Módulo 1
Introducción

David Vallejo Fernández
david.vallejo@tegnix.com

Carlos González Morcillo
carlos.gonzalez@tegnix.com

Capítulo 1
Introducción

Actualmente, la industria del videojuego goza de una muy buena salud a nivel mundial, rivalizando en presupuesto con las industrias cinematográfica y musical. En este capítulo se discute, desde una perspectiva general, el **desarrollo de videojuegos**, haciendo especial hincapié en su evolución y en los distintos elementos involucrados en este complejo proceso de desarrollo.

En la segunda parte del capítulo se introduce el concepto de **arquitectura del motor**, como eje fundamental para el diseño y desarrollo de videojuegos comerciales.

1.1. El desarrollo de videojuegos

1.1.1. La industria del videojuego. Presente y futuro

Lejos han quedado los días desde el desarrollo de los primeros videojuegos, caracterizados principalmente por su simplicidad y por el hecho de estar desarrollados completamente sobre hardware. Debido a los distintos avances en el campo de la informática, no sólo a nivel de desarrollo software y capacidad hardware sino también en la aplicación

de métodos, técnicas y algoritmos, la industria del videojuego ha evolucionado hasta llegar a cotas inimaginables, tanto a nivel de jugabilidad como de calidad gráfica, tan sólo hace unos años.

La **evolución** de la industria de los videojuegos ha estado ligada a una serie de hitos, determinados particularmente por juegos que han marcado un antes y un después, o por fenómenos sociales que han afectado de manera directa a dicha industria. Juegos como *Doom*, *Quake*, *Final Fantasy*, *Zelda*, *Tekken*, *Gran Turismo*, *Metal Gear*, *The Sims* o *World of Warcraft*, entre otros, han marcado tendencia y han contribuido de manera significativa al desarrollo de videojuegos en distintos géneros.

El videojuego *Pong* se considera como unos de los primeros videojuegos de la historia. Desarrollado por Atari en 1975, el juego iba incluido en la consola *Atari Pong*. Se calcula que se vendieron unas 50.000 unidades.

Por otra parte, y de manera complementaria a la aparición de estas obras de arte, la propia evolución de la informática ha posibilitado la vertiginosa evolución del desarrollo de videojuegos. Algunos **hitos clave** son por ejemplo el uso de la tecnología poligonal en 3D [1] en las consolas de sobremesa, el *boom* de los ordenadores personales como plataforma multi-propósito, la expansión de Internet, los avances en el desarrollo de microprocesadores, el uso de *shaders* programables [10], el desarrollo de motores de juegos o, más recientemente, la eclosión de las redes sociales y el uso masivo de dispositivos móviles.

Por todo ello, los videojuegos se pueden encontrar en ordenadores personales, consolas de juego de sobremesa, consolas portátiles, dispositivos móviles como por ejemplo los *smartphones*, o incluso en las redes sociales como medio de soporte para el entretenimiento de cualquier tipo de usuario. Esta diversidad también está especialmente ligada a distintos tipos o géneros de videojuegos, como se introducirá más adelante en esta misma sección.

La **expansión del videojuego** es tan relevante que actualmente se trata de una industria multimillonaria capaz de rivalizar con las industrias cinematográfica y musical. Un ejemplo representativo es el valor total del mercado del videojuego en Europa, tanto a nivel hardware como software, el cual alcanzó la nada desdeñable cifra de casi 11.000 millones de euros, con países como Reino Unido, Francia o Alemania a la cabeza. En este contexto, España representa el cuarto consumidor a nivel europeo y también ocupa una posición destacada dentro del *ranking* mundial.

A pesar de la vertiginosa evolución de la industria del videojuego, hoy en día existe un gran número de **retos** que el desarrollador de videojuegos ha de afrontar a la hora de producir un videojuego. En realidad, existen retos que perdurarán eternamente y que no están ligados a la propia evolución del hardware que permite la ejecución de los videojuegos. El más evidente de ellos es la necesidad imperiosa de ofrecer una experiencia de entretenimiento al usuario basada en la diversión, ya sea a través de nuevas formas de interacción, como por ejemplo la realidad aumentada o la tecnología de visualización 3D, a través de una mejora evidente en la calidad de los títulos, o mediante innovación en aspectos vinculados a la jugabilidad.

No obstante, actualmente la evolución de los videojuegos está estrechamente ligada a la **evolución del hardware** que permite la ejecución de los mismos. Esta evolución atiende, principalmente, a dos factores: i) la potencia de dicho hardware y ii) las capacidades interactivas del mismo. En el primer caso, una mayor potencia hardware implica que el desarrollador disfrute de mayores posibilidades a la hora de, por ejemplo, mejorar la calidad gráfica de un título o de incrementar la IA (Inteligencia Artificial) de los enemigos. Este factor está vinculado al **multiprocesamiento**. En el segundo caso, una mayor riqueza en términos de interactividad puede contribuir a que el usuario de videojuegos viva una experiencia más inmersiva (por ejemplo, mediante realidad aumentada) o, simplemente, más natural (por ejemplo, mediante la pantalla táctil de un *smartphone*).

Finalmente, resulta especialmente importante destacar la existencia de **motores de juego** (*game engines*), como por ejemplo *Quake*[1] o *Unreal*[2], *middlewares* para el tratamiento de aspectos específicos de un juego, como por ejemplo la biblioteca *Havok*[3] para el tratamiento de la física, o motores de renderizado, como por ejemplo *Ogre 3D* [7]. Este tipo de herramientas, junto con técnicas específicas de desarrollo y optimización, metodologías de desarrollo, o patrones de diseño, entre otros, conforman un aspecto esencial a la hora de desarrollar un videojuego. Al igual que ocurre en otros aspectos relacionados con la Ingeniería del Software, desde un punto de vista general resulta aconsejable el uso de todos estos elementos para agilizar el proceso de desarrollo y reducir errores potenciales. En otras palabras, no es necesario, ni productivo, reinventar la rueda cada vez que se afronta un nuevo proyecto.

[1] http://www.idsoftware.com/games/quake/quake/
[2] http://www.unrealengine.com/
[3] http://www.havok.com

1.1.2. Estructura típica de un equipo de desarrollo

El desarrollo de videojuegos comerciales es un proceso complejo debido a los distintos requisitos que ha de satisfacer y a la integración de distintas disciplinas que intervienen en dicho proceso. Desde un punto de vista general, un videojuego es una **aplicación gráfica en tiempo real** en la que existe una interacción explícita mediante el usuario y el propio videojuego. En este contexto, el concepto de tiempo real se refiere a la necesidad de generar una determinada tasa de *frames* o imágenes por segundo, típicamente 30 ó 60, para que el usuario tenga una sensación continua de realidad. Por otra parte, la interacción se refiere a la forma de comunicación existente entre el usuario y el videojuego. Normalmente, esta interacción se realiza mediante *joysticks* o mandos, pero también es posible llevarla a cabo con otros dispositivos como por ejemplo teclados, ratones, cascos o incluso mediante el propio cuerpo a través de técnicas de visión por computador o de interacción táctil.

> Tiempo real
>
> En el ámbito del desarrollo de videojuegos, el concepto de tiempo real es muy importante para dotar de realismo a los juegos, pero no es tan estricto como el concepto de tiempo real manejado en los sistemas críticos.

A continuación se describe la estructura típica de un equipo de desarrollo atendiendo a los distintos roles que juegan los componentes de dicho equipo [5]. En muchos casos, y en función del número de componentes del equipo, hay personas especializadas en diversas disciplinas de manera simultánea.

Los **ingenieros** son los responsables de diseñar e implementar el software que permite la ejecución del juego, así como las herramientas que dan soporte a dicha ejecución. Normalmente, los ingenieros se suelen clasificar en dos grandes grupos:

- Los **programadores del núcleo** del juego, es decir, las personas responsables de desarrollar tanto el motor de juego como el juego propiamente dicho.
- Los **programadores de herramientas**, es decir, las personas responsables de desarrollar las herramientas que permiten que el resto del equipo de desarrollo pueda trabajar de manera eficiente.

De manera independiente a los dos grupos mencionados, los ingenieros se pueden especializar en una o en varias disciplinas. Por ejemplo, resulta bastante común encontrar perfiles de ingenieros especializados

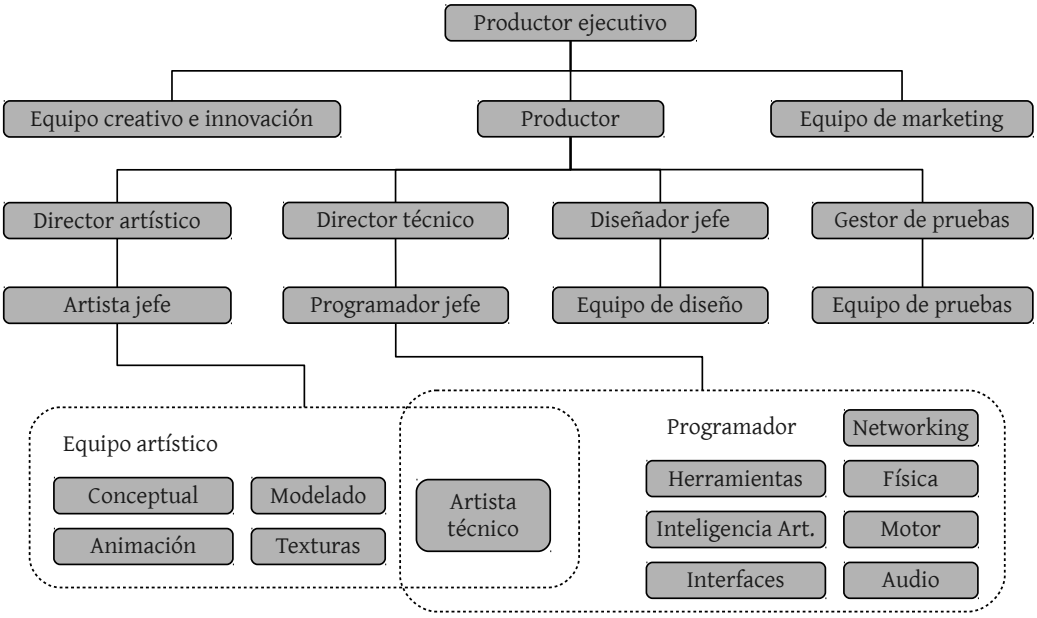

Figura 1.1: Visión conceptual de un equipo de desarrollo de videojuegos, considerando especialmente la parte de programación.

en programación gráfica o en *scripting* e IA. Sin embargo, tal y como se sugirió anteriormente, el concepto de *ingeniero transversal* es bastante común, particularmente en equipos de desarrollo que tienen un número reducido de componentes o con un presupuesto que no les permite la contratación de personas especializadas en una única disciplina.

En el mundo del desarrollo de videojuegos, es bastante probable encontrar ingenieros *senior* responsables de supervisar el desarrollo desde un punto de vista técnico, de manera independiente al diseño y generación de código. No obstante, este tipo de roles suelen estar asociados a la supervisión técnica, la gestión del proyecto e incluso a la toma de decisiones vinculadas a la dirección del proyecto. Así mismo, algunas compañías también pueden tener directores técnicos, responsables de la supervisión de uno o varios proyectos, e incluso un director ejecutivo, encargado de ser el director técnico del estudio completo y de mantener, normalmente, un rol ejecutivo en la compañía o empresa.

Los **artistas** son los responsables de la creación de todo el contenido audio-visual del videojuego, como por ejemplo los escenarios, los personajes, las animaciones de dichos personajes, etc. Al igual que ocurre en el caso de los ingenieros, los artistas también se pueden especializar en diversas cuestiones, destacando las siguientes:

General VS Específico

En función del tamaño de una empresa de desarrollo de videojuegos, el nivel de especialización de sus empleados es mayor o menor. Sin embargo, las ofertas de trabajo suelen incluir diversas disciplinas de trabajo para facilitar su integración.

- Artistas de concepto, responsables de crear bocetos que permitan al resto del equipo hacerse una idea inicial del aspecto final del videojuego. Su trabajo resulta especialmente importante en las primeras fases de un proyecto.
- Modeladores, responsables de generar el contenido 3D del videojuego, como por ejemplo los escenarios o los propios personajes que forman parte del mismo.
- Artistas de texturizado, responsables de crear las texturas o imágenes bidimensionales que formarán parte del contenido visual del juego. Las texturas se aplican sobre la geometría de los modelos con el objetivo de dotarlos de mayor realismo.
- Artistas de iluminación, responsables de gestionar las fuentes de luz del videojuego, así como sus principales propiedades, tanto estáticas como dinámicas.
- Animadores, responsables de dotar de movimientos a los personajes y objetos dinámicos del videojuego. Un ejemplo típico de animación podría ser el movimiento de brazos de un determinado carácter.
- Actores de captura de movimiento, responsables de obtener datos de movimiento reales para que los animadores puedan integrarlos a la hora de animar los personajes.
- Diseñadores de sonido, responsables de integrar los efectos de sonido del videojuego.
- Otros actores, responsables de diversas tareas como por ejemplo los encargados de dotar de voz a los personajes.

Al igual que suele ocurrir con los ingenieros, existe el rol de artista *senior* cuyas responsabilidades también incluyen la supervisión de los numerosos aspectos vinculados al componente artístico.

Los **diseñadores de juego** son los responsables de diseñar el contenido del juego, destacando la evolución del mismo desde el principio

hasta el final, la secuencia de capítulos, las reglas del juego, los objetivos principales y secundarios, etc. Evidentemente, todos los aspectos de diseño están estrechamente ligados al propio género del mismo. Por ejemplo, en un juego de conducción es tarea de los diseñadores definir el comportamiento de los coches adversarios ante, por ejemplo, el adelantamiento de un rival.

Los diseñadores suelen trabajar directamente con los ingenieros para afrontar diversos retos, como por ejemplo el comportamiento de los enemigos en una aventura. De hecho, es bastante común que los propios diseñadores programen, junto con los ingenieros, dichos aspectos haciendo uso de lenguajes de *scripting* de alto nivel, como por ejemplo *LUA*[4] o *Python*[5].

> Scripting e IA
>
> El uso de lenguajes de alto nivel es bastante común en el desarrollo de videojuegos y permite diferenciar claramente la lógica de la aplicación y la propia implementación. Una parte significativa de las desarrolladoras utiliza su propio lenguaje de *scripting*, aunque existen lenguajes ampliamente utilizados, como son *LUA* o *Python*.

Como ocurre con las otras disciplinas previamente comentadas, en algunos estudios los diseñadores de juego también juegan roles de gestión y supervisión técnica.

Finalmente, en el desarrollo de videojuegos también están presentes roles vinculados a la producción, especialmente en estudios de mayor capacidad, asociados a la planificación del proyecto y a la gestión de recursos humanos. En algunas ocasiones, los productores también asumen roles relacionados con el diseño del juego. Así mismo, los responsables de *marketing*, de administración y de soporte juegan un papel relevante. También resulta importante resaltar la figura de publicador como entidad responsable del *marketing* y distribución del videojuego desarrollado por un determinado estudio. Mientras algunos estudios tienen contratos permanentes con un determinado publicador, otros prefieren mantener una relación temporal y asociarse con el publicador que le ofrezca mejores condiciones para gestionar el lanzamiento de un título.

1.1.3. El concepto de juego

Dentro del mundo del entretenimiento electrónico, un **juego** normalmente se suele asociar a la evolución, entendida desde un punto de

[4] http://www.lua.org
[5] http://www.python.org

vista general, de uno o varios personajes principales o entidades que pretenden alcanzar una serie de objetivos en un mundo acotado, los cuales están controlados por el propio usuario. Así, entre estos elementos podemos encontrar desde superhéroes hasta coches de competición pasando por equipos completos de fútbol. El mundo en el que conviven dichos personajes suele estar compuesto, normalmente, por una serie de escenarios virtuales recreados en tres dimensiones y tiene asociado una serie de reglas que determinan la interacción con el mismo.

De este modo, existe una **interacción** explícita entre el jugador o usuario de videojuegos y el propio videojuego, el cual plantea una serie de retos al usuario con el objetivo final de garantizar la diversión y el entretenimiento. Además de ofrecer este componente emocional, los videojuegos también suelen tener un componente cognitivo asociado, obligando a los jugadores a aprender técnicas y a dominar el comportamiento del personaje que manejan para resolver los retos o puzzles que los videojuegos plantean.

Desde una perspectiva más formal, la mayoría de videojuegos suponen un ejemplo representativo de lo que se define como aplicaciones gráficas o **renderizado en tiempo real** [1], las cuales se definen a su vez como la rama más interactiva de la Informática Gráfica. Desde un punto de vista abstracto, una aplicación gráfica en tiempo real se basa en un bucle donde en cada iteración se realizan los siguientes pasos:

- El usuario visualiza una imagen renderizada por la aplicación en la pantalla o dispositivo de visualización.
- El usuario actúa en función de lo que haya visualizado, interactuando directamente con la aplicación, por ejemplo mediante un teclado.
- En función de la acción realizada por el usuario, la aplicación gráfica genera una salida u otra, es decir, existe una retroalimentación que afecta a la propia aplicación.

> **Caída de frames**
>
> Si el núcleo de ejecución de un juego no es capaz de mantener los *fps* a un nivel constante, el juego sufrirá una caída de frames en un momento determinado. Este hecho se denomina comúnmente como *ralentización*.

En el caso de los videojuegos, este ciclo de visualización, actuación y renderizado ha de ejecutarse con una frecuencia lo suficientemente elevada como para que el usuario se sienta inmerso en el videojuego, y no lo perciba simplemente como una sucesión de imágenes estáticas. En

este contexto, el **frame rate** se define como el número de imágenes por segundo, comúnmente *fps*, que la aplicación gráfica es capaz de generar. A mayor *frame rate*, mayor sensación de realismo en el videojuego. Actualmente, una tasa de 30 *fps* se considera más que aceptable para la mayoría de juegos. No obstante, algunos juegos ofrecen tasas que doblan dicha medida.

Generalmente, el desarrollador de videojuegos ha de buscar un compromiso entre los *fps* y el grado de realismo del videojuego. Por ejemplo, el uso de modelos con una alta complejidad computacional, es decir, con un mayor número de polígonos, o la integración de comportamientos inteligentes por parte de los enemigos en un juego, o NPC (Non-Player Character), disminuirá los *fps*.

En otras palabras, los juegos son aplicaciones interactivas que están marcadas por el tiempo, es decir, cada uno de los ciclos de ejecución tiene un *deadline* que ha de cumplirse para no perder realismo.

Aunque el **componente gráfico** representa gran parte de la complejidad computacional de los videojuegos, no es el único. En cada ciclo de ejecución, el videojuego ha de tener en cuenta la evolución del mundo en el que se desarrolla el mismo. Dicha evolución dependerá del estado de dicho mundo en un momento determinado y de cómo las distintas entidades dinámicas interactúan con él. Obviamente, recrear el mundo real con un nivel de exactitud elevado no resulta manejable ni práctico, por lo que normalmente dicho mundo se aproxima y se simplifica, utilizando modelos matemáticos para tratar con su complejidad. En este contexto, destaca por ejemplo la simulación física de los propios elementos que forman parte del mundo.

Por otra parte, un juego también está ligado al comportamiento del personaje principal y del resto de entidades que existen dentro del mundo virtual. En el ámbito académico, estas entidades se suelen definir como **agentes** (*agents*) y se encuadran dentro de la denominada *simulación basada en agentes* [9]. Básicamente, este tipo de aproximaciones tiene como objetivo dotar a los NPC con cierta inteligencia para incrementar el grado de realismo de un juego estableciendo, incluso, mecanismos de cooperación y coordinación entre los mismos. Respecto al personaje principal, un videojuego ha de contemplar las distintas acciones realizadas por el mismo, considerando la posibilidad de decisiones impredecibles a priori y las consecuencias que podrían desencadenar.

En resumen, y desde un punto de vista general, el desarrollo de un juego implica considerar un gran número de factores que, inevitable-

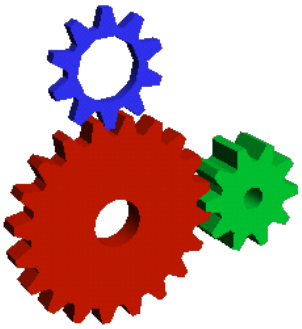

Figura 1.2: El motor de juego representa el núcleo de un videojuego y determina el comportamiento de los distintos módulos que lo componen.

mente, incrementan la complejidad del mismo y, al mismo tiempo, garantizar una tasa de *fps* adecuada para que la inmersión del usuario no se vea afectada.

1.1.4. Motor de juego

Al igual que ocurre en otras disciplinas en el campo de la informática, el desarrollo de videojuegos se ha beneficiado de la aparición de herramientas que facilitan dicho desarrollo, automatizando determinadas tareas y ocultando la complejidad inherente a muchos procesos de bajo nivel. Si, por ejemplo, los SGBD han facilitado enormemente la gestión de persistencia de innumerables aplicaciones informáticas, los motores de juegos hacen la vida más sencilla a los desarrolladores de videojuegos.

Según [5], el término *motor de juego* surgió a mediados de los años 90 con la aparición del famosísimo juego de acción en primera persona *Doom*, desarrollado por la compañía *id Software* bajo la dirección de *John Carmack*[6]. Esta afirmación se sustenta sobre el hecho de que *Doom* fue diseñado con una **arquitectura orientada a la reutilización** mediante una separación adecuada en distintos módulos de los componentes fundamentales, como por ejemplo el sistema de renderizado gráfico, el sistema de detección de colisiones o el sistema de audio, y los elementos más *artísticos*, como por ejemplo los escenarios virtuales o las reglas que gobernaban al propio juego.

Este planteamiento facilitaba enormemente la reutilización de software y el concepto de motor de juego se hizo más popular a medida que otros desarrolladores comenzaron a utilizar diversos módulos o juegos previamente licenciados para generar los suyos propios. En otras palabras, era posible diseñar un juego del mismo tipo sin apenas modificar

[6] http://en.wikipedia.org/wiki/John_D._Carmack

Figura 1.3: John Carmack, uno de los desarrolladores de juegos más importantes, en el *Game Developer Conference* del año 2010.

el núcleo o *motor* del juego, sino que el esfuerzo se podía dirigir directamente a la parte artística y a las reglas del mismo.

Este enfoque ha ido evolucionando y se ha expandido, desde la generación de ***mods*** por desarrolladores independientes o *amateurs* hasta la creación de una gran variedad de herramientas, bibliotecas e incluso lenguajes que facilitan el desarrollo de videojuegos. A día de hoy, una gran parte de compañías de desarrollo de videojuego utilizan motores o herramientas pertenecientes a terceras partes, debido a que les resulta más rentable económicamente y obtienen, generalmente, resultados espectaculares. Por otra parte, esta evolución también ha permitido que los desarrolladores de un juego se planteen licenciar parte de su propio motor de juego, decisión que también forma parte de su política de trabajo.

Obviamente, la separación entre motor de juego y juego nunca es total y, por una circunstancia u otra, siempre existen dependencias directas que no permiten la reusabilidad completa del motor para crear otro juego. La dependencia más evidente es el genero al que está vinculado el motor de juego. Por ejemplo, un motor de juegos diseñado para construir juegos de acción en primera persona, conocidos tradicionalmente como *shooters* o *shoot'em all*, será difícilmente reutilizable para desarrollar un juego de conducción.

Una forma posible para diferenciar un motor de juego y el software que representa a un juego está asociada al concepto de **arquitectura dirigida por datos** (*data-driven architecture*). Básicamente, cuando un juego contiene parte de su lógica o funcionamiento en el propio código (*hard-coded logic*), entonces no resulta práctico reutilizarla para otro juego, ya que implicaría modificar el código fuente sustancialmente. Sin embargo, si dicha lógica o comportamiento no está definido a nivel de código, sino por ejemplo mediante una serie de reglas definidas a través de un lenguaje de *script*, entonces la reutilización sí es posible y, por lo tanto, beneficiosa, ya que optimiza el tiempo de desarrollo.

Los motores de juegos se suelen adaptar para cubrir las necesidades específicas de un título y para obtener un mejor rendimiento.

Como conclusión final, resulta relevante destacar la evolución relativa a la generalidad de los motores de juego, ya que poco a poco están haciendo posible su utilización para diversos tipos de juegos. Sin embargo, el compromiso entre generalidad y optimalidad aún está presente. En otras palabras, a la hora de desarrollar un juego utilizando un determinado motor es bastante común personalizar dicho motor para adaptarlo a las necesidades concretas del juego a desarrollar.

1.1.5. Géneros de juegos

Los motores de juegos suelen estar, generalmente, ligados a un tipo o género particular de juegos. Por ejemplo, un motor de juegos diseñado con la idea de desarrollar juegos de conducción diferirá en gran parte con respecto a un motor orientado a juegos de acción en tercera persona. No obstante, y tal y como se discutirá en la sección 1.2, existen ciertos módulos, sobre todo relativos al procesamiento de más bajo nivel, que son transversales a cualquier tipo de juego, es decir, que se pueden reutilizar en gran medida de manera independiente al género al que pertenezca el motor. Un ejemplo representativo podría ser el módulo de tratamiento de eventos de usuario, es decir, el módulo responsable de recoger y gestionar la interacción del usuario a través de dispositivos como el teclado, el ratón, el joystick o la pantalla táctil. Otros ejemplo podría ser el módulo de tratamiento del audio o el módulo de renderizado de texto.

A continuación, se realizará una descripción de los distintos géneros de juegos más populares atendiendo a las características que diferencian unos de otros en base al motor que les da soporte. Esta descripción resulta útil para que el desarrollador identifique los aspectos críticos de cada juego y utilice las técnicas de desarrollo adecuadas para obtener un buen resultado.

Mercado de *shooters*

Los FPS (First Person Shooter) gozan actualmente de un buen momento y, como consecuencia de ello, el número de títulos disponibles es muy elevado, ofreciendo una gran variedad al usuario final.

Probablemente, el género de juegos más popular es el de los los denominados FPS, abreviado como *shooters*, representado por juegos como *Quake*, *Half-Life*, *Call of Duty* o *Gears of War*, entre muchos otros. En este género, el usuario normalmente controla a un personaje con una vista en primera persona a lo largo de escenarios que tradicionalmente han sido interiores, como los típicos pasillos, pero que han ido evolucionando a escenarios exteriores de gran complejidad.

Figura 1.4: Captura de pantalla del juego *Tremulous*®, licenciado bajo GPL y desarrollado sobre el motor de *Quake III*.

Los FPS representan juegos con un desarrollo complejo, ya que uno de los retos principales que han de afrontar es la inmersión del usuario en un mundo hiperrealista que ofrezca un alto nivel de detalle, al mismo tiempo que se garantice una alta reacción de respuesta a las acciones del usuario. Este género de juegos se centra en la aplicación de las siguientes tecnologías [5]:

- Renderizado eficiente de grandes escenarios virtuales 3D.
- Mecanismo de respuesta eficiente para controlar y apuntar con el personaje.

- Detalle de animación elevado en relación a las armas y los brazos del personaje virtual.
- Uso de una gran variedad de arsenal.
- Sensación de que el personaje *flota* sobre el escenario, debido al movimiento del mismo y al modelo de colisiones.
- NPC con un nivel de IA considerable y dotados de buenas animaciones.
- Inclusión de opciones multijugador a baja escala, típicamente entre 32 y 64 jugadores.

Normalmente, la tecnología de renderizado de los FPS está especialmente optimizada atendiendo, entre otros factores, al tipo de escenario en el que se desarrolla el juego. Por ejemplo, es muy común utilizar estructuras de datos auxiliares para disponer de más información del entorno y, consecuentemente, optimizar el cálculo de diversas tareas. Un ejemplo muy representativo en los escenarios interiores son los árboles BSP (Binary Space Partitioning) (árboles de partición binaria del espacio) [1], que se utilizan para realizar una división del espacio físico en dos partes, de manera recursiva, para optimizar, por ejemplo, aspectos como el cálculo de la posición de un jugador. Otro ejemplo representativo en el caso de los escenarios exteriores es el denominado *occlusion culling* [1], que se utiliza para optimizar el proceso de renderizado descartando aquellos objetos 3D que no se ven desde el punto de vista de la cámara, reduciendo así la carga computacional de dicho proceso.

En el **ámbito comercial**, la familia de motores *Quake*, creados por *Id Software*, se ha utilizado para desarrollar un gran número de juegos, como la saga *Medal of Honor*, e incluso motores de juegos. Hoy es posible descargar el código fuente de *Quake*, *Quake II* y *Quake III* [7] y estudiar su arquitectura para hacerse una idea bastante aproximada de cómo se construyen los motores de juegos actuales.

Otra familia de motores ampliamente conocida es la de *Unreal*, juego desarrollado en 1998 por *Epic Games*. Actualmente, la tecnología *Unreal Engine* se utiliza en multitud de juegos, algunos de ellos tan famosos como *Gears of War*.

Más recientemente, la compañía *Crytek* ha permitido la descarga del CryENGINE 3 SDK (Software Development Kit)[8] para propósitos no comerciales, sino principalmente académicos y con el objetivo de crear una comunidad de desarrollo. Este kit de desarrollo para aplicaciones gráficas en tiempo real es exactamente el mismo que el utilizado por la

[7] http://www.idsoftware.com/business/techdownloads
[8] http://mycryengine.com/

propia compañía para desarrollar juegos comerciales, como por ejemplo *Crysis 2*.

Otro de los géneros más relevantes son los denominados ***juegos en tercera persona***, donde el usuario tiene el control de un personaje cuyas acciones se pueden apreciar por completo desde el punto de vista de la cámara virtual. Aunque existe un gran parecido entre este género y el de los FPS, los juegos en tercera persona hacen especial hincapié en la animación del personaje, destacando sus movimientos y habilidades, además de prestar mucha atención al detalle gráfico de la totalidad de su cuerpo. Ejemplos representativos de este género son *Resident Evil*, *Metal Gear*, *Gears of War* o *Uncharted*, entre otros.

Figura 1.5: Captura de pantalla del juego *Turtlearena*®, licenciado bajo GPL y desarrollado sobre el motor de *Quake III*.

Dentro de este género resulta importante destacar los juegos de plataformas, en los que el personaje principal ha de ir avanzado de un lugar a otro del escenario hasta alcanzar un objetivo. Ejemplos representativos son las sagas de *Super Mario*, *Sonic* o *Donkey Kong*. En el caso particular de los juegos de plataformas, el avatar del personaje tiene normalmente un efecto de *dibujo animado*, es decir, no suele ne-

cesitar un renderizado altamente realista y, por lo tanto, complejo. En cualquier caso, la parte dedicada a la animación del personaje ha de estar especialmente cuidada para incrementar la sensación de realismo a la hora de controlarlo.

> **Super Mario Bros**
>
> El popular juego de Mario, diseñado en 1985 por Shigeru Miyamoto, ha vendido aproximadamente 40 millones de juegos a nivel mundial. Según el libro de los *Record Guinness*, es una de los juegos más vendidos junto a Tetris y a la saga de Pokemon.

En los juegos en tercera persona, los desarrolladores han de prestar especial atención a la aplicación de las siguientes tecnologías [5]:

- Uso de plataformas móviles, equipos de escalado, cuerdas y otros modos de movimiento avanzados.
- Inclusión de puzzles en el desarrollo del juego.
- Uso de cámaras de seguimiento en tercera persona centradas en el personaje y que posibiliten que el propio usuario las maneje a su antojo para facilitar el control del personaje virtual.
- Uso de un complejo sistema de colisiones asociado a la cámara para garantizar que la visión no se vea dificultada por la geometría del entorno o los distintos objetos dinámicos que se mueven por el mismo.

> **Gráficos 3D**
>
> *Virtua Fighter*, lanzado en 1993 por Sega y desarrollado por Yu Suzuki, se considera como el primer juego de lucha arcade en soportar gráficos tridimensionales.

Otro género importante está representado por los **juegos de lucha**, en los que, normalmente, dos jugadores compiten para ganar un determinado número de combatos minando la vida o *stamina* del jugador contrario. Ejemplos representativos de juegos de lucha son *Virtua Fighter*, *Street Fighter*, *Tekken*, o *Soul Calibur*, entre otros. Actualmente, los juegos de lucha se desarrollan normalmente en escenarios tridimensionales donde los luchadores tienen una gran libertad de movimiento. Sin embargo, últimamente se han desarrollado diversos juegos en los que tanto el escenario como los personajes son en 3D, pero donde el

movimiento de los mismos está limitado a dos dimensiones, enfoque comúnmente conocido como juegos de lucha de *scroll lateral*.

Debido a que en los juegos de lucha la acción se centra generalmente en dos personajes, éstos han de tener una gran calidad gráfica y han de contar con una gran variedad de movimientos y animaciones para dotar al juego del mayor realismo posible. Así mismo, el escenario de lucha suele estar bastante acotado y, por lo tanto, es posible simplificar su tratamiento y, en general, no es necesario utilizar técnicas de optimización como las comentadas en el género de los FPS. Por otra parte, el tratamiento de sonido no resulta tan complejo como lo puede ser en otros géneros de acción.

Los juegos del género de la lucha han de prestar atención a la detección y gestión de colisiones entre los propios luchadores, o entre las armas que utilicen, para dar una sensación de mayor realismo. Además, el módulo responsable del tratamiento de la entrada al usuario ha de ser lo suficientemente sofisticado para gestionar de manera adecuada las distintas combinaciones de botones necesarias para realizar complejos movimientos. Por ejemplo, juegos como *Street Fighter IV* incorporan un sistema de *timing* entre los distintos movimientos de un *combo*. El objetivo perseguido consiste en que dominar completamente a un personaje no sea una tarea sencilla y requiera que el usuario de videojuegos dedique tiempo al entrenaiento del mismo.

Los juegos de lucha, en general, han estado ligados a la evolución de técnicas complejas de síntesis de imagen aplicadas sobre los propios personajes con el objetivo de mejorar al máximo su calidad y, de este modo, incrementar su realismo. Un ejemplo representativo es el uso de *shaders* [10] sobre la armadura o la propia piel de los personajes que permitan implementar técnicas como el *bump mapping* [1], planteada para dotar a estos elementos de un aspecto más rugoso.

Otro género representativo en el mundo de los videojuegos es la **conducción**, en el que el usuario controla a un vehículo que normalmente rivaliza con más adversarios virtuales o reales para llegar a la meta en primera posición. En este género se suele distinguir entre *simuladores*, como por ejemplo *Gran Turismo*, y *arcade*, como por ejemplo *Ridge Racer* o *Wipe Out*.

> Simuladores F1
>
> Los simuladores de juegos de conducción no sólo se utilizan para el entretenimiento doméstico sino también para que, por ejemplo, los pilotos de Fórmula-1 conozcan todos los entresijos de los circuitos y puedan conocerlos al detalle antes de embarcarse en los entrenamientos reales.

Figura 1.6: Captura de pantalla del juego de conducción *Tux Racing*, licenciado bajo GPL por Jasmin Patry.

Mientras los simuladores tienen como objetivo principal representar con fidelidad el comportamiento del vehículo y su interacción con el escenario, los juegos arcade se centran más en la jugabilidad para que cualquier tipo de usuario no tenga problemas de conducción.

Los juegos de conducción se caracterizan por la necesidad de dedicar un esfuerzo considerable en alcanzar una calidad gráfica elevada en aquellos elementos cercanos a la cámara, especialmente el propio vehículo. Además, este tipo de juegos, aunque suelen ser muy lineales, mantienen una velocidad de desplazamiento muy elevada, directamente ligada a la del propio vehículo.

Al igual que ocurre en el resto de géneros previamente comentados, existen diversas técnicas que pueden contribuir a mejorar la eficiencia de este tipo de juegos. Por ejemplo, suele ser bastante común utilizar estructuras de datos auxiliares para dividir el escenario en distintos tramos, con el objetivo de optimizar el proceso de renderizado o incluso facilitar el cálculo de rutas óptimas utilizando técnicas de IA [11]. También se suelen usar imágenes para renderizar elementos lejanos, como por ejemplo árboles, vallas publicitarias u otro tipo de elementos.

Del mismo modo, y al igual que ocurre con los juegos en tercera persona, la cámara tiene un papel relevante en el seguimiento del juego. En este contexto, el usuario normalmente tiene la posibilidad de elegir el tipo de cámara más adecuado, como por ejemplo una cámara en primera persona, una en la que se visualicen los controles del propio vehículo o una en tercera persona.

Otro género tradicional son los juegos de **estrategia**, normalmente clasificados en tiempo real o RTS (Real-Time Strategy)) y por turnos (*turn-based strategy*). Ejemplos representativos de este género son *Warcraft*, *Command & Conquer*, *Comandos*, *Age of Empires* o *Starcraft*, entre otros.

Este tipo de juegos se caracterizan por mantener una cámara con una perspectiva isométrica, normalmente fija, de manera que el jugador tiene una visión más o menos completa del escenario, ya sea 2D o 3D. Así mismo, es bastante común encontrar un gran número de unidades

Figura 1.7: Captura de pantalla del juego de estrategia en tiempo real *0 A.D.*, licenciado bajo GPL por *Wildfiregames*.

virtuales desplegadas en el mapa, siendo responsabilidad del jugador su control, desplazamiento y acción.

Teniendo en cuenta las características generales de este género, es posible plantear diversas optimizaciones. Por ejemplo, una de las aproximaciones más comunes en este tipo de juegos consiste en dividir el escenario en una rejilla o *grid*, con el objetivo de facilitar no sólo el emplazamiento de unidades o edificios, sino también la planificación de movimiento de un lugar del mapa a otro. Por otra parte, las unidades se suelen renderizar con una resolución baja, es decir, con un bajo número de polígonos, con el objetivo de posibilitar el despliegue de un gran número de unidades de manera simultánea.

Finalmente, en los últimos años ha aparecido un género de juegos cuya principal característica es la posibilidad de jugar con un gran número de jugadores reales al mismo tiempo, del orden de cientos o incluso miles de jugadores. Los juegos que se encuadran bajo este género se denomina comúnmente MMOG (Massively Multiplayer Online Game). El ejemplo más representativo de este género es el juego *World of Warcarft*. Debido a la necesidad de soportar un gran número de jugadores en línea, los desarrolladores de este tipo de juegos han de realizar un gran esfuerzo en la parte relativa al *networking*, ya que han de proporcionar un servicio de calidad sobre el que construir su modelo de negocio, el cual suele estar basado en suscripciones mensuales o anuales por parte de los usuarios.

Al igual que ocurre en los juegos de estrategia, los MMOG suelen utilizar personajes virtuales en baja resolución para permitir la aparición de un gran número de ellos en pantalla de manera simultánea.

Además de los distintos géneros mencionados en esta sección, existen algunos más como por ejemplo los juegos deportivos, los juegos de rol o RPG (Role-Playing Games) o los juegos de puzzles.

Antes de pasar a la siguiente sección en la que se discutirá la arquitectura general de un motor de juego, resulta interesante destacar la existencia de algunas **herramientas libres** que se pueden utilizar para la construcción de un motor de juegos. Una de las más populares es

OGRE 3D[9]. Básicamente, OGRE es un motor de renderizado 3D bien estructurado y con una curva de aprendizaje adecuada. Aunque OGRE no se puede definir como un motor de juegos completo, sí que proporciona un gran número de módulos que permiten integrar funcionalidades no triviales, como iluminación avanzada o sistemas de animación de caracteres.

1.2. Arquitectura del motor. Visión general

En esta sección se plantea una visión general de la arquitectura de un motor de juegos [5], de manera independiente al género de los mismos, prestando especial importancia a los módulos más relevantes desde el punto de vista del desarrollo de videojuegos.

Como ocurre con la gran mayoría de sistemas software que tienen una complejidad elevada, los motores de juegos se basan en una **arquitectura estructurada en capas**. De este modo, las capas de nivel superior dependen de las capas de nivel inferior, pero no de manera inversa. Este planteamiento permite ir añadiendo capas de manera progresiva y, lo que es más importante, permite modificar determinados aspectos de una capa en concreto sin que el resto de capas inferiores se vean afectadas por dicho cambio.

A continuación, se describen los principales módulos que forman parte de la arquitectura que se expone en la figura 1.8.

1.2.1. Hardware, *drivers* y sistema operativo

La capa relativa al **hardware** está vinculada a la plataforma en la que se ejecutará el motor de juego. Por ejemplo, un tipo de plataforma específica podría ser una consola de juegos de sobremesa. Muchos de los principios de diseño y desarrollo son comunes a cualquier videojuego, de manera independiente a la plataforma de despliegue final. Sin embargo, en la práctica los desarrolladores de videojuegos siempre llevan a cabo optimizaciones en el motor de juegos para mejorar la eficiencia del mismo, considerando aquellas cuestiones que son específicas de una determinada plataforma.

[9] http://www.ogre3d.org/

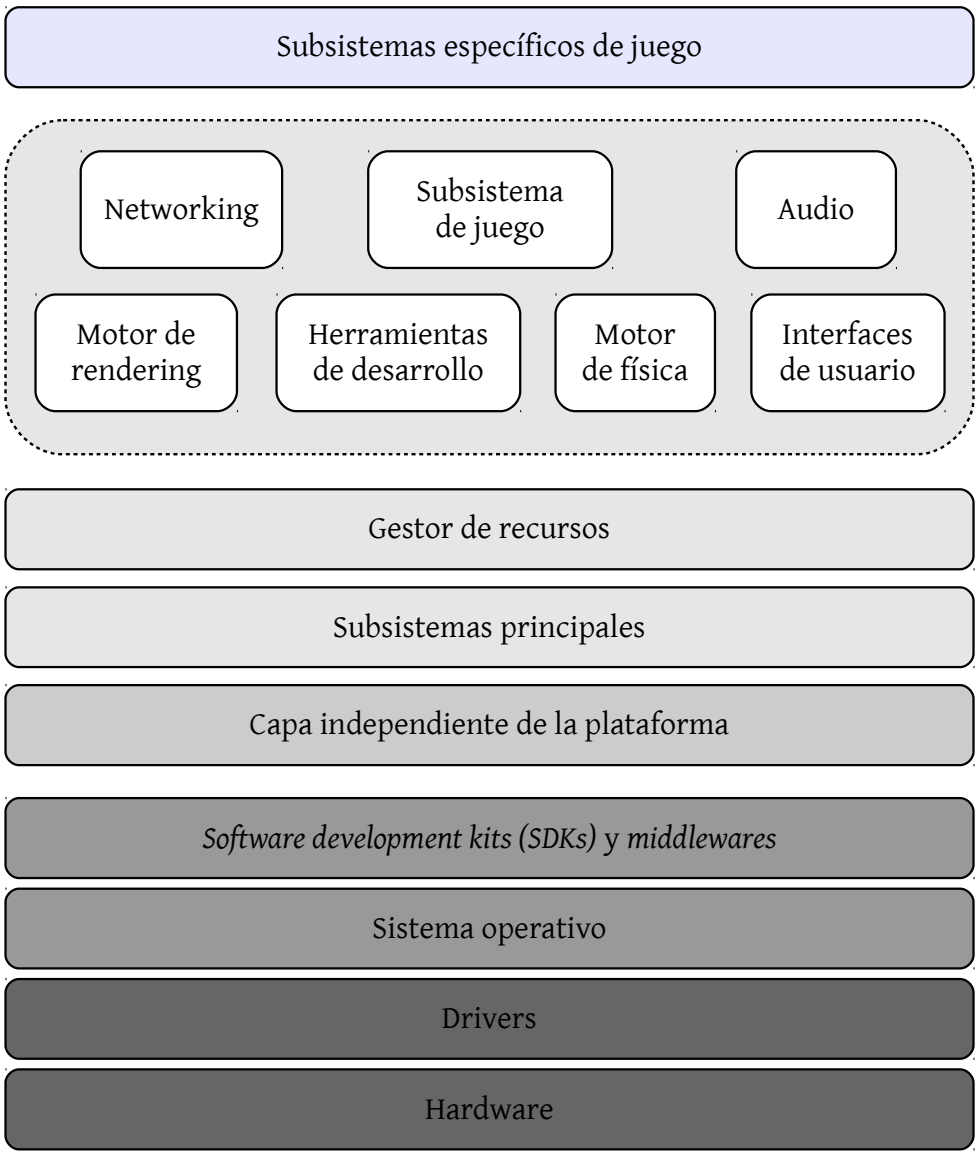

Figura 1.8: Visión conceptual de la arquitectura general de un motor de juegos. Esquema adaptado de la arquitectura propuesta en [5].

La arquitectura *Cell*

En arquitecturas más novedosas, como por ejemplo la arquitectura *Cell* usada en *Playstation 3* y desarrollada por *Sony*, *Toshiba* e IBM, las optimizaciones aplicadas suelen ser más dependientes de la plataforma final.

La capa de ***drivers*** soporta aquellos componentes software de bajo nivel que permiten la correcta gestión de determinados dispositivos, como por ejemplo las tarjetas de aceleración gráfica o las tarjetas de sonido.

La capa del **sistema operativo** representa la capa de comunicación entre los procesos que se ejecutan en el mismo y los recursos hardware asociados a la plataforma en cuestión. Tradicionalmente, en el mundo de los videojuegos los sistemas operativos se compilan con el propio juego para producir un ejecutable. Sin embargo, las consolas de última generación, como por ejemplo *Sony Playstation 3*® o *Microsoft XBox 360*®, incluyen un sistema operativo capaz de controlar ciertos recursos e incluso interrumpir a un juego en ejecución, reduciendo la separación entre consolas de sobremesa y ordenadores personales.

1.2.2. SDKs y *middlewares*

Al igual que ocurre en otros proyectos software, el desarrollo de un motor de juegos se suele apoyar en bibliotecas existentes y SDK para proporcionar una determinada funcionalidad. No obstante, y aunque generalmente este software está bastante optimizado, algunos desarrolladores prefieren personalizarlo para adaptarlo a sus necesidades particulares, especialmente en consolas de sobremesa y portátiles.

API (Application Program Interface)s gráficas

OpenGL y Direct3D son los dos ejemplos más representativos de APIs gráficas que se utilizan en el ámbito comercial. La principal diferencia entre ambas es la estandarización, factor que tiene sus ventajas y desventajas.

Un ejemplo representativo de biblioteca para el manejo de **estructuras de datos** es STL (Standard Template Library) [10]. STL es una biblioteca de plantillas estándar para C++, el cual representa a su vez el lenguaje más extendido actualmente para el desarrollo de videojuegos, debido principalmente a su portabilidad y eficiencia.

[10] http://www.sgi.com/tech/stl/

En el ámbito de los **gráficos 3D**, existe un gran número de bibliotecas de desarrollo que solventan determinados aspectos que son comunes a la mayoría de los juegos, como el renderizado de modelos tridimensionales. Los ejemplos más representativos en este contexto son las APIs gráficas *OpenGL*[11] y *Direct3D*, mantenidas por el grupo *Khronos* y *Microsoft*, respectivamente. Este tipo de bibliotecas tienen como principal objetivo ocultar los diferentes aspectos de las tarjetas gráficas, presentando una interfaz común. Mientras *OpenGL* es multiplataforma, *Direct3D* está totalmente ligado a sistemas *Windows*.

Otro ejemplo representativo de SDKs vinculados al desarrollo de videojuegos son aquellos que dan soporte a la detección y tratamiento de **colisiones** y a la gestión de la **física** de los distintas entidades que forman parte de un videojuego. Por ejemplo, en el ámbito comercial la compañía *Havok*[12] proporciona diversas herramientas, entre las que destaca *Havok Physics*. Dicha herramienta representa la alternativa comercial más utilizada en el ámbito de la detección de colisiones en tiempo real y en las simulaciones físicas. Según sus autores, *Havok Physics* se ha utilizado en el desarrollo de más de 200 títulos comerciales.

Por otra parte, en el campo del *Open Source*, ODE (Open Dynamics Engine) 3D[13] representa una de las alternativas más populares para simular dinámicas de cuerpo rígido [1].

Recientemente, la rama de la **Inteligencia Artificial** en los videojuegos también se ha visto beneficiada con herramientas que posibilitan la integración directa de bloques de bajo nivel para tratar con problemas clásicos como la búsqueda óptima de caminos entre dos puntos o la acción de evitar obstáculos.

1.2.3. Capa independiente de la plataforma

Gran parte de los juegos se desarrollan teniendo en cuenta su potencial lanzamiento en diversas plataformas. Por ejemplo, un título se puede desarrollar para diversas consolas de sobremesa y para PC al mismo tiempo. En este contexto, es bastante común encontrar una capa software que aisle al resto de capas superiores de cualquier aspecto que sea dependiente de la plataforma. Dicha capa se suele denominar *capa independiente de la plataforma*.

Aunque sería bastante lógico suponer que la capa inmediatamente inferior, es decir, la capa de SDKs *y middleware*, ya posibilita la independencia respecto a las plataformas subyacentes debido al uso de módulos estandarizados, como por ejemplo bibliotecas asociadas a C/C++,

[11] http://http://www.opengl.org/
[12] http://www.havok.com
[13] http://www.ode.org

la realidad es que existen diferencias incluso en bibliotecas estandarizadas para distintas plataformas.

Aunque en teoría las herramientas multiplataforma deberían abstraer de los aspectos subyacentes a las mismas, como por ejemplo el sistema operativo, en la práctica suele ser necesario realizar algunos ajustos en función de la plataforma existente en capas de nivel inferior.

Algunos ejemplos representativos de módulos incluidos en esta capa son las bibliotecas de manejo de hijos o los *wrappers* o envolturas sobre alguno de los módulos de la capa superior, como el módulo de detección de colisiones o el responsable de la parte gráfica.

1.2.4. Subsistemas principales

La capa de subsistemas principales está vinculada a todas aquellas utilidades o bibliotecas de utilidades que dan soporte al motor de juegos. Algunas de ellas son específicas del ámbito de los videojuegos pero otras son comunes a cualquier tipo de proyecto software que tenga una complejidad significativa.

A continuación se enumeran algunos de los subsistemas más relevantes:

- **Biblioteca matemática**, responsable de proporcionar al desarrollador diversas utilidades que faciliten el tratamiento de operaciones relativas a vectores, matrices, cuaterniones u operaciones vinculadas a líneas, rayos, esferas y otras figuras geométricas. Las bibliotecas matemáticas son esenciales en el desarrollo de un motor de juegos, ya que éstos tienen una naturaleza inherentemente matemática.
- **Estructuras de datos y algoritmos**, responsable de proporcionar una implementación más personalizada y optimizada de diversas estructuras de datos, como por ejemplo listas enlazadas o árboles binarios, y algoritmos, como por ejemplo búsqueda u ordenación, que la encontrada en bibliotecas como STL. Este subsistema resulta especialmente importante cuando la memoria de la plataforma o plataformas sobre las que se ejecutará el motor está limitada (como suele ocurrir en consolas de sobremesa).
- **Gestión de memoria**, responsable de garantizar la asignación y liberación de memoria de una manera eficiente.

- **Depuración y *logging***, responsable de proporcionar herramientas para facilitar la depuración y el volcado de *logs* para su posterior análisis.

1.2.5. Gestor de recursos

Esta capa es la responsable de proporcionar una interfaz unificada para acceder a las distintas entidades software que conforman el motor de juegos, como por ejemplo la escena o los propios objetos 3D. En este contexto, existen dos aproximaciones principales respecto a dicho acceso: i) plantear el gestor de recursos mediante un enfoque centralizado y consistente o ii) dejar en manos del programador dicha interacción mediante el uso de archivos en disco.

> Ogre 3D
>
> El motor de *rendering* Ogre 3D está escrito en C++ y permite que el desarrollador se abstraiga de un gran número de aspectos relativos al desarrollo de aplicaciones gráficas. Sin embargo, es necesario estudiar su funcionamiento y cómo utilizarlo de manera adecuada.

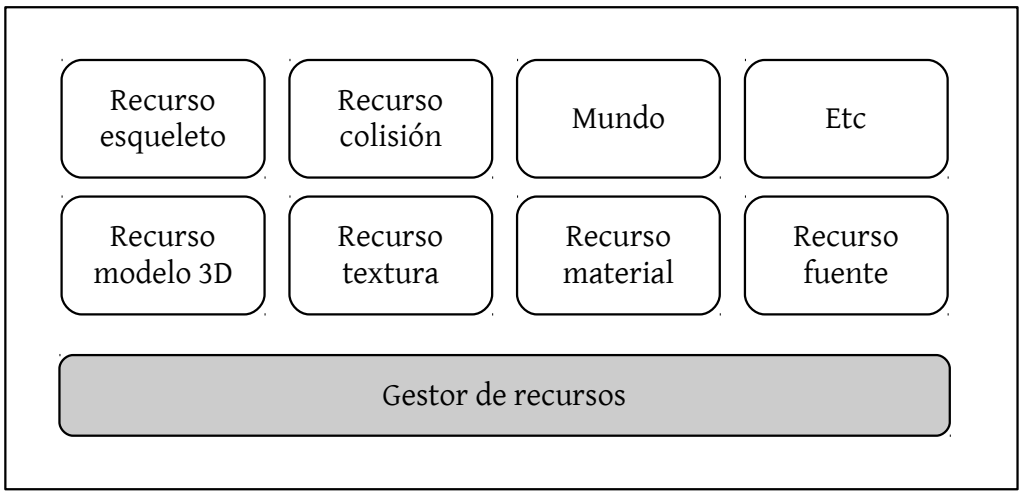

Figura 1.9: Visión conceptual del gestor de recursos y sus entidades asociadas. Esquema adaptado de la arquitectura propuesta en [5].

La figura 1.9 muestra una visión general de un gestor de recursos, representando una interfaz común para la gestión de diversas entidades como por ejemplo el mundo en el que se desarrolla el juego, los objetos 3D, las texturas o los materiales.

En el caso particular de *Ogre 3D* [7], el gestor de recursos está representado por la clase *Ogre::ResourceManager*, tal y como se puede apreciar en la figura 1.10. Dicha clase mantiene diversas especializaciones, las cuales están ligadas a las distintas entidades que a su vez gestionan distintos aspectos en un juego, como por ejemplo las texturas (clase *Ogre::TextureManager*), los modelos 3D (clase *Ogre::MeshManager*) o las fuentes de texto (clase *Ogre::FontManager*). En el caso particular de Ogre 3D, la clase *Ogre::ResourceManager* hereda de dos clases, *ResourceAlloc* y *Ogre::ScriptLoader*, con el objetivo de unificar completamente las diversas gestiones. Por ejemplo, la clase *Ogre::ScriptLoader* posibilita la carga de algunos recursos, como los materiales, mediante *scripts* y, por ello, *Ogre::ResourceManager* hereda de dicha clase.

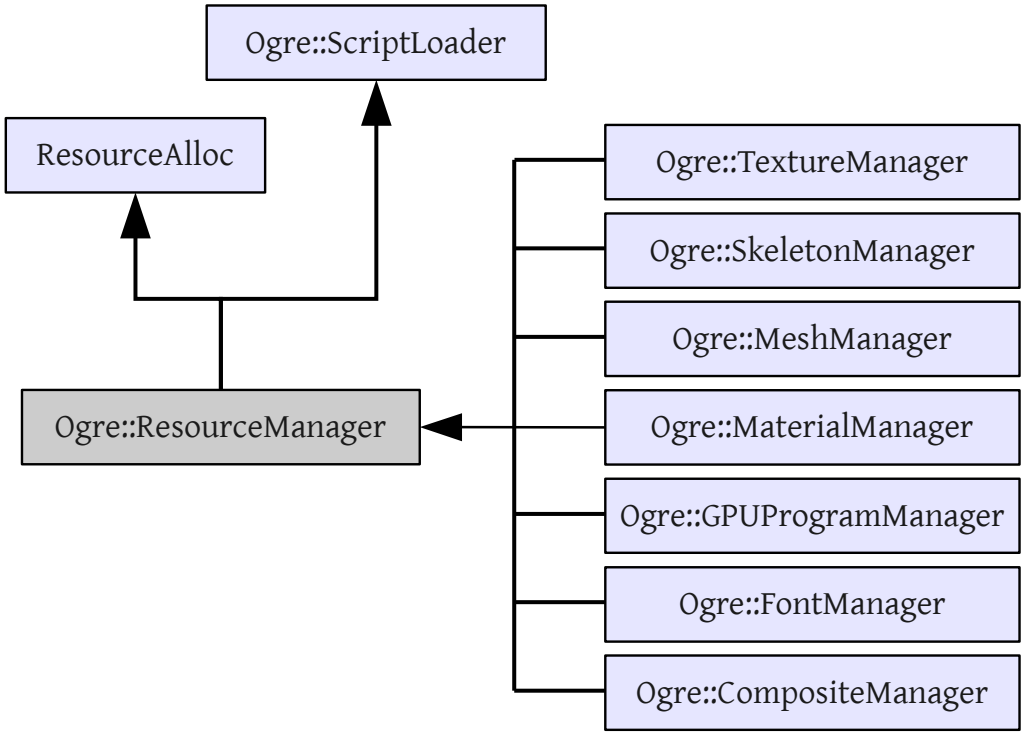

Figura 1.10: Diagrama de clases asociado al gestor de recursos de *Ogre 3D*, representado por la clase *Ogre::ResourceManager*.

1.2.6. Motor de *rendering*

Debido a que el componente gráfico es una parte fundamental de cualquier juego, junto con la necesidad de mejorarlo continuamente, el motor de renderizado es una de las partes más complejas de cualquier motor de juego.

> *Shaders*
>
> Un *shader* se puede definir como un conjunto de instrucciones software que permiten aplicar efectos de renderizado a primitivas geométricas. Al ejecutarse en las unidades de procesamiento gráfico (*Graphic Processing Units - GPUs*), el rendimiento de la aplicación gráfica mejora considerablemente.

Al igual que ocurre con la propia arquitectura de un motor de juegos, el enfoque más utilizado para diseñar el motor de renderizado consiste en utilizar una arquitectura multi-capa, como se puede apreciar en la figura 1.11.

A continuación se describen los principales módulos que forman parte de cada una de las capas de este componente.

La capa de **renderizado de bajo nivel** aglutina las distintas utilidades de renderizado del motor, es decir, la funcionalidad asociada a la representación gráfica de las distintas entidades que participan en un determinado entorno, como por ejemplo cámaras, primitivas de *rendering*, materiales, texturas, etc. El objetivo principal de esta capa reside precisamente en renderizar las distintas primitivas geométricas tan rápido como sea posible, sin tener en cuenta posibles optimizaciones ni considerar, por ejemplo, qué partes de las escenas son visibles desde el punto de vista de la cámara.

> Optimización
>
> Las optimizaciones son esenciales en el desarrollo de aplicaciones gráficas, en general, y de videojuegos, en particular, para mejorar el rendimiento. Los desarrolladores suelen hacer uso de estructuras de datos auxiliares para aprovecharse del mayor conocimiento disponible sobre la propia aplicación.

Esta capa también es responsable de gestionar la interacción con las APIs de programación gráficas, como *OpenGL* o *Direct3D*, simplemente para poder acceder a los distintos dispositivos gráficos que estén disponibles. Típicamente, este módulo se denomina *interfaz de dispositivo gráfico* (*graphics device interface*).

Así mismo, en la capa de renderizado de bajo nivel existen otros componentes encargados de procesar el dibujado de distintas primiti-

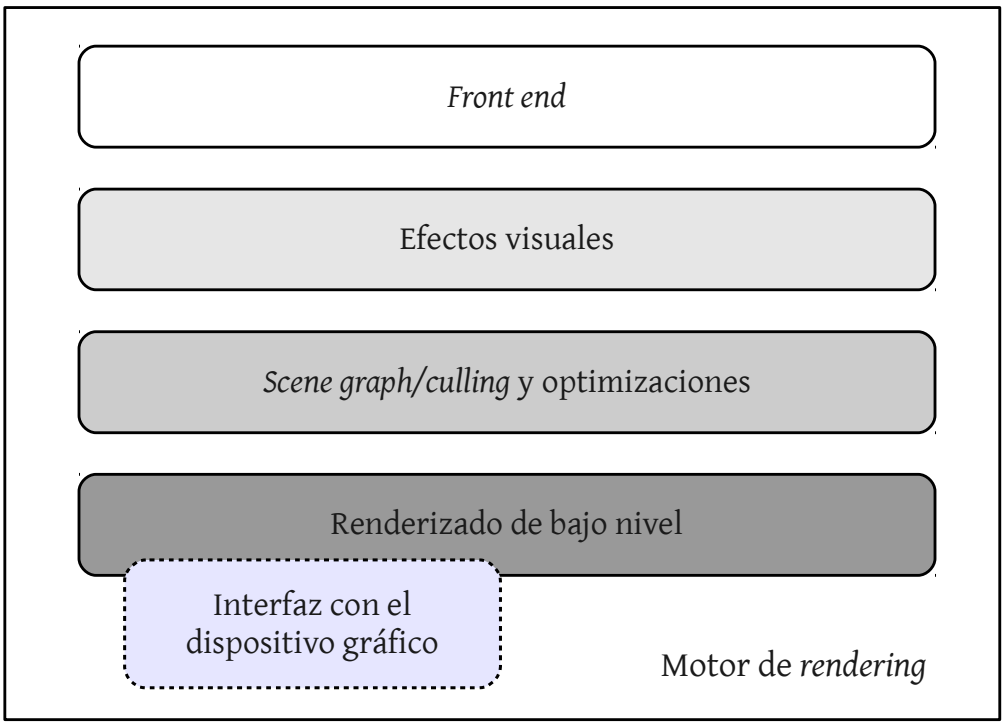

Figura 1.11: Visión conceptual de la arquitectura general de un motor de rendering. Esquema simplificado de la arquitectura discutida en [5].

vas geométricas, así como de la gestión de la cámara y los diferentes modos de proyección. En otras palabras, esta capa proporciona una serie de abstracciones para manejar tanto las primitivas geométricas como las cámaras virtuales y las propiedades vinculadas a las mismas.

Por otra parte, dicha capa también gestiona el estado del hardware gráfico y los *shaders* asociados. Básicamente, cada primitiva recibida por esta capa tiene asociado un material y se ve afectada por diversas fuentes de luz. Así mismo, el material describe la textura o texturas utilizadas por la primitiva y otras cuestiones como por ejemplo qué *pixel* y *vertex shaders* se utilizarán para renderizarla.

La capa superior a la de renderizado de bajo nivel se denomina ***scene graph/culling* y optimizaciones** y, desde un punto de vista general, es la responsable de seleccionar qué parte o partes de la escena se enviarán a la capa de *rendering*. Esta selección, u optimización, permite incrementar el rendimiento del motor de *rendering*, debido a que se limita el número de primitivas geométricas enviadas a la capa de nivel inferior.

Aunque en la capa de *rendering* sólo se dibujan las primitivas que están dentro del campo de visión de la cámara, es decir, dentro del *viewport*, es posible aplicar más optimizaciones que simplifiquen la complejidad de la escena a renderizar, obviando aquellas partes de la misma que no son visibles desde la cámara. Este tipo de optimizaciones son críticas en juegos que tenga una complejidad significativa con el objetivo de obtener tasas de *frames* por segundo aceptables.

Una de las optimizaciones típicas consiste en hacer uso de estructuras de datos de *subdivisión espacial* para hacer más eficiente el renderizado, gracias a que es posible determinar de una manera rápida el conjunto de objetos potencialmente visibles. Dichas estructuras de datos suelen ser árboles, aunque también es posible utilizar otras alternativas. Tradicionalmente, las subdivisiones espaciales se conocen como *scene graph* (grafo de escena), aunque en realidad representan un caso particular de estructura de datos.

Por otra parte, en esta capa también es común integrar métodos de *culling*, como por ejemplo aquellos basados en utilizar información relevante de las oclusiones para determinar qué objetos están siendo solapados por otros, evitando que los primeros se tengan que enviar a la capa de *rendering* y optimizando así este proceso.

Idealmente, esta capa debería ser independiente de la capa de renderizado, permitiendo así aplicar distintas optimizaciones y abstrayéndose de la funcionalidad relativa al dibujado de primitivas. Un ejemplo representativo de esta independencia está representado por OGRE (Object-Oriented Graphics Rendering Engine) y el uso de la filosofía *plug & play*, de manera que el desarrollador puede elegir distintos diseños de grafos de escenas ya implementados y utilizarlos en su desarrollo.

> Filosofía *Plug & Play*
>
> Esta filosofía se basa en hacer uso de un componente funcional, hardware o software, sin necesidad de configurar ni de modificar el funcionamiento de otros componentes asociados al primero.

Sobre la capa relativa a las optimizaciones se sitúa la capa de **efectos visuales**, la cual proporciona soporte a distintos efectos que, posteriormente, se puedan integrar en los juegos desarrollados haciendo uso del motor. Ejemplos representativos de módulos que se incluyen en esta capa son aquéllos responsables de gestionar los sistemas de partículos (humo, agua, etc), los mapeados de entorno o las sombras dinámicas.

Finalmente, la capa de ***front-end*** suele estar vinculada a funcionalidad relativa a la superposición de contenido 2D sobre el escenario 3D. Por ejemplo, es bastante común utilizar algún tipo de módulo que per-

mita visualizar el menú de un juego o la interfaz gráfica que permite conocer el estado del personaje principal del videojuego (inventario, armas, herramientas, etc). En esta capa también se incluyen componentes para reproducir vídeos previamente grabados y para integrar secuencias cinemáticas, a veces interactivas, en el propio videojuego. Este último componente se conoce como IGC (In-Game Cinematics) *system*.

1.2.7. Herramientas de depuración

Debido a la naturaleza intrínseca de un videojuego, vinculada a las aplicaciones gráficas en tiempo real, resulta esencial contar con buenas herramientas que permitan depurar y optimizar el propio motor de juegos para obtener el mejor rendimiento posible. En este contexto, existe un gran número de herramientas de este tipo. Algunas de ellas son herramientas de propósito general que se pueden utilizar de manera externa al motor de juegos. Sin embargo, la práctica más habitual consiste en construir herramientas de *profiling*, vinculadas al análisis del rendimiento, o depuración que estén asociadas al propio motor. Algunas de las más relevantes se enumeran a continuación [5]:

> Versiones *beta*
>
> Además del uso extensivo de herramientas de depuración, las desarrolladoras de videojuegos suelen liberar versiones betas de los mismos para que los propios usuarios contribuyan en la detección de *bugs*.

- Mecanismos para determinar el tiempo empleado en ejecutar un fragmento específico de código.
- Utilidades para mostrar de manera gráfica el rendimiento del motor mientras se ejecuta el juego.
- Utilidades para volcar *logs* en ficheros de texto o similares.
- Herramientas para determinar la cantidad de memoria utilizada por el motor en general y cada subsistema en particular. Este tipo de herramientas suelen tener distintas vistas gráficas para visualizar la información obtenida.
- Herramientas de depuración que gestión el nivel de información generada.
- Utilidades para grabar eventos particulares del juego, permitiendo reproducirlos posteriormente para depurar *bugs*.

1.2.8. Motor de física

La detección de colisiones en un videojuego y su posterior tratamiento resultan esenciales para dotar de realismo al mismo. Sin un mecanismo de detección de colisiones, los objetos se *traspasarían* unos a otros y no sería posible interactuar con ellos. Un ejemplo típico de colisión está representado en los juegos de conducción por el choque entre dos o más vehículos. Desde un punto de vista general, el sistema de detección de colisiones es responsable de llevar a cabo las siguientes tareas [1]:

1. La **detección de colisiones**, cuya salida es un valor lógico indicando si hay o no colisión.
2. La **determinación de la colisión**, cuya tarea consiste en calcular el punto de intersección de la colisión.
3. La **respuesta a la colisión**, que tiene como objetivo determinar las acciones que se generarán como consecuencia de la misma.

> ED auxiliares
>
> Al igual que ocurre en procesos como la obtención de la posición de un enemigo en el mapa, el uso extensivo de estructuras de datos auxiliares permite obtener soluciones a problemas computacionalmente complejos. La gestión de colisiones es otro proceso que se beneficia de este tipo de técnicas.

Debido a las restricciones impuestas por la naturaleza de tiempo real de un videojuego, los mecanismos de gestión de colisiones se suelen aproximar para simplificar la complejidad de los mismos y no reducir el rendimiento del motor. Por ejemplo, en algunas ocasiones los objetos 3D se aproximan con una serie de líneas, utilizando técnicas de intersección de líneas para determinar la existancia o no de una colisión. También es bastante común hacer uso de árboles BSP para representar el entorno y optimizar la detección de colisiones con respecto a los propios objetos.

Por otra parte, algunos juegos incluyen sistemas realistas o semirealistas de simulación dinámica. En el ámbito de la industria del videojuego, estos sistemas se suelen denominar ***sistema de física*** y están directamente ligados al sistema de gestión de colisiones.

Actualmente, la mayoría de compañías utilizan motores de colisión/-física desarrollados por terceras partes, integrando estos *kits* de desarrollo en el propio motor. Los más conocidos en el ámbito comercial son *Havok*, el cual representa el estándar de facto en la industria debido a su potencia y rendimiento, y *PhysX*, desarrollado por *NVIDIA* e integrado en motores como por ejemplo el del *Unreal Engine 3*.

En el ámbito del *open source*, uno de los más utilizados es ODE. Otra opción muy interesante es *Bullet*[14], el cual se utiliza actualmente en proyectos tan ambiciosos como la suite 3D *Blender*.

1.2.9. Interfaces de usuario

En cualquier tipo de juego es necesario desarrollar un módulo que ofrezca una abstracción respecto a la interacción del usuario, es decir, un módulo que principalmente sea responsable de procesar los **eventos de entrada** del usuario. Típicamente, dichos eventos estarán asociados a la pulsación de una tecla, al movimiento del ratón o al uso de un *joystick*, entre otros.

Desde un punto de vista más general, el módulo de interfaces de usuario también es responsable del **tratamiento de los eventos** de salida, es decir, aquellos eventos que proporcionan una retroalimentación al usuario. Dicha interacción puede estar representada, por ejemplo, por el sistema de vibración del mando de una consola o por la fuerza ejercida por un volante que está siendo utilizado en un juego de conducción. Debido a que este módulo gestiona los eventos de entrada y de salida, se suele denominar comúnmente *componente de entrada/salida del jugador* (*player I/O component*).

El módulo de interfaces de usuario actúa como un puente entre los detalles de bajo nivel del hardware utilizado para interactuar con el juego y el resto de controles de más alto nivel. Este módulo también es responsable de otras tareas importantes, como la asocación de acciones o funciones lógicas al sistema de control del juego, es decir, permite asociar eventos de entrada a acciones lógicas de alto nivel.

En la gestión de eventos se suelen utilizar patrones de diseño como el patrón *delegate* [3], de manera que cuando se detecta un evento, éste se traslada a la entidad adecuada para llevar a cabo su tratamiento.

1.2.10. *Networking* y multijugador

La mayoría de juegos comerciales desarrollados en la actualidad incluyen modos de juegos multijugador, con el objetivo de incrementar la jugabilidad y duración de los títulos lanzados al mercado. De hecho, algunas compañías basan el modelo de negocio de algunos de sus juegos en el **modo *online***, como por ejemplo *World of Warcraft* de *Blizzard Entertainment*, mientras algunos títulos son ampliamente conocidos por su exitoso modo multijugador *online*, como por ejemplo la saga *Call of Duty* de *Activision*.

[14] http://www.bulletphysics.com

El retraso que se produce desde que se envía un paquete de datos por una entidad hasta que otra lo recibe se conoce como *lag*. En el ámbito de los videojuegos, el *lag* se suele medir en milésimas de segundo.

Aunque el modo multijugador de un juego puede resultar muy parecido a su versión *single-player*, en la práctica incluir el soporte de varios jugadores, ya sea *online* o no, tiene un profundo impacto en diseño de ciertos componentes del motor de juego, como por ejemplo el modelo de objetos del juego, el motor de renderizado, el módulo de entrada/salida o el sistema de animación de personajes, entre otros. De hecho, una de las filosofías más utilizadas en el diseño y desarrollo de motores de juegos actuales consiste en tratar el modo de un único jugador como un caso particular del modo multijugador.

Por otra parte, el **módulo de *networking*** es el responsable de *informar* de la evolución del juego a los distintos actores o usuarios involucrados en el mismo mediante el envío de paquetes de información. Típicamente, dicha información se transmite utilizando *sockets*. Con el objetivo de reducir la latencia del modo multijugador, especialmente a través de Internet, sólo se envía/recibe información relevante para el correcto funcionamiento de un juego. Por ejemplo, en el caso de los FPS, dicha información incluye típicamente la posición de los jugadores en cada momento, entre otros elementos.

1.2.11. Subsistema de juego

El subsistema de juego, conocido por su término en inglés *gameplay*, integra todos aquellos módulos relativos al funcionamiento interno del juego, es decir, aglutina tanto las propiedades del mundo virtual como la de los distintos personajes. Por una parte, este subsistema permite la definición de las reglas que gobiernan el mundo virtual en el que se desarrolla el juego, como por ejemplo la necesidad de derrotar a un enemigo antes de enfrentarse a otro de mayor nivel. Por otra parte, este subsistema también permite la definición de la mecánica del personaje, así como sus objetivos durante el juego.

Este subsistema sirve también como capa de *aislamiento* entre las capas de más bajo nivel, como por ejemplo la de *rendering*, y el propio funcionamiento del juego. Es decir, uno de los principales objetivos de diseño que se persiguen consiste en independizar la lógica del juego de la implementación subyacente. Por ello, en esta capa es bastante común encontrar algún tipo de sistema de ***scripting*** o lenguaje de alto

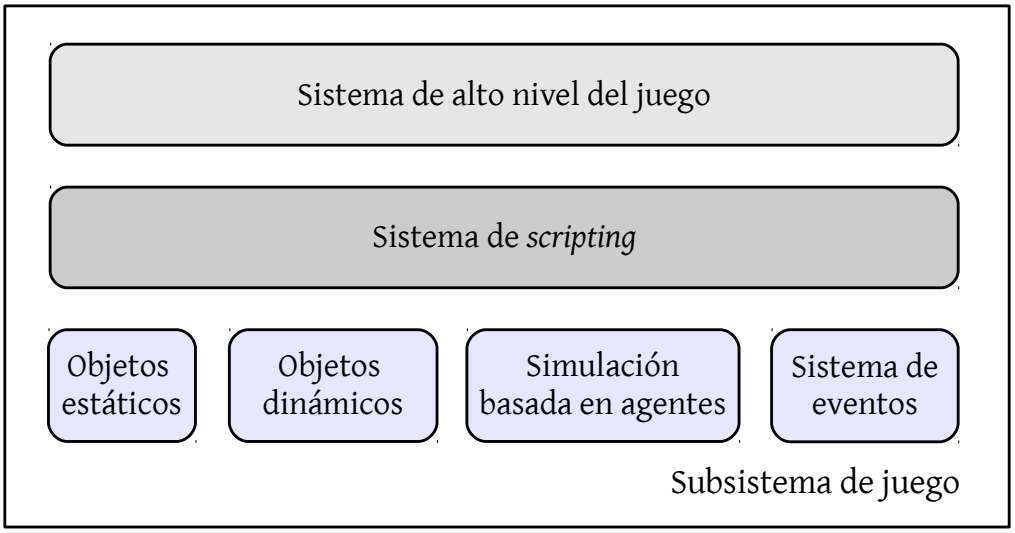

Figura 1.12: Visión conceptual de la arquitectura general del subsistema de juego. Esquema simplificado de la arquitectura discutida en [5].

nivel para definir, por ejemplo, el comportamiento de los personajes que participan en el juego.

Los diseñadores de los niveles de un juego, e incluso del comportamiento de los personajes y los NPCs, suelen dominar perfectamente los lenguajes de script, ya que son su principal herramienta para llevar a cabo su tarea.

La capa relativa al subsistema de juego maneja conceptos como el *mundo del juego*, el cual se refiere a los distintos elementos que forman parte del mismo, ya sean estáticos o dinámicos. Los tipos de objetos que forman parte de ese mundo se suelen denominar *modelo de objetos del juego* [5]. Este modelo proporciona una simulación en tiempo real de esta colección heterogénea, incluyendo

- Elementos geométricos relativos a fondos estáticos, como por ejemplo edificios o carreteras.
- Cuerpos rígidos dinámicos, como por ejemplo rocas o sillas.
- El propio personaje principal.

- Los personajes no controlados por el usuario (NPCs).
- Cámaras y luces virtuales.
- Armas, proyectiles, vehículos, etc.

El modelo de objetos del juego está intimamente ligado al *modelo de objetos software* y se puede entender como el conjunto de propiedades del lenguaje, políticas y convenciones utilizadas para implementar código utilizando una filosofía de orientación a objetos. Así mismo, este modelo está vinculado a cuestiones como el lenguaje de programación empleado o a la adopción de una política basada en el uso de patrones de diseño, entre otras.

En la capa de subsistema de juego se integra el **sistema de eventos**, cuya principal responsabilidad es la de dar soporte a la comunicación entre objetos, independientemente de su naturaleza y tipo. Un enfoque típico en el mundo de los videojuegos consiste en utilizar una *arquitectura dirigida por eventos*, en la que la principal entidad es el evento. Dicho evento consiste en una estructura de datos que contiene información relevante de manera que la comunicación está precisamente guiada por el contenido del evento, y no por el emisor o el receptor del mismo. Los objetos suelen implementar manejadores de eventos (*event handlers*) para tratarlos y actuar en consecuencia.

Por otra parte, el **sistema de *scripting*** permite modelar fácilmente la lógica del juego, como por ejemplo el comportamiento de los enemigos o NPCs, sin necesidad de volver a compilar para comprobar si dicho comportamiento es correcto o no. En algunos casos, los motores de juego pueden seguir en funcionamiento al mismo tiempo que se carga un nuevo *script*.

Finalmente, en la capa del subsistema de juego es posible encontrar algún módulo que proporcione funcionalidad añadida respecto al tratamiento de la IA, normalmente de los NPCs). Este tipo de módulos, cuya funcionalidad se suele incluir en la propia capa de software específica del juego en lugar de integrarla en el propio motor, son cada vez más populares y permiten asignar comportamientos preestablecidos sin necesidad de programarlos. En este contexto, la *simulación basada en agentes* [17] cobra especial relevancia.

Este tipo de módulos pueden incluir aspectos relativos a problemas clásicos de la IA, como por ejemplo la búsqueda de caminos óptimos entre dos puntos, conocida como *pathfinding*, y típicamente vinculada al uso de algoritmos *A** [11]. Así mismo, también es posible hacer uso de información *privilegiada* para optimizar ciertas tareas, como por ejemplo la localización de entidades de interés para agilizar el cálculo de aspectos como la detección de colisiones.

1.2.12. Audio

Tradicionalmente, el mundo del desarrollo de videojuegos siempre ha prestado más atención al componente gráfico. Sin embargo, el apartado sonoro también tiene una gran importancia para conseguir una inmersión total del usuario en el juego. Por ello, el **motor de audio** ha ido cobrando más y más relevancia.

Así mismo, la aparición de nuevos formatos de audio de alta definición y la popularidad de los sistemas de cine en casa han contribuido a esta evolución en el cada vez más relevante apartado sonoro.

Actualmente, al igual que ocurre con otros componentes de la arquitectura del motor de juego, es bastante común encontrar desarrollos listos para utilizarse e integrarse en el motor de juego, los cuales han sido realizados por compañías externas a la del propio motor. No obstante, el apartado sonoro también requiere modificaciones que son específicas para el juego en cuestión, con el objetivo de obtener un alto de grado de fidelidad y garantizar una buena experiencia desde el punto de visto auditivo.

1.2.13. Subsistemas específicos de juego

Por encima de la capa de subsistema de juego y otros componentes de más bajo nivel se sitúa la capa de subsistemas específicos de juego, en la que se integran aquellos módulos responsables de ofrecer las características propias del juego. En función del tipo de juego a desarrollar, en esta capa se situarán un mayor o menor número de módulos, como por ejemplo los relativos al sistema de cámaras virtuales, mecanismos de IA específicos de los personajes no controlados por el usuario (NPCs), aspectos de renderizados específicos del juego, sistemas de armas, puzzles, etc.

Idealmente, la línea que separa el motor de juego y el propio juego en cuestión estaría entre la capa de subsistema de juego y la capa de subsistemas específicos de juego.