de gestión de datos complejos o sistemas de procesamiento de datos que necesitan una enorme flexibilidad en cómo se filtran, buscan o clasifican los datos.

En nuestro caso se abstraen los algoritmos de filtrado de las listas de objetos en clases que implementan la interfaz *FiltrableInterface*. Si por ejemplo queremos filtrar nuestra ListaPartidas por la fecha en la que se creó, basta con pasar a nuestro método *filterBy*() de la lista un objeto que implemente la interfaz anterior. De esta manera cualquier objeto que implemente (correctamente) la interfaz nos permitirá filtrar la lista. La complejidad del filtrado de elementos queda delegada a la clase que filtra, mientras que la clase Lista solo sabe que si le pasa un objeto que implemente la interfaz *FilterableInterface* esta le va a decir para cada el elemento si este pasa el filtro o no. Como lo haga le es indiferente.

Es cierto que los filtrados realizados en este módulo no son especialmente complejos. Sin embargo, hacerlo de esta manera permitió modificar las funcionalidades de filtrado para hacerlo por grupos de usuarios de forma fácil, pues toda la lógica del filtrado estaba separada. Sólo hubo que modificar las clases que se encargaban de filtrar (o crear nuevas) y el resto de la aplicación seguía funcionando sin problemas.

Objetivo: Realizar cálculos sobre los datos

Las clases bajo el *namespace DataCalculation* se encargan de realizar cálculos sobre listas de objetos. En este caso al igual que en el anterior se decide usar el Patrón

Estrategia para separar el algoritmo de cálculo (medias, desviaciones típicas) del listado que contiene los datos.

Objetivo: Abstraer la persistencia de datos de la lógica de la aplicación.

Clases creadas para este fin:

- \Factory\FactoryDataManager.

Clase que instancia la clase proveedora de datos adecuada.

- \DAO\DataProvider, DataRemover, DataSaver.

Son interfaces que establecen los métodos que permiten recuperar los datos desde allá donde persistan y devolverlos adecuados para su uso en la aplicación.

- \DAO\Database\DBDataProvider, DBDataRemover, DBDataSaver.

Clase que implementa las interfaces anteriores y que trabajan con bases de dato como persistencia.

Razones: Drupal trabaja solo con bases de datos y ya proporciona una interfaz para abstraer el código del tipo de base de datos usada (soporta varios tipos de bases de datos SQL como MySQL o MariaDB). Sin embargo, ¿qué ocurriría si se necesitara recuperar los tipos de infracciones de un servicio web en lugar de recuperarlos de una tabla de la base de datos? Habría que buscar cada zona del código donde se hubieran recuperado estos datos y modificarlos. ¿Y si se quisieran recuperar todos los datos de un *web service*? Posiblemente habría que buscar y cambiar una enorme cantidad de código.

Para solucionar este problema y poder enfrentarnos a posibles cambios futuros se ha adoptado el Patrón Objeto de Acceso a Datos o *DAO Pattern* (Gamma et al., 1994).

Este patrón pretende abstraer la capa de persistencia de datos de la lógica del software para lo que establece unas interfaces que proporcionan los métodos generales de creación, actualización, recuperación y eliminación (comúnmente llamado CRUD). Esto por sí mismo puede no parecer muy útil pues se requiere de una implementación de las interfaces y si tenemos que instanciar esas clases estamos dependiendo de ellas directamente. Para ello se recurre al Patrón Factoría o *Factory Pattern* (Gamma et al. 1994) que nos permite delegar una clase externa la instanciación de otras clases según un contexto.

En nuestro caso si queremos recupera un listado con todos los usuarios del simulador, recurrimos al método FactoryDataManager::createDataProvider($config) que nos devuelve una instancia de una clase que implemente la interfaz *DataProvider* y lo que se devuelve es una referencia a la interfaz, no a la clase. Esto se denomina Principio de Inversión de Dependencia (*Dependency Inversion Principle*) y establece que:

A. Las clases de alto nivel no deberían depender de las clases de bajo nivel. Ambas deberían depender de las abstracciones.

B. Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.

La idea es evitar siempre que sea posible que clases instancien otras clases más específicas directamente, forzando a la lógica a permanecer estática. En este caso estamos evitando saber si la recuperación de datos se hace de una base de datos o de un servidor o de un fichero. Cedemos la instanciación de la clase concreta a una factoría a la que solo le pasamos un variable de configuración. Esta variable podría ser genérica (en nuestro caso siempre es ‘database’), contextual o configurable desde el *Backend*.

Como hemos mencionado, en nuestro caso es una situación de previsión a futuro. Sin embargo, las buenas prácticas siempre benefician a largo plazo (Martin, 2009).

Objetivo: Renderizar ciertos elementos como contenido HTML.

Clases creadas para este fin:

- Renderers\Renderable.

Es una interfaz que fuerza a que un objeto devuelva un array renderizable de Drupal.

- \DatoInstantaneo, \Infraccion.

Como se desea que estos objetos se puedan devolver como un elemento HTML, se ha implementado en ellos la interfaz anterior.

Razones: La idea era poder tener una serie de objetos que deberían poder pintarse sobre una imagen (un mapa de la simulación). Inicialmente no se habían establecido todos los objetos que se iban a utilizar y por ello se decidió crear una interfaz que separara la implementación de las clases de la lógica. Cuando pintamos los elementos en el mapa creamos un array de arrays de renderizado. Esto arrays los recuperamos invocando el método *renderableArray()* de la interfaz *Renderable*. El tipo de objeto que pudiera haber en ese array es indiferente, basta con saber que los objetos implementan esta interfaz para saber que tienen que tener una implementación de este método.

Hasta el momento sólo dos clases se pintan en el mapa, pero si se deseara, por ejemplo, tener otros datos y para ello se creara una clase, se podrían renderizar estos datos como HTML para poder ser incluidos en el mapa, bastando con implementar la interfaz *Renderable*.

5.6. Diseño general del módulo.

En esta sección se intentarán dar unas ideas de lo que se pretende conseguir con el módulo una vez se han obtenido lo datos. Hemos presentado de forma genérica varios de los *hooks* implementados para integrar el módulo en Drupal. También hemos visto que para crear una partida hay que realizar una petición HTTP al recurso de nuestro *endpoint* con datos en el cuerpo de la petición con un formato JSON determinado.

Ahora vamos a ver cuál el flujo genérico de operaciones que hemos decidido para nuestro módulo.

Nuestro módulo pretende servir los datos de dos formas: una orientada a los usuarios y otra orientada a los administradores.

Los usuarios podrán almacenar partidas desde el simulador mediante un servicio REST, ver datos relacionados con sus partidas como medias, gráficas de infracciones cometidas, detalles específicos de una partida en concreto, etc.

Los administradores tendrán acceso a los mismos datos que un usuario, pero además podrán realizar cálculo filtrados por grupos. En concreto podrán agrupar y comparar datos de medias y desviaciones típicas de Velocidades, RPM o Consumo agrupadas por simulación y grupos de usuarios. Cuando un usuario se da de alta en la aplicación debe dar unos datos que permiten crear estos grupos como son: edad, kilometraje medio anual, años de carnet o si es un usuario habitual de videojuegos.

Utilizando estos datos se pueden calcular los datos en tiempo real y se muestran en gráficas.

Operaciones relacionadas con los usuarios.

Se va a presentar en primer lugar como funciona el servicio web para crear partidas de forma remota.

- **El fichero rjsimulador_partida.resource.inc**

Este fichero contiene la definición de nuestro recurso REST y las funciones necesarias para el almacenamiento de datos.

Vamos a ver cómo es una definición de un recurso REST usando el módulo *Services* de Drupal:

```
      function partida_resource_definition() {
  $resources = array(
    'rjsimulador_partida' => array(
      'operations' => array(
        'create' => array(
          'help' => 'Crea una nueva partida junto con sus datos e
infracciones',
          'file' => array(
            'type' => 'inc',
            'module' => 'rjsimulador',
            'name' => 'resources/rjsimulador_partida.resource'
          ),
          'callback' => '_rjsimulador_partida_add',
          'access callback' => 'user_access',
          'access arguments' => array('crear partidas'),
          'access arguments append' => FALSE,
          'args' => array(
            array(
              'name' => 'partida',
              'type' => 'struct',
              'description' => 'El objeto partida',
              'source' => 'data',
              'optional' => FALSE,
            ),
          ),
        ),
        'retrieve' => array(
          'help' => 'Recupera una partida',
          'file' => array(
            'type' => 'inc',
            'module' => 'rjsimulador',
            'name' => 'resources/rjsimulador_partida.resource'
          ),
          'callback' => '_rjsimulador_partida_retrieve',
          'access callback' => 'user access',
          'access arguments' => array('crear partidas'),
          'access arguments append' => FALSE,
          'args' => array(
            array(
              'name' => 'id_partida',
              'type' => 'int',
              'description' => 'El id de la partida a recuperar',
              'source' => array('path' => 0),
              'optional' => FALSE,
            ),
          ),
```

```
        ),
      ),
    ),
  );
  return $resources;
}

function _rjsimulador_partida_add($partida) {
  $provider = FactoryDataManager::createDataProvider();
  $usuario = $provider->loadSimulatorUser();

  try {
    $tiposSimulaciones = Constants::getTiposSimulacion();
    if (!array_key_exists($partida["id_simulacion"],
$tiposSimulaciones)) {
      throw new InvalidArgumentException("No existe una simulación con
el ID " . $partida["id_simulacion"] . ".");
    }

    $newPartida = new Partida($usuario->getUid(), REQUEST_TIME,
$partida['id_simulacion']);
    $newPartida->setConsumoMedio($partida['consumo_medio']);
    $newPartida->setConsumoTotal($partida['consumo_total']);
    $newPartida->setTiempoTotal($partida['tiempo_total']);

    $tiposInfracciones = Constants::getTiposInfracciones();
    foreach ($partida['infracciones'] as $infraccion) {
      if (!array_key_exists($infraccion["id_infraccion"],
$tiposInfracciones)) {
        throw new InvalidArgumentException("No existe una infracción
con el ID " . $infraccion["id_infraccion"] . ".");
      }

      $record = new Infraccion($infraccion['instante'],
$infraccion['id_infraccion']);
      $record->setPosicionX($infraccion['posicion_x']);
      $record->setPosicionY($infraccion['posicion_y']);
      $record->setPosicionZ($infraccion['posicion_z']);
      $record->setObservaciones($infraccion['observaciones']);
      $newPartida->getListaInfracciones()->add($record);
    }

    foreach ($partida['datos'] as $dato) {
      $record = new DatoInstantaneo($dato['instante'],
$dato['velocidad'], $dato['rpm'], $dato['marcha']);
      $record->setPosicionX($dato['posicion_x']);
      $record->setPosicionY($dato['posicion_y']);
      $record->setPosicionZ($dato['posicion_z']);
      $record->setConsumoInstantaneo($dato['consumo_instantaneo']);
      $record->setConsumoTotal($dato['consumo_total']);
      $newPartida->getListaDatos()->add($record);
    }

    // Creamos una transacción para almacenar la partida; si algo
falla hacemos rollback
    $transaction = db_transaction();
    try {
      $newPartida->save();
      // Hacemos commit deseteando la variable.
      unset($transaction);
    } catch (Exception $e) {
      $transaction->rollback();
      throw $e;
    }
```

```
    return array("message" => "Partida creada correctamente");
  } catch (Exception $e) {
    watchdog_exception('rjsimulador', $e, 'Error inserting new
Partida.');
    return services_error('Error al insertar la partida en la BBDD.',
406, array("error" => "Error al guardar la partida: " . $e-
>getMessage()));
  }
}
```

La función *partida_resource_definition* devuelve un array con los datos necesario para crear un recurso REST. En este caso concreto se están creando dos operaciones: una CREATE que nos permitirá crear una partida a partir de los datos recibidos y una RETRIEVE que nos permitirá recuperar una partida.

Si nos centramos en la creación de partidas vemos que el array tiene una estructura parecida al hook_menu. Tenemos unas *callback* de acceso para controlar quien tiene permisos para acceder al recurso, tenemos cuales son los datos recibidos, etc. En concreto un recurso CREATE se mapea con el verbo HTTP POST y espera recibir un objeto partida en formato JSON. La partida tiene que respetar el formato siguiente:

```
{
  "id_simulacion": 5,
  "consumo_medio": 15.7168,
  "consumo_total": 0.4355,
  "tiempo_total": 254.5819,
  "infracciones": [
    {
      "instante": 12.1645,
      "id_infraccion": 20,
      "posicion_x": 737.7866,
      "posicion_y": 0.1021,
      "posicion_z": 125.8678,
      "observaciones": "Has estado por debajo de la velocidad
mínima de 50 km/h durante 15.6427 segundos. Min. Velocidad alcanzada
en ese tiempo: 0 km/h."
    },
    {
      "instante": 16.3607,
      "id_infraccion": 20,
      "posicion_x": 760.2309,
      "posicion_y": 0.0729,
      "posicion_z": 130.6455,
      "observaciones": "Has estado por debajo de la velocidad
mínima de 50 km/h durante 2.4626 segundos. Min. Velocidad alcanzada en
ese tiempo: 0 km/h."
```

```
          },
          {
            "instante": 18.8158,
            "id_infraccion": 20,
            "posicion_x": 777.8208,
            "posicion_y": 0.102,
            "posicion_z": 127.3162,
            "observaciones": "Has estado por debajo de la velocidad mínima de 50 km/h durante 1.8661 segundos. Min. Velocidad alcanzada en ese tiempo: 0 km/h."
          }
        ],
        "datos": [
          {
            "instante": 0.0329,
            "posicion_x": 734.4,
            "posicion_y": 0.1021,
            "posicion_z": 101.6997,
            "velocidad": 0.0,
            "rpm": 0.0,
            "marcha": 1,
            "consumo_instantaneo": 0.0,
            "consumo_total": 0.0
          },
          {
            "instante": 3.0355,
            "posicion_x": 734.4,
            "posicion_y": 0.1034,
            "posicion_z": 101.6997,
            "velocidad": 0.0001,
            "rpm": 1000.0,
            "marcha": 1,
            "consumo_instantaneo": 0.0,
            "consumo_total": 0.0
          }
        ]
      }
```


Además, la cabecera de la petición debe tener dos valores para que el servicio funcione correctamente. Estos son:

- Content-type = application/json. Indica el tipo de formato que lleva el contenido pasado en el cuerpo de la petición permitiendo al receptor interpretar el contenido.
- X-CSRF-Token = {Token recibido al validarse como usuario}. Cuando un usuario se valida en el servicio a través de los servicios REST, recibe un *token* que permite identificar al usuario durante el tiempo que permanezca activa la sesión en el servidor.

Acabamos de ver cómo definir un recurso REST que nos va a permitir almacenar una partida mediante una petición POST a una URL en concreto, pasando los datos de la partida en un formato concreto

Ahora se va a presentar un diagrama de flujo (Figura 5.10) de cómo responde el servicio REST cuando se recibe una partida en nuestro servicio web de creación cuya dirección es http://{servidor}/api_v1/rjsimulador_partida.

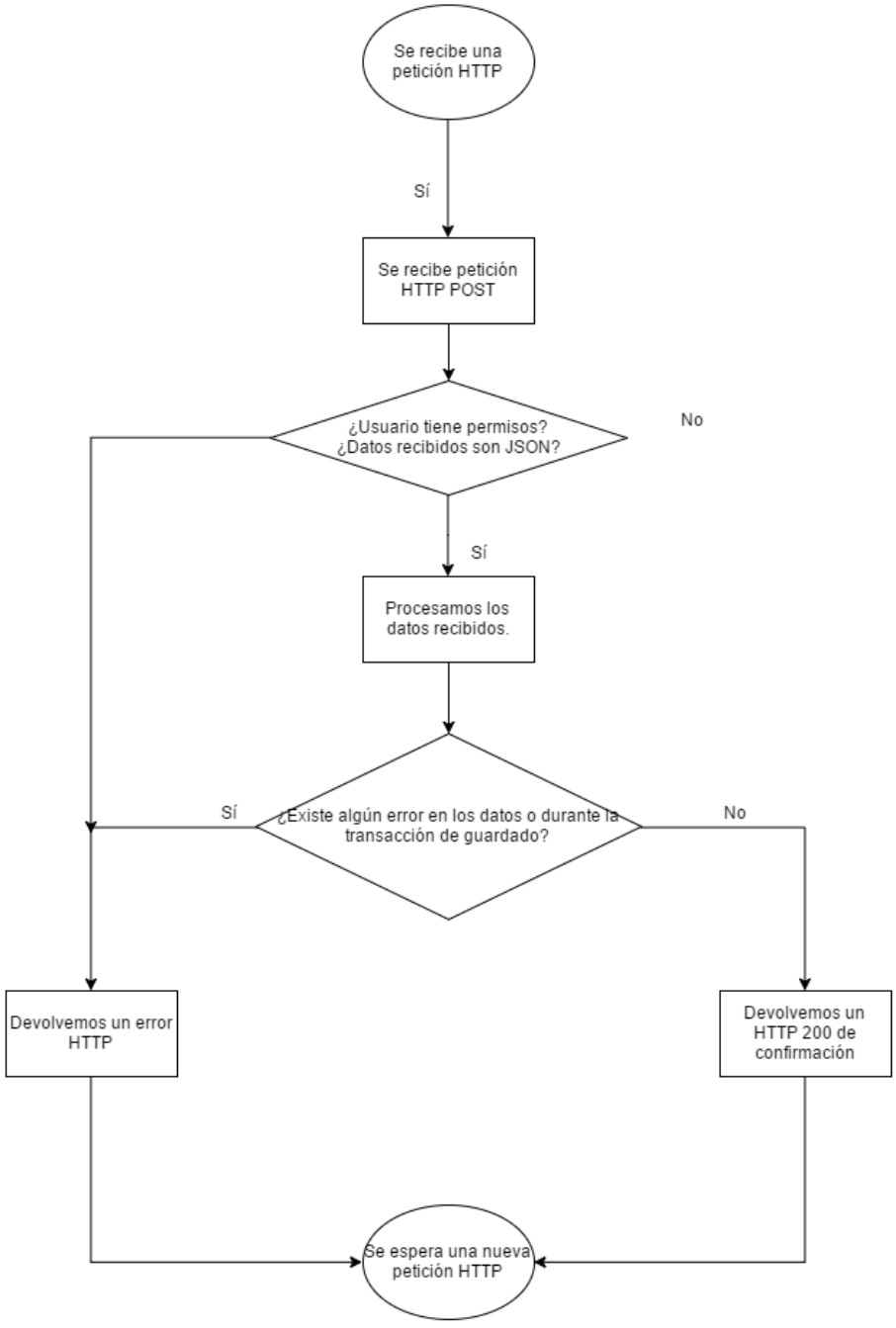


Figura 5.10 – Diagrama de flujo de recepción de una partida.

Una vez hemos activado el servicio REST, este queda a la espera de recibir peticiones POST para almacenar partidas.

El módulo *Services* se encarga de gestionar la entrada de datos al servicio web. En nuestro caso, se encargará de que solo los usuarios con el permiso "crear partidas" puedan usar este recurso. Además, se encargará de convertir los datos de JSON a un array asociativo PHP sin necesidad de que lo hagamos de forma manual en nuestra *callback.*

Una vez se recibe la entrada se comienza el procesamiento de datos. Se hacen comprobaciones como que el ID de simulación exista en la base de datos o que el ID de las infracciones (si existen) también existan. Al final se crea un objeto Partida que contiene los datos validados junto con un listado de Infracciones y DatosInstantaneos.

Una vez hecho esto abrimos una transacción y lanzamos el método de persistencia *save*(). Si durante el proceso de guardado existe algún error realizamos un *rollback* de la transacción y devolvemos un error de que no se ha podido hacer la operación. Cuando se comienza una transacción se crea un punto guardado que mantiene el estado actual de la base de datos. La operación de *rollback* o reversión devuelve la base de datos a ese estado previo sin tener en cuenta los cambios hechos a partir de dicho punto. Haciendo *rollback* en el caso de que exista algún error evitamos el almacenamiento de datos huérfanos. La operación de guardado de la partida debe ser una operación de todo o nada.

Sí el proceso de persistencia de datos va correctamente, se devuelve una respuesta HTTP 200 de que todo ha ido bien.

Una vez un usuario ha almacenado su primera partida en el servidor, cuando se valide en la web le aparecerá un enlace de menú "*My data about Simulations*" donde podrá ver ciertas gráficas, estadísticas, etc, genéricas relacionadas con sus simulaciones.

En la primera pantalla podrá ver gráficas del número de partidas creadas por simulación, número de infracciones cometidas por simulación, etc. En esta primera pantalla además aparece un gráfico que permite ver un porcentaje con el total de infracciones cometidas en el Simulador en general, permitiendo que el usuario pueda ver rápidamente cuales son los tipos de infracciones que más veces comete.

Todos los gráficos se han creado utilizando los elementos proporcionados por la API del módulo *Charts* de Drupal. Este módulo proporciona un "puente" entre el sistema de arrays de renderizado y el formato de la librería *Highcharts* o Google Charts (según configuración del módulo). En este trabajo se ha optado por usar la librería *Highcharts* por tener una interfaz de usuario muy atractiva visualmente.

En la parte inferior de esta pantalla aparece un listado con todas las simulaciones disponibles. Desde esta tabla se puede acceder a otra pantalla en la que el usuario podrá ver datos asociados a las partidas de una simulación en concreto.

En la pantalla de Partidas el usuario podrá ver gráficos de cómo han ido evolucionando su consumo medio o el tiempo medio empleado en finalizar la simulación. Estos datos aparecen contrastados con los datos medios de todos los usuarios, permitiendo hacer una comparativa genérica de cómo está siendo la conducción del usuario frente al resto de usuarios. También es posible ver el número y tipo de infracciones cometidas en las últimas cinco partidas.

En la parte inferior de esta pantalla tenemos un listado de todas las partidas del usuario para esa simulación en concreto. En este listado se puede ordenar por la fecha de creación de la partida y aparecen datos generales de la partida como el consumo

medio o el tiempo empleado en esa simulación. Por último, es posible ver los datos específicos de una partida en concreto.

En la página de la partida aparecen varias tablas de datos y, de forma visual, tenemos una representación de los datos y de las infracciones de la partida seleccionada.

La parte más atractiva es la representación del mapa del escenario con los datos colocados en la posición en la que se crearon. Esto permite tener una visión global de cómo se realizó una simulación, donde se cometieron ciertas infracciones y el camino seguido, entre otras cosas.

Operaciones relacionadas con los administradores.

Cuando un usuario tiene permisos de administración del módulo, puede realizar ciertas operaciones de análisis más complejos que el resto de usuarios.

En primer lugar, es posible acceder a la configuración del módulo desde la pantalla de listado de módulos de la web, en concreto en la URL admin/config/media/rjsimulador. Desde esta pantalla podemos:

- Establecer el grupo de filtrado por defecto de la sección de análisis de datos (ver más adelante).
- Configurar el nombre de las infracciones actuales y añadir nuevas infracciones. Sólo se pueden crear nuevas infracciones o modificar las existentes, pero no eliminarlas. Esto se debe a que no se desea que desaparezca información de partidas anteriores una vez ya se han guardado los datos.
- Configurar el nombre de las simulaciones actuales y añadir nuevas. LAS razones de que no se puedan eliminar son las mismas que para el caso anterior. En este caso podría estudiarse extender la funcionalidad para borrar una Simulación, siempre que se eliminaran todos los datos asociados a la misma como las partidas, y los datos asociados a estas mismas.

La sección principal de la sección administrativa del módulo es "*Analysis of Simulations"* y se puede acceder desde admin/simulaciones_analysis.

Se diseñaron tres secciones que permiten analizar los datos almacenados.

- *Analysis of Simulations by User* (Análisis de Simulaciones por Usuario).

Esta sección contiene un listado de todos los usuarios de la aplicación que han almacenado al menos una partida en el servidor. Básicamente permite a un administrador ver los mismos datos que podría ver un usuario particular.

- *Data analysis* (Análisis de datos)

Permite realizar un análisis de datos de las partidas de los usuarios. Permite calcular medias de datos por usuarios, grupos de usuarios o usuarios particulares.

Análisis general de datos: Permite ver un gráfico con medias de distintos datos por simulación para todos los usuarios.

Análisis de datos por grupos: Permite ver los datos en función de grupos de usuarios. Estos grupos se pueden generar de forma dinámica y se recarga por Ajax. En concreto es posible crear los grupos por edad, experiencia de conducción, kilometraje anual y jugadores habituales de juegos y todas las combinaciones posibles entre estos elementos. Como ya se ha mencionado estos datos son obligatorios cuando el usuario se va a dar de alta en la web.

Análisis de datos por usuario: permite ver los datos de un usuario respecto al resto de usuarios, el resto de usuarios de su mismo grupo de edad, el resto de usuarios

de su misma experiencia o kilometraje medio anual. Además, es posible añadir otros grupos con los que comparar los datos del usuario. Este filtrado se hace respecto al resto de usuarios, sin contar los datos del usuario actual.

- *Infraction analysis* (Análisis de infracciones)

El sistema de análisis de infracciones permite ver el número medio de infracciones cometidas por partida en función del tipo de infracción para una simulación en concreta.

Sigue un esquema simular al análisis de datos, permitiendo realizar comparaciones entre el número de infracciones medias por partida y usuario o por grupos.

Al igual que en el caso anterior los grupos se pueden añadir, modificar o combinar para obtener toda la información deseada.

En la sección de Pruebas se pondrán algunas comparaciones obtenidas mediante este módulo.

5.7. Documentación de la API del módulo.

Como se ha mencionado es imposible explicar todos y cada uno de los ficheros del módulo.

Por ello se optó por intentar documentar del código siguiendo ciertas convenciones de forma que se pudiera generar un documento de la API del mismo. Recordemos que una API (Interfaz de Programación de Aplicaciones) es el conjunto de funciones, métodos o subrutinas que ofrece una librería, componente o módulo para poder ser utilizada por otro software como una capa de abstracción.

Este documento está hecho con fines educativos y por ello se ha optado por sacar los elementos con visibilidad privada. En una API expuesta al público sólo se muestra los elementos públicos, de esta forma no se viola uno de los principios básicos de la programación orientada a objetos como es la encapsulación.

Toda la documentación generada está disponible en la carpeta del proyecto Documentation/RJSimuladorDocs/Doxygen desde donde se puede visualizar de forma interactiva en un navegador (abrir el index.html).

Existe una versión de la documentación generada con un software diferente en DocumentationRJSimualdorDocs/PHPDocumentor.

6. Conclusiones y líneas futuras.

Se ha mencionado recurrentemente durante la memoria que se pretendía desarrollar el esqueleto de un sistema que combinara una aplicación de Simulación de conducción junto con una plataforma web y que permitiera la comunicación de forma cliente – servidor.

Para el desarrollo de este sistema ha sido necesario investigara y adquirir conocimientos de tres áreas claramente diferenciadas: motores gráficos, tecnologías web que permitieran montar una API REST e información relativa a las técnicas de conducción.

Sin embargo, existe un enorme margen de mejora e ideas que se han quedado en el tintero que podrían aplicarse a futuro.

Voy a exponer posibles desarrollos futuros en dos áreas distintas: por un lado, desarrollos orientado a mejorar la plataforma web y por otro a mejorar el simulador.

Mejoras viables para la plataforma web.

1. Gestionar un buffer de entrada para los datos recibidos.

Si se intentara mandar información desde el simulador de gran tamaño, el servidor y la plataforma no están configuradas para ello. Utilizar un buffer de almacenamiento de entrada evitaría que se pudiera perder información durante los envíos de datos desde la aplicación.

2. Desarrollo de una aplicación WebGL que pueda incluirse sobre la plataforma

Unity permite exportar aplicaciones en formato WebGL. Sería muy interesante desarrollar una aplicación que actúe como servidor central y que utilice la API de Network de Unity para mantener las aplicaciones conectadas entre sí y no solo poder recibir datos sino también poder invocar llamadas remotas en los clientes.

La idea en general viene representada en la figura 6.1:

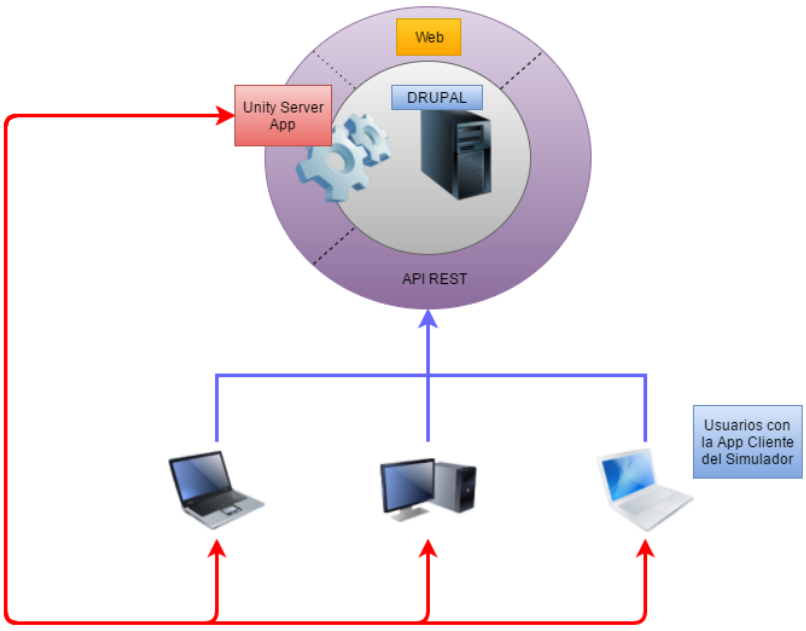


Figura 6.1 – Posible ecosistema futuro.

Actualmente las partes disponibles son el servidor con Drupal y la Web junto con la API REST. Los clientes solo pueden conectarse en una dirección que es la marcad con azul. La propuesta es crear una aplicación Unity que pueda montarse sobre Drupal (exportándola como WebGL) de tal forma que se pueda establecer una comunicación bidireccional permitiendo al servidor invocar métodos en los clientes mediante RPC (*Remote Procedure Call*). Esto permitiría ampliar enormemente las posibilidades del simulador permitiendo a un controlador influir ene l desarrollo de la simulación, tal vez gestionando la aparición de eventos.

3. Definir nuevos recursos web.

Podrían desarrollarse servicios web por ejemplo para recuperar datos de una partida almacenada o recuperar lo datos que se muestran para los usuarios. En concreto el servicio de recuperar partidas se ha implementado en el módulo de Drupal aunque son se usa, pero puede servir de punto de partida.

Esto podría permitir crea una aplicación móvil que permitiría al usuario ver datos de sus partidas desde e cualquier lugar o recibir notificaciones *push* cuando se añadieran nuevas simulaciones dándoles la opción de practicarlas.

Mejoras viables para el simulador:

1. Desarrollar un sistema de tráfico propio.

El *plugin Road & Traffic System*, aunque cumplidor, acaba dando más problemas de los que soluciona. Sería muy interesante desarrollar un sistema de tráfico personalizado, posiblemente basado en el sistema de navegación de Unity para dotar de más realismo al resto de vehículo frente al sistema de viajes entre nodos del *Traffic System*.

2. Crear un escenario entorno infinito.

He pensado varias veces en cómo podría hacer y mi primera idea sería optar por un número pequeño de terrenos que puedan ser conectados entre sí.

Supongamos un caso muy simple en el que tenemos 4 terrenos. Ponemos un terreno central uno por encima y otro por debajo: En el instante en que el jugador pase por un *trigger* que indica le cambio de terreno, se casaca un nuevo terreno de una lista de terrenos desactivados y se coloca por delante del nuevo terreno, mientras que el terreno más alejado se devuelve a esta lista para poder ser usado en el futuro. Esta técnica se denominada *Object Pooling* (Tutorial Unity, 2015) y se utiliza frecuentemente para evitar la instanciación y destrucción de objetos cuando estos se van a utilizar de forma continuada. La figura 6.2 muestra de forma resumida la idea:

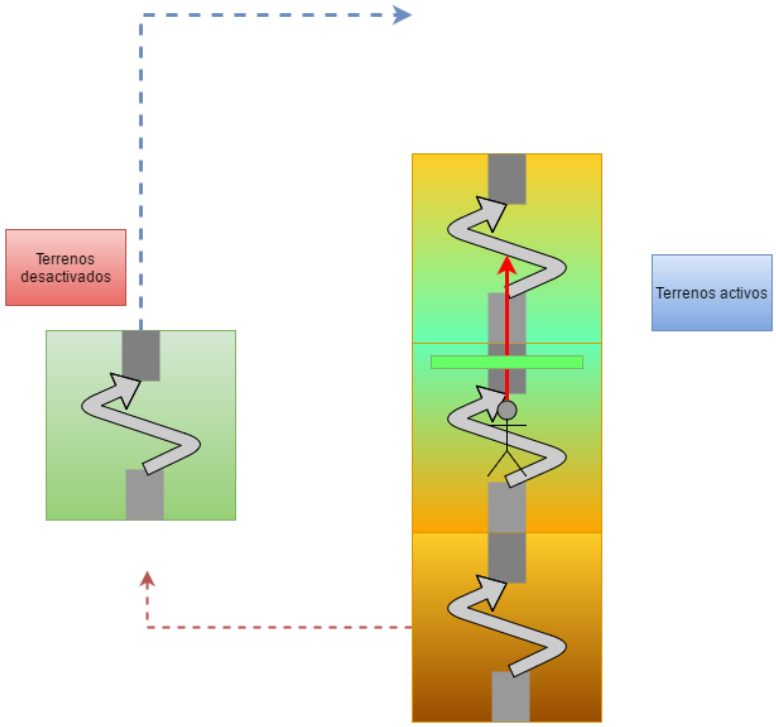


Figura 6.2 – Como crear un escenario infinito.

En la figura cada terreno empieza y acaba con un tramo de carretera que debe ser igual en todos los terrenos para que estos encajen visualmente entre sí.

Mientras cada terreno comience y acabe dela misma manera, la estructura interna de carreteras puede ser tan compleja como se desee.

Esta metodología permitiría crear un escenario infinito con una baja carga computacional y fácilmente extensible. Bastaría con añadir nuevos terrenos al listado de objetos inactivos y recuperarlos de forma aleatoria. Por ejemplo, se podrían crear terrenos con eventos especial que sucederían de forma aleatoria. Por ejemplo, podríamos crear un escenario que sea una carretera que va entre dos montañas y podríamos crear un evento que simule un desprendimiento para comprobar la reacción del conductor. Este evento podría activarse de forma aleatoria lo que impediría que un conductor pierda la concentración por estar moviéndose por los mismos terrenos continuamente. Este sistema de eventos podría combinarse muy bien con la aplicación de servidor mencionada anteriormente. Cederíamos el lanzamiento de estos eventos a una mano humana que podría activarlos cuando lo deseara y ver el comportamiento del conductor.

Pongamos un caso concreto de uso: existe un terreno que tiene el antes mencionado evento de un desprendimiento. En un momento dado el conductor pasa de un terreno a otro y este activa el terreno con el evento. En la aplicación Unity tenemos un mini mapa que va mostrando la posición del conductor (algo viable gracias a la API de *Network* de Unity que permitiría conexiones con baja latencia). Entonces cuando se acerca a la zona del evento, el administrador aumenta el ritmo de recopilación de datos y activa el evento para que este se lance. Esto permite tener más información para estudiar el comportamiento de los conductores frente a ciertos eventos.

7. Presupuesto económico

En esta sección se va a presentar un coste desglosado de los elementos necesarios para el desarrollo de este sistema y también se estimará su viabilidad económica en el caso de que se quisiera usar en un entorno de producción.

Los materiales necesarios para el desarrollo de este proyecto y su coste de venta al público asociados han sido:

- Portátil ASUS GL552VW ……..…… 1290,00 €
- Portátil Toshiba Satellite A500 ……. 750,00 €
- Unity 5, Community Version …… 0,00 €
- Visual Studio, Community Version …..... 0,00 €
- Asset Realistic Car Controller…… 50,00 $ (44,94 €)
- Asset Rewired …… 45,00 $ (40,48 €)
- Asset Modern City Pack …… 16,00 $ (14,39 €)
- Asset West Old Town Vol. 1 ……. 80,00 $ (71,96 €)
- Logitech G27 ……. 249,95 €
- PHPStorm …..... 199,00 € por usuario y año
- CMS Drupal y módulos …..... 0,00 €
- Git …..... 0,00 €

Las tablas de amortización publicadas por Cuéntica (2015) establecen un tiempo de amortización máximo para distintos tipos de inversiones. En nuestro caso concreto vamos a tener en cuenta el tanto por ciento de amortización lineal máximo del 25% para equipos para procesos de información y del 33% para sistemas y programas informáticos.

De esta manera recalculamos el coste de la siguiente manera:

- Portátil ASUS GL552VW ……..…… 322,25 € (25%)
- Portátil Toshiba Satellite A500 ……..……... 187,50 € (25%)
- Unity 5, Community Version …..... 0,00 €
- Visual Studio, Community Version …..... 0,00 €
- Asset Realistic Car Controller……..…..…… 14,83 € (33%)
- Asset Rewired ……...…… 13,36 € (33%)
- Asset Modern City Pack …… 4,75 € (33%)
- Asset West Old Town Vol. 1 ……. 23,75 € (33%)
- Logitech G27 ……. 62,49 € (25%)
- PHPStorm …..... 199,00 € por usuario y año
- CMS Drupal y módulos …..... 0,00 €
- Git …..... 0,00 €

El coste económico total del material prorrateado anual es de 827,93 €.

Se estimaron inicialmente las 150 horas de desarrollo asociadas a un TFG. Como se ha mantenido un desarrollo ágil mediante *Kanban*, a lo largo del ciclo de desarrollo se ha acabado por tener un *Cycle Time* de 3.2 horas y se han realizado 82 tareas, lo que conlleva un total de 262,4 horas. El coste por hora de desarrollo de un programador puede variar bastante dependiendo de su nivel (junior o senior), de las tecnologías que conoce y del campo de experiencia. Si estableceremos un sueldo bruto de un programador junior en 12 €/h tenemos un coste total de desarrollo de 3148,80 €.

El coste final del desarrollo del proyecto se estimaría en 3976,73 €.

Estos precios son precios pretende ser el coste de desarrollo que le llevaría a un autónomo desarrollar este proyecto, sin incluir los costes fijos asociados a cualquier empresa, como costes de local, impuestos de autónomo, luz, mantenimiento de servidores, etc...

Es importante entender la viabilidad de un proyecto antes de afrontarlo económicamente. En nuestro caso, el coste en horas de trabajo está muy por encima de lo estimado, algo que, en una empresa habría resultado en disminución de los beneficios e incluso en pérdidas. Esto demuestra la importancia de la estimación y la necesidad de ser capaz de adaptarse a los cambios. Aunque nuestra estimación por tarea era corta, hemos tenido que realizar 22 tareas más de las esperadas (incluidas tareas de formación e investigación) y, aun así, se han podido finalizar porque estábamos preparados para que hubiera cambios.

8. Bibliografía y recursos.

Bibliografía.

- Administración electrónica del Gobierno de España (2014). "Espacios de Colaboración".
  Disponible en http://administracionelectronica.gob.es/ctt/eco#descripcion.
- Anderson, D. (2010). "Kanban: Successful Evolutionary Change for Your Technology Business", editorial Blue Hole Press.
- Autodesk (2016). "3DS Max Overview".
  Disponible en http://www.autodesk.es/products/3ds-max/overview.
- Beck, K., Beedle, M., Bennekun, A., Cockburn, A., Cunninghanm, W., Fowler, M., …, Thomas, D. (2001). "Agile Manifesto".
  Disponible en http://www.agilemanifesto.org/.
- Blackman, S. (2011). "Beginning 3D Game Development with Unity", editorial Apress.
- Blender (2016). "Blender Project".
  Disponible en http://www.blender.org.
- Bootstrap (2016).
  Disponible en http://getbootstrap.com/.
- Comisariado Europeo del Automóvil (2015). "Las 10 claves de la conducción eficiente".
  Disponible en http://www.cea-online.es/area_tecnica/conduccion_eficiente.asp
- Composer (2016). "Get Composer".
  Disponible en https://getcomposer.org/.
- Concrete5 (2015). "CMS Features".
  Disponible en https://www.concrete5.org/documentation/background/cms_features.
- Creighton, R. H. (2010). "Unity 4.x Game Development by Example Beginners's Guide", editorial Packt.
- Crytek (2015). "Cry Engine Features".
  Disponible en http://cryengine.com/features.
- Cuéntica (2015). "Tabla de amortización de inversiones para sociedades y autónomos en estimación directa normal".
  Disponible en https://ayuda.cuentica.com/tabla-anos-y-porcentajes-de-amortizacion-sociedades-a-partir-de-2015/
- DGT (2015a). "Manual del conductor".

Disponible en http://www.seguridadpublica.es/Autoescuela/manual1.html.

- DGT (2015b). “Reglamento General de Circulación”
  Disponible en http://www.dgt.es/Galerias/seguridad-vial/normativa-legislacion/reglamento-trafico/2015/reglamento_trafico184.pdf
- DGT (2016). “Nuevo mínimo histórico en el número de víctimas mortales por accidente desde 1960”.
  Disponible en http://www.dgt.es/es/prensa/notas-de-prensa/2016/20160104-nuevo-minimo-historico-numero-victimas-mortales-accidente-desde-1960.shtml.
- Doctrine (2015). “Doctrine-Project”.
  Disponible en http://www.doctrine-project.org.
- DriveSim (2016).
  Disponible en http://d rivesimsimulator.com/.
- Drupal (2015a). “Drupal overview”
  Disponible en https://www.drupal.org/drupal-7.0.
- Drupal (2015b). “Creating modules”.
  Disponible en https://www.drupal.org/node/1074360.
- Drupal (2015c). “Render API”.
  Disponible en https://www.drupal.org/node/930760.
- Drupal (2015d). “Schema API”.
  Disponible en https://www.drupal.org/developing/api/schema.
- Epic Games (2016). “Paragon”.
  Disponible en https://www.epicgames.com/paragon/
- Epic Games (2015). “What is UE 4”.
  Disponible en https://www.unrealengine.com/unreal-engine-4.
- Gamma E., Helm, R., Johnson, R., y Vissides, J. (1994). “Design Patterns”.
- González, L. (2014). “Los 4 gráficos que explican definitivamente las rotondas”.
  Disponible en
  http://verne.elpais.com/verne/2014/10/20/articulo/1413804261_000126.html.
- Forward Development (2015). “City Car Driving”.
  Disponible en http:/citycardriving.com.
- iRacing (2015).
  Disponible en http://www.iracing.com/.
- iRacing (2016). “Testimonials”.
  Disponible en http://www.iracing.com/testimonials/.
- Kanban Tool (2015). “Kanban Cycle Time vs Lead Time”.

Disponible en http://kanbantool.com/kanban-cycle-time-vs-lead-time.

- La Voz de Galicia (2016).
  Disponible en http://www.lavozdegalicia.es/noticia/galicia/2016/04/07/circular-glorieta/0003_201604G7P10991.htm?utm_source=facebook&utm_medium=referral&utm_campaign=fbgen
- Liferay (2015a). "Case Studies".
  Disponible en https://www.liferay.com/es/resources
- Liferay (2015b). "Overview".
  Disponible en https://www.liferay.com/es/digital-experience-platform.
- Lockhart, J. (2015). "Modern PHP. New features and Good Practices", editorial O'Reilly.
- Macmillan Dictionary (2015). "*Video arcade*".
  Disponible en http://www.macmillandictionary.com/dictionary/british/video-arcade.
- Manual Autodesk (2016). "NURBS".
  Disponible en https://knowledge.autodesk.com/support/3ds-max/learn-explore/caas/CloudHelp/cloudhelp/2017/ENU/3DSMax/files/GUID-72D000EA-A330-425C-A612-6E5B83842517-htm.html
- Manual Unity (2015a). "Clase *Wheel Collider*".
  Disponible en http://docs.unity3d.com/Manual/class-WheelCollider.html.
- Manual Unity (2015b). "Navigation and pathfinding".
  Disponible en https://docs.unity3d.com/Manual/Navigation.html.
- Manual Unity (2015c). "Physic Material".
  Disponible en http://docs.unity3d.com/es/current/Manual/class-PhysicMaterial.html
- Manual Unity (2015d). "*Rendering Paths*".
  Disponible en http://docs.unity3d.com/Manual/RenderingPaths.html.
- Manual Unity (2015e). "*UI System*".
  Disponible en https://docs.unity3d.com/Manual/UISystem.html.
- Martin, R. C. (2009). "Clean code", editorial Prentice Hall.
- Mercedes-Benz (2015). "The Mercedes-Benz Driving Simulation Center".
  Disponible en https://www.mercedes-benz.com/en/mercedes-benz/innovation/the-mercedes-benz-driving-simulation-center/
- Organización Mundial de la Salud (2015). "Informe sobre la situación mundial de la seguridad vial 2015".

Disponible en http://www.who.int/violence_injury_prevention/road_safety_status/2015/Summary_GSRRS2015_SPA.pdf?ua=1.

- Paniagua, R. (2015). “Entorno virtual para la simulación de conducción basado en Unity”.
- PHP Framework Interoperability Group (2015a). “FAQs”. Disponible en http://www.php-fig.org/faqs/.
- PHP Framework Interoperability Group (2015b). “PSR 4” Disponible en http://www.php-fig.org/psr/psr-4/.
- Polyphony Digital (2015). “Videojuego de simulación Gran Turismo” Disponible en http://www.gran-turismo.com/es/.
- Pozo, J. I . y Gómez, M. A. (2006). “Aprender y enseñar ciencia”.
- Scripting API Unity (2015). “Terrain” Disponible en https://docs.unity3d.com/ScriptReference/Terrain.SetNeighbors.html
- Symfony (2016). “Routing”. Disponible en http://symfony.com/doc/current/book/routing.html.
- Symfony (2015). “Showcase”. Disponible en http://symfony.com/showcase/.
- Totten, C. (2012). “Game Character Creation with Blender and Unity”, editorial John Willey & Sons.
- Trello (2016). “Trello Software” Disponible en https://www.trello.com.
- Tutorial Unity (2015). “Object pooling” Disponible en https://unity3d.com/es/learn/tutorials/topics/scripting/object-pooling.
- Unity (2015). “Unity 5”. Disponible en http://unity3d.com/es/5.
- W3C (2014). “Resource Definition Framework”. Disponible en https://www.w3.org/RDF/
- Wesolowski, J. (2013). “*How to plan optimizations with Unity*”. Disponible en https://software.intel.com/en-us/articles/how-to-plan-optimizations-with-unity.
- XAMPP (2015). “XAMPP Software”. Disponible en https://www.apachefriends.org/es/index.html.

- Youtube (2010). “Gran Turismo 5 vs. Real Life Comparison: Laguna Seca in a Turbo Miata”.
  Disponible en https://www.youtube.com/watch?v=31bH2t2X95g.

Recursos.

Plataforma web en Drupal.

- Aplicación Drupal 7.
  Disponible en https://www.drupal.org/project/drupal.
- Highcharts. Librería JavaScript de gráficos.
  Disponible en http://www.highcharts.com/products/highcharts.
- Módulo Charts.
  Disponible en https://www.drupal.org/project/charts.
- Módulo Libraries API.
  Disponible en https://www.drupal.org/project/libraries.
- Módulo Services.
  Disponible en https://www.drupal.org/project/services.
- Módulo X Autoload.
  Disponible en https://www.drupal.org/project/xautoload.

Simulador Unity.

- BoneCracker Games (2016). “Realistic Car Controller”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/16296.
- Chezer, B. (2015). “Modern City Pack”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/36182.
- Guavaman Enterprises (2016). Rewired.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/21676.
- Jankowski, J, (2014). “Simple Modular Street Kit”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/13811.
- Profi Developers (2012). “West European Old Town Vol. 1”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/2999.
- RKD (2013). “Sky5x One”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/6332.
- UnityHTTP.
  Disponible en https://github.com/andyburke/UnityHTTP.

- VR (2015). “Radio Tower – Low Poly”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/2299.
- Wired Developments (2016). “Road & Traffic System”.
  Disponible en https://www.assetstore.unity3d.com/en/#!/content/21626.