universidad de león

Escuela de Ingenierías I.I.

Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Diseño y desarrollo de un motor de videojuego utilizando C++ y SFML. Ejemplo de uso: Graphiure

Autor: Dorian Cadenas Álvarez

Tutor: Lidia Sánchez González

UNIVERSIDAD DE LEÓN
Escuela de Ingenierías I.I.
GRADO EN INGENIERÍA INFORMÁTICA
Trabajo de Fin de Grado

ALUMNO: Dorian Cadenas Álvarez

TUTOR: Lidia Sánchez González

TÍTULO: Diseño y desarrollo de un motor de videojuego utilizando C++ y SFML. Ejemplo de uso: Graphiure

CONVOCATORIA: Julio, 2015

RESUMEN:

El presente Trabajo Fin de Grado (TFG) consiste en el estudio y desarrollo de un motor de videojuegos. Los objetivos que se marcan es comprender las bases de un motor de videojuego, entender sus fundamentos y arquitecturas, diferentes enfoques con sus ventajas y desventajas y comprender cómo afecta cada una de sus partes al desarrollo y ejecución del juego final que se desarrolla sobre el motor. Para ello, se ha realizado un proyecto inicial, intentando clonar el juego del Tetris, y con el que realizar la toma de contacto con el lenguaje C++ y las librerías SDL y SFML. Mientras, se procedía a la lectura y estudio de bibliografía recomendada, en los que se analiza la arquitectura de los motores, los diferentes enfoques con sus ventajas y desventajas y algunos motores de ejemplo que implementan dichas estrategias. Posteriormente, con las bases establecidas y escogido la librería SFML para la realización del proyecto, se procedió a programar un motor sencillo aplicando algunas de las estrategias aprendidas y entender y experimentar de primera mano la estructura, funcionamiento y complejidad de un motor. Para ello se desarrolló conjuntamente un esbozo de videojuego que permite probar y observar el efecto del motor sobre el juego. Para el juego se utilizó como ayuda la librería tinyXML2 para el manejo de recursos. El resultado del proyecto resulta satisfactorio tanto en resultados técnicos del motor realizado como en el cumplimiento de los objetivos marcados.

Palabras clave: motor de videojuego, arquitectura de motor, videojuego.

Firma del alumno:	**VºBº Tutor:**

Índice de contenidos

Índice de figuras

Índice de tablas

1 Introducción

Los videojuegos hoy en día se mantienen como la principal industria de ocio audiovisual e interactivo. Se ha convertido en uno de los más grandes negocios y se prevee que crezca aún más. La mayoría de los videojuegos tienen una cosa en común: la base sobre la que se construyen, denominado motor de videojuego. En la actualidad, hay diversos motores para diversos tipos de videojuegos, algunos más adecuados para móviles, otros para consolas o PCs, otros trabajan mejor en 2D o en 3D, etc.

A excepción de los grandes grupos de trabajo en proyectos grandes o empresas grandes, muchos de los desarrolladores hoy en día producen videojuegos en diversas plataformas con ayuda de motores que les liberan de la tediosa tarea de programar, reduciendo éste al mínimo y siendo ése el abanderado objetivo de los grandes motores de videojuegos.

Sin embargo, tiene un coste, no conocen cómo funciona un motor de videojuegos internamente y están limitados al motor, encontrándose con situaciones en las que quisieran hacer algo concreto y el motor no se lo permite, teniendo que adaptar su idea o buscar otro motor que se ajuste a lo que precisan. Ante el ojo entrenado, una persona puede detectar el motor utilizado en un determinado juego debido a las restricciones que imponen los motores para conseguir esa propiedad de no tener que programar.

En el presente documento se profundiza en los misterios del motor, para conocer su compleja arquitectura y tratar de hacer una pequeña réplica para comprender su funcionamiento y ver de primera mano sus puntos fuertes y débiles. El objetivo final es saber cómo funciona, para conocer de antemano el mejor motor a usar, y si hay que hacer alguna modificación o cambio, saber por dónde ir.

1.1 Motivación

De siempre me han llamado la atención los videojuegos y cómo se desarrollan. En un principio quise realizar un juego como proyecto, pero me fijé en que muchos juegos hoy en día, y refiriéndome a juegos normales desarrollados por pocas personas, se apoyan en motores de videojuegos que llevan por bandera el desarrollar un juego sin tener que programar casi nada.

Ante esa situación, decidí ir en contra de todos los consejos y avisos, y tratar de realizar un motor de videojuego sencillo para ver qué se está haciendo realmente bajo un videojuego y no dejarme llevar por la facilidad y rapidez de hacer tantos juegos se quieran con los motores más actuales que hay, sin conocer bien lo que se está haciendo.

1.2 Objetivo

El objetivo del proyecto es estudiar el motor de videojuego. Para ello se pretende:

- Estudiar las bases de un motor de videojuego.
- Entender los fundamentos y arquitecturas del motor.
- Ver distintos enfoques y estrategias posibles, con sus ventajas y desventajas, y ejemplos de motores o juegos que lo implementan.
- Comprender cómo afecta cada una de las partes del motor al desarrollo y ejecución del juego final que se desarrolla sobre el motor.
- Realizar un motor sencillo para aplicar algunos conocimientos adquiridos.
- Experimentar con el motor para comprobar su estructura, funcionamiento y la complejidad inherente de los motores de videojuegos.

1.3 Recursos

Los recursos de los que se dispone y que se han usado en el desarrollo son:

- Un ordenador utilizado como servidor para el control de versiones con un *Pentium Processor* G2030 de 3GHz, memoria RAM de 4GB y con SO Ubuntu de 64 bits.
- Un ordenador sobremesa con placa base *Maximus* IV, procesador Intel i7, 12GB de RAM y tarjeta gráfica AMD Gigabyte HD 6870. Sistema operativo *ArchLinux*.
- Un portátil Acer con Intel Core i5 y gráfica *Nvidia GeForce* GT. Sistema operativo *ArchLinux*.

1.4 Estructura del documento

En el presente documento se podrán distinguir dos partes claras. Obviando el apartado 1, donde se detallan las motivaciones, objetivos y los recursos que se disponen para ello, se tiene los apartados 2 y 3 donde se explica la parte teórica del proyecto, y en el apartado 4 se detalla la parte práctica.

En el apartado 2, se define el motor de videojuego y alguna característica y se pasa a detallar su arquitectura y composición, empezando desde la base y avanzando en un cierto orden hasta llegar a la parte final de un motor, donde pertenecería el juego y que se explica en el apartado 3.

Dicha parte teórica no es totalmente utilizada en la parte práctica del proyecto, principalmente por su complejidad, avanzados conocimientos y escaso tiempo, al mismo tiempo tampoco es totalmente completo por la amplitud del trabajo. Sin embargo, es un conocimiento de importancia para comprender las bases y realizar un mejor trabajo.

El apartado 4 se centra en el proyecto realizado, donde se detalla el proceso de codificación seguido y las estrategias del motor desarrollado. También se pueden encontrar algunos tips y detalles que han surgido durante el desarrollo.

A continuación, se expone una breve descripción de los resultados y conclusiones obtenidas en el apartado 5 y por último, se expone unas posibles líneas de futuro en el apartado 6 y se adjunta en los anexos los manuales del usuario y del programador.

2 El motor de videojuego

2.1 Definición

El término de motor de videojuego apareció a los mediados de los 90 por los juegos *shooter* de primera persona (FPS) como los juegos de Doom de la compañía Id Software. (1)

Hace referencia a una serie de rutinas de programación que abstraen de la implementación de tareas comunes relacionadas con el juego, como son la representación gráfica 2D o 3D, la colisión, la física, la entrada de datos, sonido, inteligencia artificial, animación, scripting, redes, administración de memoria ...

El proceso de desarrollo de un videojuego puede variar notablemente por reutilizar o adaptar un motor de videojuego para crear diferentes juegos. (2)

La línea entre un juego y su motor, es a menudo borrosa. Algunos motores hacen una distinción clara mientras que otros casi no tienen separación alguna. En un juego, el motor podría saber cómo se dibuja exactamente un orco, mientras que en otro, ofrecería funciones generales y el orco se definiría enteramente con datos. Ningún estudio hace una perfecta separación, lo cual es comprensible ya que la definición de esos componentes a menudo cambia con la consolidación del diseño del juego.

Por ello, nos podemos encontrar con motores que no pueden ser usados más que en un juego en concreto como pudiera ser el motor del Pac-man, otros que sirven para juegos muy similares, o que pueden ser adaptados para juegos dentro de un género específico, y lo ideal, sería un motor que sirviera para cualquier juego imaginable. Pero esto último es hoy por hoy, imposible.

Esto ocurre debido a que cualquier trozo de código eficiente, implica necesariamente hacer concesiones, las cuales se basan en suposiciones sobre en cómo se usará el software o el objetivo hardware para el que se ejecuta. Un ejemplo sencillo es el renderizado de interiores y exteriores, ya que el motor que renderice un interior, se centrará en las partes que quedan ocultas a la cámara, como objetos detrás de paredes y hará cálculos exactos, los cuales pierden importancia en exteriores con objetos lejanos, donde cobra importancia el nivel de detalle de los objetos al tener más cosas a dibujar.

Reservamos pues, el término motor de juego para el software extensible y reutilizable para varios juegos sin mayores modificaciones. No es del todo exacto, ya que normalmente se pueden reutilizar sólo en juegos similares (o del mismo género). En ocasiones los motores también pueden servir a otros géneros distintos al que fueron diseñados inicialmente. (1)

2.1.1 Tipos de motores

Los motores de videojuegos pueden venir en varios niveles de programación y madurez, siendo bastante diferentes entre ellos. Para hacer una clasificación rápida, se tienen tres tipos de motores (3):

- ***Roll-your-own game engines*** (el más bajo nivel). A pesar del coste y tiempo que lleva hacer uno, hay gente y compañías que hacen y lanzan su propio motor por variados motivos. Utilizan interfaces de aplicaciones disponibles al público, como las APIs de XNA, DirectX, OpenGL, SDL... para crear sus propios motores. También pueden utilizar otras librerías, ya sea de código abierto o comercial, para facilitar el desarrollo.

  En general, ofrecen más flexibilidad y tienen los componentes y la integración tal y como desean. El único problema es la cantidad de tiempo que lleva y por ello suele ser la opción menos preferida.

- **Motores prácticamente listos** (nivel medio). Son la mayoría de los motores, los cuales están listos para funcionar, ofreciendo el renderizado, las colisiones, las físicas... y suelen ser motores ya maduros, realizados por mucha gente y muchas horas de trabajo. Requieren un poco de programación, incluyendo posibles códigos de bajo nivel y son más restrictivos, pero más optimizados en sus tareas.

- **Motores de apuntar y clickar**. Los motores de más alto nivel, los cuales están siendo cada vez más comunes. Están construidos para ser lo más amigables al usuario posible e incluyen herramientas que permiten crear un juego a base de apuntar y hacer click, con un mínimo de programación.

  El principal problema es que pueden ser muy restrictivos y limitados a uno o dos géneros de juegos o a un par de tipos de gráficos distintos. A pesar de todo, es posible hacer grandes juegos con estos motores y permiten hacer un trabajo rápido sin mucho trabajo.

2.1.2 Motores dirigidos por datos

Al principio, los juegos eran programados directamente en el código al ser pequeños y favorecido por lo primitivo de los gráficos y sonidos con los que el hardware podía. Hoy en día ya son mucho más complejos y de mayor calidad, además de ser mucho más grandes en contenidos.

Los recursos de los ingenieros son normalmente un cuello de botella debido a que el talento es más limitado y caro y a que producen contenidos más lentamente que la otra parte del equipo, los artistas y diseñadores de juegos debido precisamente a la complejidad de los juegos.

Por ello, se consideró buena idea poner al menos parte del poder para crear contenido directamente en manos de los responsables de producir dicho contenido (artistas y diseñadores).

Cuando el comportamiento del juego puede ser controlado, en parte o en su totalidad, por datos provistos por los artistas y diseñadores, se dice que el *engine* es dirigido por datos.

Las arquitecturas dirigidas por datos pueden mejorar la eficiencia de los equipos y sobre todo su tiempo al no tener que recompilar el producto cada vez que se hace un pequeño cambio en el diseño o nivel, siendo de las cosas que más cambia en el juego.

Sin embargo, esto viene con un coste, las herramientas deben permitir que se defina el juego de una manera dirigida por datos y el código que corre en tiempo de ejecución tiene que manejar la amplia posible entrada de una forma robusta.

Todo *engine* debería tener algunos componentes como dirigidos por datos, pero no todos o se cae en el riesgo de perder mucha productividad y complicar mucho el motor, asi que hay que escoger bien y seguir el mantra KISS (*Keep It Simple, Stupid* – Mantenlo simple, estúpido). (1)

2.2 Géneros de juegos y las características de sus motores

Los motores de juegos son típicamente específicos a un género concreto. Un motor diseñado para un juego de peleas entre dos en un ring, será muy diferente de un juego multijugador masivo online, de un *shooter* en primera persona o de un juego de estrategia en tiempo real.

Sin embargo, también hay muchas partes en las que se solapan. En todos los juegos 3D e incluso 2D, independientemente del género al que pertenecen, requieren de alguna forma de entrada del usuario desde un teclado, un joystick y/o un ratón, un HUD (*Heads-Up Display* = Presentación de información) incluyendo el renderizado de texto, el audio, renderizado de 3D en malla... (1)

1. *Shooters* de primera persona (FPS)

Ejemplos de *shooters* son Call of Duty, Quake y Half-Life entre otros. Esos juegos pueden estar en una gran variedad de entornos, desde interiores confinados hasta exteriores grandes e incluyen una variedad de armas. Los FPS más modernos pueden incluir vehículos y su libre albedrío en cuando a itinerancia. (4)

Es uno de los géneros más exigentes en cuanto al motor gráfico ya que su finalidad es sumergir al jugador en el juego, con un mundo hiperrealista y detallado. El género suele centrarse en tecnologías tales como:

- Eficiencia en el renderizado de grandes mundos virtuales en 3D
- Gran fidelidad en las animaciones de las armas y brazos del jugador
- Una alta variedad de armas
- Mecanismo de control de cámara sensible
- Un modelo de colisión y movimiento del personaje permisivo
- Escalable para capacidades multijugador online

2. Plataformas y otros juegos de tercera persona

Incluyendo juegos en 2D y 3D, los juegos de plataforma son típicamente los juegos cuya principal mecanismo de juego es saltar entre plataformas y/o obstáculos. No se incluyen aquellos en los que el salto es automático. Este género tiene muchos juegos conocidos como Mario, Sonic, Donkey Kong y otros más actuales. (5)

Estos juegos tienen mucho en común con los *shooters* de primera persona, con la excepción de enfatizar en el personaje, sus habilidades y locomoción. Por lo que exige más calidad para el personaje. El hecho de estar más enlazado con los dibujos animados y no con el realismo, permite también exigir menos en el renderizado.

Las tecnologías en las que suele estar enfocado son:

- Movimiento de plataformas, cuerdas, escaleras, enrejados y otros modos de movimiento interesantes
- Elementos del entorno como partes de un puzzle
- Una cámara en tercera persona siguiendo al personaje y usualmente controlado por el jugador
- Un complejo sistema de colisión para la cámara que permita no perder de vista al personaje

3. *Fighting games*

Típicamente dos jugadores cuyos avatares se pelean en una zona tal como un ring. Representados por juegos como Tekken o Soul Calibur. (6)

Su tecnología es más sencilla, enfocándose en:

- Un gran conjunto de animaciones de pelea
- Gran precisión en la detección de colisión
- Un sistema de entrada capaz de detectar combinaciones complejas de botones y joysticks
- Fondos relativamente estáticos

Con los avances, se han ido introduciendo otras características como

- Gran definición de los personajes
- Gran fidelidad de la animación de los personajes
- Mejor realismo en la física asociada al pelo y ropas de los personajes

Algunos juegos de lucha ocurren en un mundo virtual más grande y no confinado en una arena. Se suele considerar un género aparte y comparten muchos requerimientos con los *shooters* de primera persona o los juegos de estrategia en tiempo real.

4. Estrategia en tiempo real

Es un género que no progresa por turnos. El jugador se encuentra en un campo de juego grande y despliega las unidades de batalla de forma estratégica en un intento de aplastar a su oponente. Normalmente incluye la posibilidad de crear unidades y estructuras durante el juego, los cuales están limitados por una serie de recursos también limitados. (7)

Ejemplos de juegos los encontramos en Age of Empires, Civilization y Warcraft entre otros.

Se suele restringir el cambio de los ángulos de visión para permitir ver a través de grandes distancias, lo cual permite usar optimizaciones varias en el renderizado. En los juegos más viejos se usaba una proyección ortográfica para simplificar mucho el renderizado, mientras que hoy en día a veces usan la proyección en perspectiva e incluso 3D real, pero ambos usan un mundo con base de rejilla, que delimita las construcciones y otros elementos, alineándolos de forma adecuada.

Las características del motor suelen ser:

- Cada unidad es de baja resolución, permitiendo al juego soportar un gran número en la pantalla al mismo tiempo
- El usuario a veces puede construir nuevas estructuras sobre el terreno
- La interacción del usuario es típicamente a través del ratón con simples clicks y selección en base a un área seleccionada, y a través de menús que contienen comandos, unidades, equipos...

5. MMOG

Juegos multijugadores masivos online (*Massively Multiplayer Online Game*), cuyo más claro ejemplo puede ser World of Warcraft. Consisten en un largo número de jugadores jugando al mismo tiempo. Por ese mismo motivo, necesariamente se juegan a través de internet y normalmente tienen al menos un mundo virtual persistente. (8)

Debido a ese largo número de jugadores, son muy exigentes con las bases de datos y los servidores donde corren el juego. La fidelidad de los gráficos son más bajos.

6. Otros géneros

Los juegos de carreras son bastante lineales, como lo fueron los FPS de antaño, sin embargo, su velocidad es mucho mayor, enfocan los detalles en los vehículos, pista y sus alrededores inmediatos y realizan trucos al renderizar objetos lejanos como puede ser utilizar imágenes en dos dimensiones para los árboles, colinas y montañas. De igual forma, la pista suele dividirse en regiones bidimensionales simples llamados sectores, los cuales se usan para optimizar la representación y la visibilidad, ayudar a la inteligencia artificial, buscar vehículos no controlados por el jugador y otros problemas técnicos.

También tiene importancia la cámara, pudiendo estar en tercera persona o en el interior del coche, y comprobar las colisiones en zonas estrechas como los túneles.

Los RPG (*Role playing game* o juegos de rol) son otro género que comparten mucho con los MMOG por ser dos géneros que van muy juntos. Consiste en tomar el rol del personaje y controlar sus acciones inmerso en un mundo virtual. Se basan en los juegos de rol clásicos, jugados con papel y lápiz, y su punto fuerte es la historia y la capacidad del personaje de crecer y hacerse fuerte.

Otros géneros no mencionados tienen también sus propios requerimientos. Por ello, los motores difieren entre sí según el género para el que están hechos e incluso cuando se mezclan los géneros. Sin embargo, hay bastantes tecnologías que se solapan entre sí, siendo comunes a lo largo de varios géneros y especialmente en el contexto del hardware, donde las diferencias por cuestiones de optimización empiezan a desaparecer. (1)

2.3 Arquitectura de un motor

Un motor de videojuego, o como es comúnmente conocido por su traducción en inglés, *videogame engine* o *engine* a secas, se compone normalmente de dos partes; una suite de herramientas y un componente de tiempo de ejecución. (1)

2.3.1 El componente de tiempo de ejecución

Como se puede ver en la figura 2.1, el componente en tiempo de ejecución puede ser bastante complejo. Se estructura en capas, de forma que las capas superiores del sistema depende de capas inferiores, pero no viceversa, lo que evita dependencias circulares que impiden la reusabilidad y dificultan la estabilidad y la modularidad.

- **Capas de hardware, drivers y sistema operativo.** El hardware representan la máquina, como puede ser un ordenador o una consola. Los drivers son los componentes software que manejan de forma directa el hardware y los dispositivos, abstrayendo la comunicación entre ellos y otros. El sistema operativo se construye encima de los drivers y está ejecutándose todo el tiempo, coordinando los programas que se ejecutan y los recursos de los que dispone. Aunque lo asociemos a sistemas operativos como Windows, Linux o Mac, el sistema operativo puede ser una simple capa de librería compilada de forma directa en el juego, como puede pasar en las consolas, con lo que el juego suele hacer suya la maquina al completo. Hoy en día ya no pasa por la diversidad de funciones que cumple una consola, poniéndose al nivel de ordenadores.
- **SDKs y middlewares de terceros**. Los *engines* suelen tener software de terceros a bajo nivel, como SDKs, que ofrecen APIs, o los middlewares. Ejemplos son:
    - Estructuras de datos y algoritmos: como el stl de C++ y Loki (una librería de plantillas de programación).
    - Gráficos: OpenGL, DirectX, libgcm (interfaz para el hardware grafico de la playstation 3), Edge (*engine* de animación de Sony y Naughty Dog) entre otros.
    - Colisiones y física como Havok, PhysX y Open Dynamics Engine (ODE)
    - Animación de caracteres como el propio Havok animation, ya que la línea entre la física y la animación es cada vez más borrosa, Granny (Una herramienta que incluye exportadores de modelos y animaciones 3D para la mayoría de modeladores y animadores como Maya, 3D Studio Max… etc, es una librería para la lectura y manipulación de los modelos y animaciones exportados y un sistema de animación en tiempo de ejecución.)
    - Inteligencia artificial, que hasta hace poco, se trataba de forma personal en cada juego. Kynogon creó Kynapse y provee un poco de IA como búsqueda de rutas, evitar obstáculos, identificación de vulnerabilidades explotables...

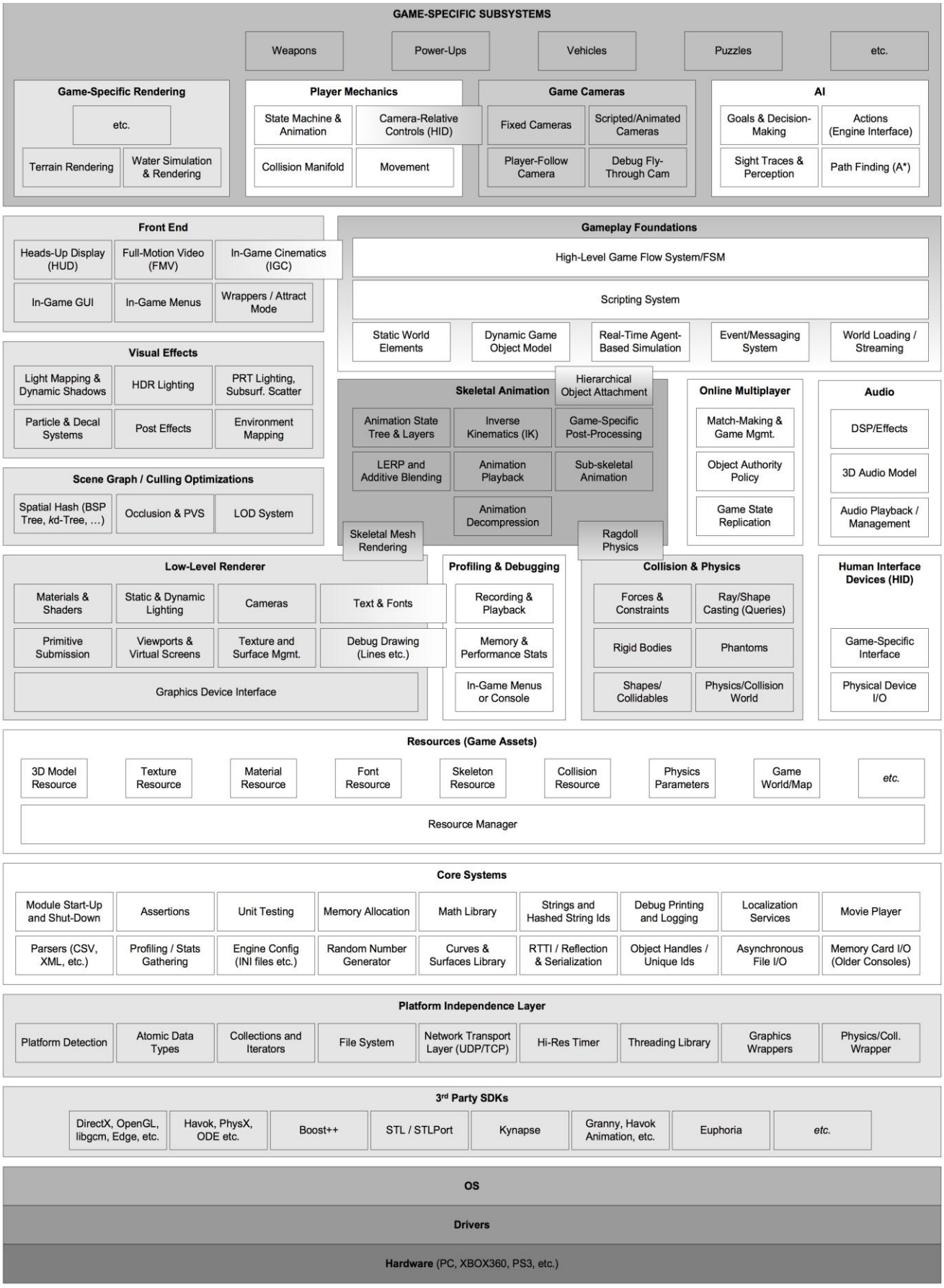


Figura 2-1. Arquitectura de un motor. Fuente: Game Engine Architecture, Jason Gregory. (1)

  - Biomecánica de los modelos de personajes, como hacen Endorphin y Euphoria, que producen animaciones de los personajes usando avanzados modelos biomecánicos de humanos reales. Este apartado también empieza a fundirse con la física y de ahí que Paquetes como la de Havok, traten de juntar la física con la animación.

- **Capa de plataforma independiente.** Donde se maneja las diferencias entre hardwares y sistemas operativos para poder ser ejecutados en más de una plataforma, lo que les permite un mayor alcance de mercado. Lo consigue realizando *wrappers* o sustituyendo llamadas a las más usadas librerías estándar de C, llamadas al sistema operativo y otros APIS básicos de la plataforma y así proporcionar consistencia entre plataformas incluso con librerías "estándar" como la de C.
- **Núcleo del sistema.** Se concentra las herramientas y utilidades básicas que requiere un *engine* como pueden ser lo siguiente:
  - Las aserciones son líneas de comprobación de errores en código que son escritas para capturar errores de lógica y violaciones de presupuestos hechos por el programador. Normalmente son retirados en la última compilación del juego.
  - Gestión de la memoria. Los motores implementan su propio manejo de la memoria para asegurarse de un alto rendimiento en las asignaciones y liberaciones y limitar los efectos negativos de la desfragmentación.
  - Librerías matemáticas: Los juegos son por naturaleza muy intensivos en las matemáticas, por lo que suelen tener no una, sino varias librerías matemáticas que les facilitan operaciones con vectores, matrices, trigonometría, geometría con líneas, esferas, integración numérica, resolución de sistemas de ecuaciones y varias cosas más.
  - Algoritmos y estructuras de datos propios. Excepto cuando el desarrollador decide fiarse de librerías externas como STL, normalmente se requiere tener un conjunto de estructuras y algoritmos básicos y más avanzados que suelen estar optimizados para evitar asignaciones dinámicas de memoria y asegurarse de un óptimo rendimiento en las plataformas objetivo.
- **Administrador de recursos**. Presente de alguna forma en todos los juegos, el administrador de recursos presenta una interfaz o varias para el acceso a cualquier recurso del juego, como texturas, mapas, ficheros... Algunos *engines*, como OGRE 3D lo hace de una forma enteramente centralizada y consistente y otros un poco más personalizado, dejando a veces al programador acceder de forma directa al recurso.
- **Motor de renderizado.** Uno de las más grandes y complejas capas del sistema. Se pueden estructurar de muchas formas. Una de las formas más usadas hoy en día siguen unas filosofías de diseño fundamentales, dirigidos principalmente por el diseño del hardware gráfico 3D del que dependen. Una forma común que se sigue es la de la imagen 2.2.

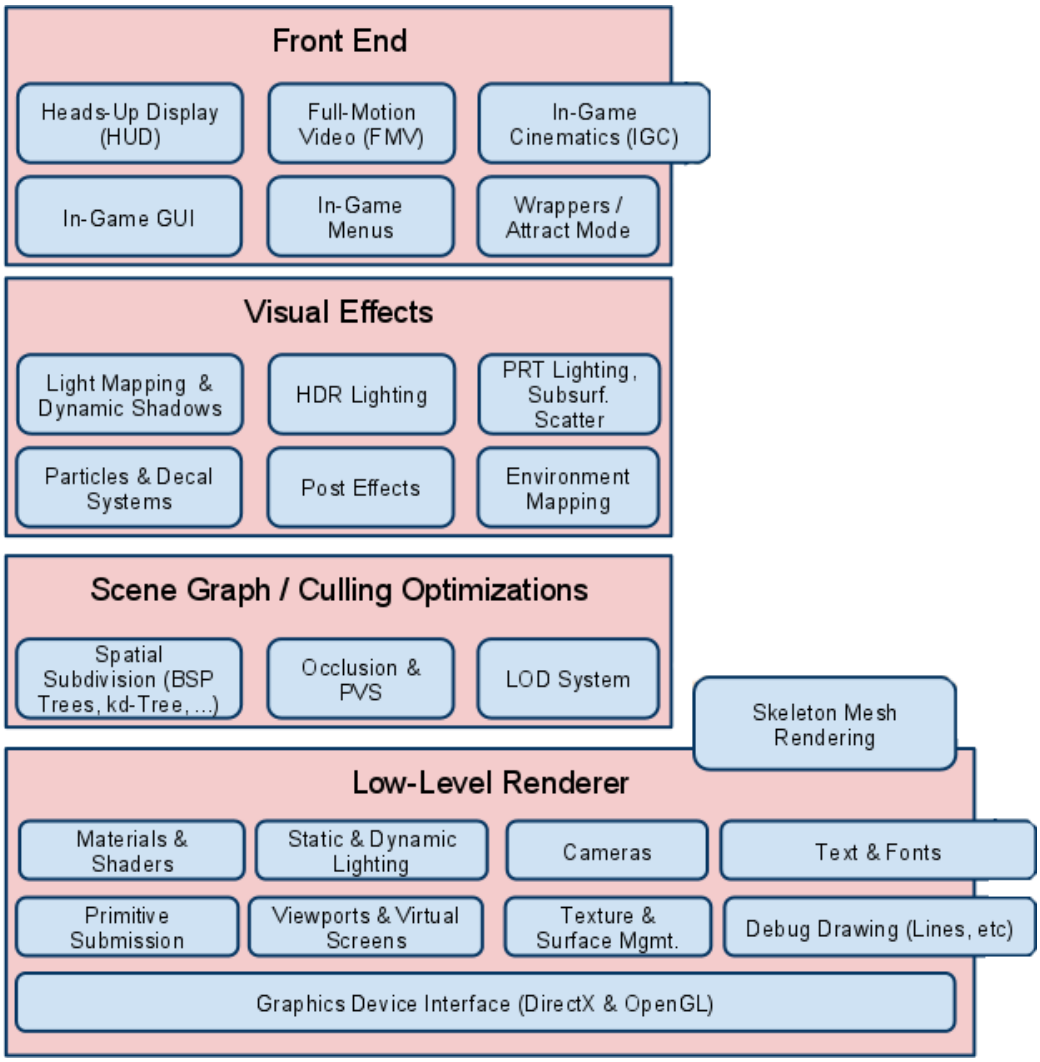


Figura 2-2. Motor de renderizado. Fuente: Game Engine Architecture, Jason Gregory. (1)

- Renderizado a bajo nivel, donde se enfoca a renderizar una serie de primitivas lo más rápido y rico posible, sin pararse mucho en lo que se ve y no. Se subdivide a su vez en

  - Interfaz de dispositivo gráfico. SDKs gráficos como DirectX y OpenGL requieren bastante código sólo para enumerar los gráficos disponibles, inicializarlos y setear la configuración entre otras muchas cosas. Por ello se suele tener un componente encargado de esa tarea. Se puede encontrar otro nombre distinto según la terminología que use el motor.

  - Otros componentes de renderizado. El resto de componentes cooperan para hacer primitivas geométricas, como líneas, puntos, partículas, texto, sombras y cualquier cosa que se quiera dibujar y rápido, sin mucho detalle.

  - Grafo de escena/Optimizaciones *culling*. Para juegos con mapas sencillos y pequeños, un simple borrado de los objetos que la cámara no ve puede valer, pero para mapas más grandes, se precisa de una más avanzada estructura de dato de subdivisión espacial para mejorar la eficiencia permitiendo determinar los objetos probables de ser vistos con rapidez. Esa subdivisión espacial puede ser de varias formas, como un árbol binario, un árbol n-ario o incluso una jerarquía esférica. También se le llama a veces un grafo de escena. Portales u oclusión *culling* pueden ser aplicados en esta capa del renderizado. De forma ideal, el renderizado de bajo nivel no tiene porqué saber nada del tipo de subdivisión espacial que usa la escena, lo que permite reutilizar código.

  - Efectos visuales como pueden ser sistemas de partículas (fuego, humo...), calcomanías, mapeo de entorno y de iluminación, sombras dinámicas, efectos de pantalla completa posteriores al renderizado de la imagen en el buffer (por ejemplo anti alising, corrección de color, desenfoque)...

  - *Front End*. Gráficos superpuestos a la escena por varios motivos como pueden ser los HUD, menús, consolas u otras herramientas de depuración o desarrollo (las cuales pueden ser luego lanzadas a producción o ser sólo para desarrollo) y probablemente cualquier interfaz gráfica que permita al jugador manipular elementos como el inventario del personaje, configurar unidades de batalla u otras tareas del juego complejas. En el libro de Game Engine Arquitecture también incluyen los videos y las cinemáticas del juego, que permite secuencias cinematográficas coreografiarse con el juego en sí como puede ser una conversación entre dos aldeanos mientras el jugador camina por ahí.

- **Herramientas de perfiles y depuración**. Los juegos son sistemas de tiempo real y por ello los ingenieros necesitan herramientas que permitan depurar fallos y comprobar el rendimiento para optimizar los recursos y comprobar fallos. Por ejemplo, suelen usar mucho las herramientas de análisis de la memoria al ser uno de los recursos más escasos. Las herramientas de depuración también permiten facilidades como un menú en el sistema o en consola, la habilidad de grabar y poder reproducir luego escenas y otros.

- **Colisiones y física**. Parte muy importante para todos los juegos. Suele estar fuertemente acoplado colisión y física ya que cuando las colisiones son detectadas, casi siempre se resuelven como una parte de la física y sus restricciones lógicas. Hoy en día casi nadie hace su propio *engine* de colisión/física, si no que usan uno tercero. Dos grandes conocidos son Havok y PhysX por Nvidia. Uno libre, es Open Dynamics Engine (ODE).

- **Animación.** Hay cinco tipos de animación usados en un juego:
    - Animación de sprite o textura
    - Animación de cuerpo rígido en jerarquía
    - Animación de esqueleto
    - Animación de vértices
    - Morph targets

  La animación de esqueleto es el más presente. Permite un personaje 3D detallado ser "posado" por un animador con un simple sistema de huesos. Al moverse los huesos, los vértices del modelo se mueven de forma acorde. Este renderizado se encuentra en cooperación entre los sistemas de renderizado y animación, pero con una interfaz bien definida. También hay colaboración con el sistema de físicas si se usa la técnica de *rag dolls* (muñecos de trapo), en el que el movimiento del personaje es simulado por la física que determina posición y orientación de las partes del cuerpo.

- **Dispositivos de Interfaz Humana (HID)**. Todo juego necesita procesar la entrada del jugador, que puede venir de varios dispositivos como el teclado, el ratón, un joystick, algo más especializado como la WiiMote, volantes... También se les llama a veces el componente I/O del jugador ya que también pueden darle una retroalimentación al jugador a través del dispositivo con la vibración, la resistencia de la tecla, sonidos y otras formas de retroalimentación. A veces permiten al jugador personalizar el mapeo de botones del mando al juego.

- **Audio**. Varía mucho en sofisticación.

- **Multijugador online y/o en red**. Los juegos multijugadores se pueden dividir en al menos cuatro formas:
    - Multijugador de una sola pantalla: Dos o más dispositivos de entrada se conectan y los jugadores coexisten en un único mundo virtual, mientras la cámara mantiene a todos los jugadores a la vista al mismo tiempo.

    - Multijugadores de pantalla partida: Con varios dispositivos y un único mundo virtual, la pantalla se divide en secciones, una por cada jugador donde tienen su propia cámara.

    - Multijugador en red: Múltiples ordenadores o consolas son conectados en red y cada máquina tiene un jugador.

    - Juegos multijugadores masivos online (*Massively multiplayer online games* (MMOG)): Cientos de jugadores juegan simultáneamente en un único mundo virtual gigante y persistente que es hospedado por una potente batería de servidores centrales.
      Son muy similares a los juegos de un único jugador, pero el soporte de varios jugadores tiene impacto en el diseño de ciertos

componentes del motor, siendo afectados la renderización, el modelo de objetos del mundo, la entrada del jugador, las animaciones y el sistema de control del jugador. Adaptar un motor para multijugador no es imposible pero si puede ser una tarea gigantesca, por lo que mejor diseñarlo desde el principio.

- **Sistema de fundamentos de *gameplay***. El término *gameplay* se refiere a las acciones que toman lugar en el juego, las reglas que gobiernan el mundo virtual, las habilidades del personaje/s (conocido también como mecanismos del jugador) y de otros personajes y objetos del juego, y los objetivos y metas del jugador. Normalmente es implementado en el mismo lenguaje que el resto del *engine*, en un alto nivel lenguaje scripting o ambos. Para pasar del juego en sí, y el bajo nivel del *engine*, se suele meter esta capa.
    - Mundos de juego y modelos de objetos
        - Objetos: Típicamente un fondo estático como edificios, terreno... personajes jugables, personajes no jugables, armas, proyectiles, vehículos, luces, cámaras, objetos rígidos como rocas, sillas...
        - El mundo del juego suele estar íntimamente atado al modelo de objeto software, lo que viene a ser las características del lenguaje, políticas y convenciones usadas en el software orientado a objetos. En el contexto del motor, viene a ser responder las siguientes preguntas entre otras:
            - ¿El motor está diseñado en una orientación a objetos?
            - Lenguaje que se va a usar.
            - ¿Cómo se organiza la jerarquía de clases?
            - Poliformismo tradicional o plantillas y política de diseño.
            - Cómo son referenciados los objetos, punteros clásicos, punteros inteligentes...
            - Cómo se identifican los objetos de forma única, por dirección de memoria, por nombre, por identificador global único...
            - Cómo se maneja el tiempo de vida de los objetos.
            - Cómo se simulan los estados de los objetos en el tiempo.
    - Sistema de eventos: Los objetos necesitan comunicarse unos con otros y puede llevarse a cabo de varias formas. Esas formas pueden ser llamando a una función del objeto destino, o hacerlo con un sistema de eventos como se suele usar en las interfaces gráficas típicas.
    - Sistema de scripting. Muchos motores usan un lenguaje de script para hacer reglas de juego específicas de forma más fácil y rápida. De otra forma, se tendría que recompilar y reenlazar el ejecutable del juego cada vez que se hiciera un cambio a la lógica o estructuras de datos utilizadas en el motor. El lenguaje permite cambios simplemente recargando el script, algunos permitiéndolo hacer incluso en tiempo de ejecución.

    - Fundamentos de inteligencia artificial: el sistema de inteligencia artificial para los enemigos y otros objetos del juego, como puede ser la búsqueda de rutas por donde moverse sin chocar.
- **Subsistemas específicos del juego** Si se pudiera trazar una línea de división entre el juego y el motor, se establecería entre los subsistemas específicos del juego y los fundamentos del *gameplay*. Estos subsistemas incluyen, pero no se limitan, a las mecánicas de juego del jugador, a los sistemas de la cámara durante el juego, inteligencia artificial más específica, armas, vehículos...

2.3.2 La suite de herramientas

Los juegos son por naturaleza aplicaciones multimedia y los datos que recibe un juego viene en multitud de formas, ya sea datos de modelos 3D, bitmaps de texturas, animaciones, ficheros de audio... los cuales deben ser creados y manipulados por artistas. Las herramientas que suelen utilizar son llamados aplicaciones de creación de contenido digital (*digital content creation* - DCC). Ejemplos son Autodesk's Maya y 3ds Max, Photoshop, SoundForge... Muchos *engines* vienen con un editor propio para la creación de mundos. Todavía hoy en día usan herramientas existentes para crear los mundos.

El resultado de esas herramientas no suele ser directamente utilizables por el juego básicamente por dos razones, los datos generados son mucho más complejos de lo que necesita el *engine*, por ejemplo maya guarda un gráfico directo acíclico de nodos de escena con una compleja red de interconexiones, historial, posiciones, orientación y escalado en una jerarquía, junto con la descomposición en translaciones, rotaciones y otros. El *engine* sólo necesita una fracción pequeña de esos datos para renderizar. El otro motivo es que el formato en el que guardan suele ser muy lento a la hora de leer y a veces es un formato propietario. Por ello, es exportado a formatos estandarizados o propios y procesados al gusto para el *engine*.

La arquitectura de una suite de herramientas de un motor de videojuego se puede diseñar de varias formas. Algunas herramientas son piezas de software independientes, otros se construyen encima de capas del motor y otros están incluidos dentro del propio motor.

2.4 Gestión de la memoria

Hay dos tipos memorias, la memoria RAM, principal memoria volátil y la RAM de Video (VRAM) normalmente mucho más pequeña y específica para guardar objetos que serán usados por la tarjeta gráfica.

Hay un tercer tipo de memoria, la memoria virtual. Cuando nos quedamos sin espacio en la RAM, bloques de memoria sin usar de la RAM se guardan en disco duro hasta que se vuelven a necesitar. Tiene una ventaja y una desventaja. La ventaja es tener mucha más memoria, la desventaja es que cualquier operación de disco duro es muy costosa. No se aconseja fiarse de la memoria virtual ya que las consolas normalmente no disponen de ella.

Muchos juegos extienden el sistema de gestión de memoria dado para conseguir más rendimiento, eficiencia y mejorar la depuración, ya que la asignación de memoria dinámica puede ser costoso. La gestión por defecto de C está designado para un gran número de escenarios posibles, pero falla en los juegos donde la asignación y deasignación puede ser muy agresiva.

Una forma de optimizar puede ser usando patrones, pudiendo ser más efectivo una asignación personalizada que una general. Algunos ejemplos son:

- Asignaciones basadas en pilas: Cuando un nuevo nivel es cargado, se asigna memoria y ya no hay más asignaciones para ese nivel. Al final del nivel la memoria es liberada. Por ello estructuras de datos de pila son adecuados. Se mantiene una referencia al top de la pila y todo por debajo de ella se considera usada. El problema es que la memoria no puede ser liberada en cualquier orden.

- Asignación de pila de doble final. Donde se usa dos pilas, una que crece hacia arriba, y otra que crece hacia abajo. Es normalmente más eficiente. Un ejemplo es el juego de Hydro Thunder (un arcade) en el que el stack inferior lo usaba para cargar y liberar niveles y el superior para memoria temporal que era asignado y liberado en cada frame. Esta forma permitía al juego no sufrir problemas de desfragmentación.

- Asignaciones *pool*. Es frecuente asignar pequeños trozos de memoria del mismo tamaño, por ejemplo, matrices, listas, iteradores... A menudo este enfoque es la elección perfecta. Reserva un largo espacio de memoria múltiplo del tamaño de los elementos que aloja y cada elemento de la memoria es añadido a una lista enlazada de elementos libres. Cada vez que se pide un espacio de memoria, simplemente usamos el siguiente elemento libre en la lista. Cuando es liberado, se coloca de nuevo esa dirección de memoria en la lista. El coste es de O(1).

- Asignaciones de memoria de simple frame y doble buffer. Todos los motores de juegos guardan algún dato temporal en el bucle principal del juego y puede ser descartado o usado en la siguiente iteracción.

    - *Single-frame*: se reserva memoria y se maneja de la misma forma que la pila vista anteriormente. En cada iteracción, el puntero del *stack* se coloca al fondo de la pila y las asignaciones que se realizan durante el ciclo crecen del puntero hacia arriba del bloque (es decir, cada iteracción, el puntero se reinicia al principio y se va usando esa memoria. En el siguiente ciclo, al volverse a colocar el puntero, es como borrar toda esa memoria y se vuelve a empezar) Todo ello sin liberar de forma expresa la memoria. El único problema es que se requiere gran pericia del programador, cualquier cosa que se use, sólo valdrá en ese frame y nunca debe guardar un puntero para el siguiente frame.

- *Double Buffered allocators*. Permite usar un bloque de memoria del frame i en el frame i+1. Para ello se crean dos singles-frames del mismo tamaño y se intercambian en cada frame. Muy útil para cachear los resultados de procesos asíncronos en consolas *multicore* como la playstation 3 o la xbox 360. El hilo corre y produce resultados después de que haya terminado el frame i, de forma que estando en el frame i+1 donde los *buffers* han sido intercambiados, ni el hilo ni el resultado resultan sobrescritos ya que se encuentra en el *buffer* inactivo.

La fragmentación de la memoria, es otro de los problemas con la asignación de memoria. No es un gran problema en sistemas que soportan memoria virtual, los cuales mapean los trozos discontinuos de memoria en una memoria única y contigua. Las páginas de memoria pueden ser pasadas también a disco en caso de ser necesario, como se ha comentado antes. Sin embargo, en varias consolas modernas que soportan esto, los motores no los utilizan debido a la sobrecarga de rendimiento que lleva inherente.

En el caso de usar los patrones de pila y *pool* para la memoria, son transparentes a esos problemas, ya que la pila es siempre contigua y la liberación es siempre en el mismo orden de uso, mientras que en pool, aunque no tenga ese orden, todos los trozos son del mismo tamaño.

En el momento que los objetos a guardar sean de distinto tamaño o asignados/liberados en distinto orden, no se puede usar esa técnica. Por lo que hay otros medios como desfragmentar periódicamente, algo relativamente fácil pero que busca complicaciones como la invalidación de punteros y el coste de la desfragmentación. Otras formas de llevarlo a cabo son desfragmentaciones parciales, desfragmentando una parte en cada frame en vez de todo.

Enfocándonos en la caché, cada acceso a la RAM usa la caché de la CPU. Si los datos ya están en la caché, entonces el acceso es mucho más rápido. Por lo que se recomienda juntar los datos relacionados. De forma similar, la alineación de la memoria también cuenta para el rendimiento.

Cada vez que la caché es usada ineficientemente, el rendimiento puede degradarse varios órdenes de magnitud. Se le llama normalmente tirar caché y es de lo peor que puede pasar.

Una forma de bajar el ratio de fallos de caché es organizar los datos en bloques contiguos lo más pequeño posible y acceder a ello secuencialmente. Si el juego está tirando caché, se puede resolver organizando los datos, pero lo más probable es que se necesite reducir el tamaño de los datos.

Para el caché de las instrucciones, sigue el mismo principio, pero se requiere otro enfoque para implementarlo, conociendo las reglas del compilador C/C++, por ejemplo:

- El código de una función es casi siempre contigua
- Las funciones están ordenadas en memoria en el mismo orden en el que aparecen en el fichero .cpp

- Las unidades de translación .obj también quedan juntas en memoria siempre. Es decir, el enlazador nunca parte los objetos para meter entre medias otro.

Por ello, se aconseja los métodos de alto rendimiento lo más pequeños posibles, evitar llamar funciones en mitad de secciones de alto rendimiento, juntar las funciones llamadas lo más cerca posible de la llamada y nunca en una unidad distinta. Y usar las funciones *inline* con cabeza.

En el caso de la VRAM, como no hay comunicación directa entre el disco y la VRAM, cada vez que se necesita una nueva textura hay que parar todo para hacer la comunicación, por lo que lo más inteligente es mandar las cosas en lotes. (1) (9)

2.5 Guardar y cargar opciones

Ya sea por el juego en sí o por la configuración tales como el video, o el modo de depuración o similares, los motores tienen varias formas de guardar y recuperar opciones variadas:

- Ficheros de configuración. Cuyo formato puede ser muy variado pero normalmente muy simple, como txt, o xml también es muy utilizado. Consiste normalmente en listas de llave-valor. Ej: [section] key1=value key2=value

- Ficheros binarios comprimidos. El formato preferido para las tarjetas de memoria de la consola ya que suele ser un espacio muy limitado.

- El registro de Windows: Ofrece una base de datos de opciones global y se guarda como un árbol, en el que los nodos interiores conocidos como registros actúan como carpetas, y los nodos hojas guardan las opciones individuales como pares llave-valor. Cualquier aplicación/juego puede reservarse un subárbol entero para su uso exclusivo.

- Opciones por comandos de línea.

- Variables de entorno.

- Perfiles de usuarios online: Con el avance de comunidades online de jugadores como Xbox Live, cada usuario puede crear un perfil con configuraciones, trofeos y otros datos, y guardarlo de forma online en un servidor central, pudiendo acceder a él en varios sitios con conexión online.

En algunos motores, se diferencia opciones globales de opciones por usuario.

2.6 Los recursos y el sistema de ficheros

Un juego es un programa multimedia y se maneja con ficheros de audio, imagen, video... por lo que un *engine* tiene que ser capaz de guardar y manejar un largo rango de diferentes tipos de media, texturas, bitmask, 3D,audio, animación, clips, datos de colisión y física... Y como la memoria es un bien escaso, el *engine* tiene que asegurarse de que sólo se carga una vez.

Un motor de videojuego puede ser escrito para usar las funciones de entrada/salida de ficheros que incluye la librería estándar de C, o incluso usar la API nativa del sistema operativo. Sin embargo la mayoría prefiere implementar una librería personalizada *wrapeando* dichas funciones.

Hay al menos tres ventajas y es que pueden garantizar el mismo comportamiento entre plataformas, incluso en las partes donde lo nativo es inconsistente, tener implementadas solo las funciones necesarias, y por ello menos esfuerzo de mantenimiento. Y por último se puede proveer de funcionalidades avanzadas como puede ser leer fichero a través de la red.

Tanto la librería estándar de C y la librería I/O son síncronos. Cuando nos referimos al I/O asíncrono, estamos hablando de *streaming*, la acción de ir cargando datos en segundo plano mientras el programa principal continúa corriendo. Muchos juegos evitan las escenas de cargas leyendo los datos de los próximos niveles en *streaming*. Las texturas y los sonidos son probablemente los más usuales, pero se puede con cualquier tipo de dato, incluyendo geometrías, animaciones y otros.

Todo gestor de recursos se compone de dos componentes diferenciados pero integrados. Un componente maneja las herramientas del *engine* que transforma los recursos en una forma binaria preparada para su uso en el *engine*. El otro componente maneja los recursos en tiempo de ejecución, asegurándose de cargarlos en memoria cuando se necesitan y liberarlos. En algunos *engines* está bien centralizado mientras que en otros no existe como un módulo per-se. Los requisitos básicos que se piden al componente de tiempo de ejecución son:

- Se asegura de que sólo haya una copia de cada recurso en memoria en cada momento.

- Controla el ciclo de vida de cada recurso cargado y libera los recursos que ya no se necesita más.

- Maneja la carga de recursos compuestos, que son aquellos recursos que se componen a su vez de otros recursos como en los modelos 3D que se componen de uno o más texturas, de un esqueleto, animaciones del esqueleto, materiales...

- Mantiene la integridad de las referencias. Incluye las referencias internas dentro de un mismo recurso y las referencias externas que son entre recursos distintos.

- Controla el uso de memoria en los recursos cargados y se asegura de que están en su sitio correspondiente.

- Permite personalizar el tratamiento del recurso a realizar cuando se carga en memoria.

- Normalmente ofrece una interfaz unificada para manejar un amplio rango de tipos de recursos. De forma ideal el gestor de recursos es fácilmente extensible para poder manejar nuevos tipos de recursos.

- Maneja el *streaming* de los recursos si el motor soporta esta característica.

2.6.1 Organización de recursos

Los recursos suelen estar organizados en un árbol de directorios a conveniencia de la gente que crea los contenidos. Otros *engines* meten los recursos en un solo fichero, como un zip.

Los tres principales costes en los recursos son los tiempos de búsqueda (cabezal del disco), el tiempo para abrir cada archivo individual y el tiempo de leer los datos a memoria. Los dos primeros pueden ser no triviales y cuando se usa un único fichero grande, se mejora esos tiempos. Las ventajas concretas de usar los zip son:

- Es de formato abierto, las librerías zlib y zziplib usadas para leer y escribir en ZIP están disponibles de forma libre e incluso el SDK de zlib es totalmente libre.

- Los ficheros virtuales de dentro del zip tienen su ruta relativa, por lo que se puede tratar el zip como un subsistema de archivos para la mayoría de propósitos.

- Los zip pueden ser comprimidos y ocupar menos espacio. Para cada fichero, el sistema de ficheros del SO, mantiene algo de información. De igual forma, si no encajan en los bloques (*cluster*) del disco duro, queda algo de espacio libre no aprovechable, ya que normalmente los sistemas de ficheros sólo permiten un fichero por bloque. Un cálculo de ejemplo:

  > 500 ficheros de sonidos de efectos, cada uno de medio segundo de duración y a 44KHz mono.
  >
  > 0.5 segundos * 44KHz mono = 22,000 bytes
  >
  > 32,768 bytes mínimo de tamaño de bloque – 22,000 bytes de cada fichero = 10,768 bytes desperdiciados por fichero
  >
  > 10,768 bytes desperdiciados * 500 ficheros = 5.13MB espacio total desperdiciado.

  El problema es que puede no compensar el tiempo necesario para descomprimir una vez cargado. Puede resultar muy útil cuando estamos leyendo los datos de medios con un DVD-ROM o similar ya que lee menos datos.

- Al ser sólo un fichero en vez de varios, el tiempo de carga es menor. Las lecturas de disco duro son lentas ya que tardan en buscar el lugar, y una vez encontrados, la lectura/escritura también tarda. Si hablamos de medios como los discos, es aún peor, ya que su organización es en espiral en vez de sectores, por lo que la búsqueda de los datos ha de recorrer la espiral desde el interior al exterior y el láser ha de recorrer trasversalmente esa espiral a velocidad constante, por lo que el disco ha de ir ajustando su velocidad a medida que busca. Si la localización es errónea, ha de volver a empezar.
  La única cosa más lenta es tener a alguien tipeando los datos desde el teclado. Por ello es buena idea agrupar los recursos en un solo bloque de datos para leer de golpe todo junto.

- Los zip son archivos modulares, los recursos pueden ser agrupados y manejados como una unidad. Un ejemplo de utilidad es la localización, en el que todos los productos que necesitan ser localizados se pueden guardar en diferentes versiones de zip, según la zona o lenguaje. El motor carga el zip adecuado en cada momento.

Debido a su lentitud, se debería tratar los datos de forma similar a un adicto a las compras. Mantenerle lejos de las tiendas hasta que se tiene una lista larga de cosas necesarias a comprar. Cuando el juego precisa de recursos, espera a tener varios para leerlos todos de golpe.

Por ello es buena idea agrupar los recursos. El problema es que agrupando los ficheros por niveles o similar, nos podemos encontrar con que hay recursos duplicados en esas agrupaciones.

Un equilibrio entre agrupaciones y duplicados es tener los datos comunes en un bloque, y los datos específicos para el nivel o pantalla en sus propios bloques. Por lo que sólo sería necesario buscar en disco duro un par de veces al principio, y las siguientes veces, al tener lo común ya cargado, sólo se busca una vez para el nivel concreto.

2.6.2 Identificación, registro y tiempo de vida de recursos

Identificadores únicos. Los objetos del juego necesitan identificadores y los *string* son de forma natural una buena opción. Sin embargo C/C++ no compara tan bien y rápido, por lo que una solución es *hashear* los *string* y tener la velocidad de comparación de un *int*. En Naughty Dog usa una variante del algoritmo de CRC-32 para los hash y en dos años de desarrollo de Uncharted: Drake's Fortune no han encontrado una sola colisión entre los *strings*. (10)

Cada recurso debe ser identificado (GUID - *globally unique identifier*) y una opción muy común es por la ruta del recurso (como *string* o como *hash*) ya que es claro y garantía de ser único ya que el SO se encarga de que no pueda existir dos ficheros con la misma ruta. Pero no es la única opción, algunos motores usan un GUID menos intuitivo, como un código hash de 128 bits, dado por herramientas que garantizan unicidad. Puede pasar porque en algunos motores la ruta no es una forma viable, como por ejemplo en Unreal, donde guarda los recursos en ficheros

largos denominados paquetes. Para evitarlo, Unreal organiza los recursos dentro del paquete en una jerarquía como las rutas del sistema de ficheros, de forma que junto al nombre único del paquete, el GUID viene dado por el nombre del paquete y su ruta dentro del paquete.

Para que el gestor se asegure de que sólo se carga una vez, se mantiene un registro de las cosas que se han cargado. Una forma sencilla de llevarlo a cabo es con un diccionario (o *hashmap*) usando de llave el GUID y devolviendo el puntero del recurso cuando es solicitado y ya está cargado, o si no está, devolver un error y/o cargarlo.

El problema de cargarlo si no se encuentra, tiene un problema, y es que es muy lento cargarlo y seguramente haya también que procesar algo una vez cargado. Si la petición se hace durante el juego, puede afectar a los fps o incluso congelar la pantalla unos segundos. Por ese motivo, los motores suelen escoger entre dos alternativas:

- Se desactiva por completo la carga durante el juego. Los recursos son cargados en masa justo antes de jugar (por ejemplo, las pantallas de *loading*... y similares)

- La carga se hace de forma asíncrona. Mientras el jugador está en el nivel A, se van cargando en segundo plano los elementos del nivel B. Suele ser preferible que dar una pantalla de carga, pero es considerablemente más difícil de implementar.

También, en el registro, se ha de tener en cuenta el tiempo de vida de los recursos. Se define como el periodo de tiempo entre su primera carga y cuando la memoria reclama ese espacio del recurso para otras cosas. Es otro de los trabajos del gestor de recursos, ya sea haciéndolo de forma automática o dando el API necesario para hacerlo manualmente. Hay requerimientos distintos:

- Algunos recursos se cargan en cuanto arranca el juego y deben permanecer en memoria todo el tiempo del juego. Es decir, tiempo de vida infinito y son llamados a veces recursos *load-and-stay-resident* (LSR). Por ejemplo los recursos referentes al protagonista y al HUD. Todos los recursos que sean visibles o audibles en todo el juego deben ser tratados como LSR.

- Algunos recursos tienen tiempo de vida coincidente con el nivel en el que está

- Otros recursos tienen menor tiempo de vida como las animaciones o los audios que forman una escena en el juego. Tienen que ser cargados por adelantado y se eliminan una vez han cumplido su función.

- Recursos como la música de fondo, de ambiente o de efectos, o los videos de pantalla completa, son cargados y reproducidos en *streaming*. El ciclo de vida es más complejo de definir ya que cada byte solo persiste por unas fracciones de segundo en la memoria, aunque la pieza sea de mucho más tiempo.

Decidir cuándo se carga un recurso es relativamente fácil, pero no tanto descargar y menos cuando el recurso es compartido entre varios niveles. No se desea descargar un recurso en el nivel A cuando en el B va a volver a necesitarlo. Una posible solución es contar las referencias a los recursos, cuando un nivel va a cargarse, la lista de recursos necesarios se pasa y su contador se incrementa (antes de cargarlos), y luego, se decrementa todos los recursos que el nivel actual usa y va a liberar. De esa forma, sólo libera los recursos con conteo 0. Y de la misma forma, carga todos aquellos que pasaron de conteo 0 a uno o más.

2.6.3 Formatos de ficheros

Los ficheros, según el tipo de recurso que sea, tienen un formato potencialmente distinto de otros, por ejemplo png en imágenes, mientras que en audio es wav, los bitmaps en bmp... A veces un solo formato puede servir para recursos de distintos tipos. Por ejemplo, el SDK de Granny por las herramientas de Rad Game (11) implementa un formato de fichero flexible y abierto que puede usarse para los datos d ella malla 3D, el esqueleto y las animaciones del esqueleto. De hecho, el mismo formato puede ser usado para guardar virtualmente cualquier tipo de dato imaginable.

Muchos *engines* lanzan su propio formato de fichero por varias razones. Puede ser necesario si no existe un formato estandarizado que guarde la información que el *engine* necesite. También puede ser debido a que los *engines* procuran procesar poco para minimizar los tiempos de cargas de los recursos por lo que los formatos que lanzan ya tienen eso en mente, sólo lo esencial y de forma que se realice rápido sin tener que calcular mucho.

La seguridad es otro punto para el formato propietario, si se usa un formato propietario (con una organización propietaria del contenido del fichero) se evita manos indeseadas en el contenido.

2.6.4 Referencias cruzadas

Estos GUIDs permiten también usarlos como referencias cruzadas en vez de los punteros. Para ello el gestor de recursos mantiene una tabla de búsqueda de recursos global, donde establece el puntero de cada recurso cargado (lo que implica que los GUID sean globalmente únicos o no funcionará). Luego de cargar los objetos, se puede hacer una pasada por todos los objetos para convertir todas sus referencias cruzadas en punteros mirando en la tala de búsqueda.

2.7 El tiempo

Los juegos son simulaciones en tiempo real, dinámicos e interactivos. El tiempo juega un rol importante y hay distintos tipos de tiempos a manejar en el motor; tiempo real, tiempo local de una animación, tiempo del juego, ciclos del cpu...

En un programa normal, la mayoría de la pantalla es estática y sólo una parte cambia, por lo que las interfaces de usuario usan una técnica conocida como

rectángulo de invalidación, en el que sólo en los recuadros donde haya cambiado algo se redibujan.

Es una técnica muy similar usada en viejos juegos en 2D para no tener que dibujar tanto. Pero en juegos 3D reales, se implementa de forma distinta, ya que en cuanto la cámara se mueve, absolutamente toda la pantalla cambia.

Para la ilusión de movimiento, se presenta una serie de imágenes en rápida sucesión, lo que implica un bucle. El bucle principal de renderizado tiene una estructura como la siguiente:

while (!quit) {

actualizar la cámara basado en inputs o por una ruta predefinida

actualiza los elementos de la escena (posiciones, orientaciones...)

renderiza la escena (dibuja un frame en un buffer)

intercambia los buffers de la pantalla y se actualiza la pantalla

}

De igual forma, un juego está compuesto por subsistemas interactuando; renderizado, animación, colisión, audio, entrada de usuario, redes... La mayoría de subsistemas de los motores precisan de "servicio" periódico mientras el juego corre. Sin embargo el ratio de actualización varía en cada subsistema, por ejemplo la parte de renderizado necesitaría un ratio de 30 a 60hz, en sincronización con el refresco de pantalla, mientras que una simulación dinámica puede necesitar 120, otros, como la IA podría ser sólo una o dos veces por segundo y sin necesidad de sincronizarse con la renderización.

Hay varias arquitecturas para implementar esa actualización periódica. La más sencilla es usando un bucle para actualizar todo, normalmente llamado el bucle de juego. En un ejemplo de juego sencillo, como puede ser el remake realizado de tetris, en primer lugar inicializa algunas cosas como puede ser los subsistemas gráficos, de audio y otros, y seguidamente se entra en el bucle de juego, donde sólo se sale cuando el juego acaba o es interrumpido. En el bucle se lee la entrada de usuario, se actualizan los objetos del juego (movimientos), se comprueban colisiones, y finalmente, se pinta el frame con todo actualizado. Bucles más avanzados puede implicar más bucles anidados o varios bucles seguidos.

El problema del bucle sencillo es que tiende a ser inflexible y no permite actualizar algo en otra frecuencia. Se puede usar hilos para dividir el bucle de juego y que cada actualización corra por su parte de forma concurrente al resto. Al renderizar, la CPU se tira mucho tiempo esperando a la tarjeta gráfica a que procese lo que le ha enviado y con los hilos el problema se resuelve.

Tampoco es buena idea poner todo en su propio hilo ya que hay que comunicarlos entre ellos y se complica además de terminar siendo ineficiente. Una técnica híbrida es buena solución, permitiendo a los subsistemas desacoplados de otros, dando la sensación de multitarea y sin los problemas que llevan los multihilos. La técnica es llamada multitarea cooperativa, en el que va dando tiempo de proceso a cada uno en *round-robin*.

Un par de arquitecturas usadas para implementar los bucles de juego son los siguientes:

- Framework dirigido por retrollamadas (*Callback-driven framework*): En un motor basado en *framework*, el bucle de juego está escrito pero mayormente vacío y el programador puede escribir llamadas para rellenar. Un ejemplo son los métodos *frameStarted* y *frameEnded* del *frameListener* de Ogre3D que permite ser sobrescritos para meter funciones propias.

- Actualizaciones basadas en eventos: en juegos, un evento es cualquier cambio interesante en el estado del juego o de su entorno. Puede ser pulsaciones de botón, un enemigo disparando, una explosión... La mayoría de motores incluyen un sistema de eventos que permite a los subsistemas subscribirse a tipos particulares de eventos que le interesan y responder ante esos eventos. Suele ser bastante similar a los sistemas de eventos como el de java. Algunos motores establecen su sistema de eventos para implementar el servicio periódico de algunos o todos los subsistemas. Para ello, el sistema de eventos debe permitir a los eventos ser publicado en el futuro (es decir, ser apilados para ser entregado más tarde).

2.7.1 Frame rate y tiempo delta

El ratio de frames de un juego describe cómo de rápido va la secuencia de imágenes en la pantalla. Los hercios (Hz) define el número de ciclos por segundo, que puede ser usado para describir el ratio de cualquier proceso periódico. En los juegos y filmes el ratio de frames es típicamente medido en frames por segundo (FPS) que es lo mismo que los herzios para todas las intenciones y propósitos. Los videos tradicionalmente corren a 24fps y los juegos en Norte América y Japón a 30 o 60fps que es el refresco natural de la televisión NTSC, mientras que en europa y la mayoría del resto del mundo es a 50fps, refresco natural de la televisión PAL o SECAM.

El tiempo que pasa entre frames se conoce como tiempo de frame o tiempo delta. El tiempo delta es un término muy común debido a que la duración de ese tiempo suele representarse con el símbolo matemático de Δt (ignorando tecnicismos, puesto que ese símbolo sería realmente el periodo de frame ya que es el inverso de la frecuencia de frames pero los programadores rara vez usan el término periodo en este contexto).

Si un juego es renderizado a 30fps, el tiempo delta es de 1/30 segundos (33,3ms) y análogamente para los 60fps, sería 1/60 segundos. Los milisegundos son una unidad ampliamente utilizada en los juegos.

Los fps afectan a la velocidad del juego. Poniendo de ejemplo que en un juego de naves, la nave avanza 40 metros por segundo (o si hablamos en 2D, 40 píxels por segundo) una forma simple de conseguir esto es establecer una velocidad a la nave y multiplicarla por la duración del frame para hallar el incremento del movimiento y haciéndola independiente de los fps Hablando técnicamente, es una simple forma de integración numérica llamada método explícito de Euler.

Ese método funciona bien cuando la velocidad es constante. Para manejar velocidades variables se necesita métodos algo más complejos. Sin embargo, todas las técnicas de integraciones numéricas usan el tiempo delta, lo que lo hace dependiente a esa duración.

Antiguamente no se hacía caso al tiempo delta y directamente movían los objetos en términos de metros (o pixels) por frame. El problema es que con cpu más rápidas se veía como el juego íba más rápido al ser directamente dependiente de la velocidad del cpu.

Posteriormente, para evitar esa dependencia, leían el registro temporal de la cpu al inicio y al final del frame y con ello obtenían el tiempo que había pasado, es decir, el tiempo delta. Con ese tiempo, se le podía pasar a los subsistemas y/o guardarlo y que reaccionaran de forma acorde. Muchos motores, sino la mayoría, usan esta técnica. El problema es que usaban ese tiempo para estimar cuánto duraría el frame siguiente. Aparte de lo poco preciso que es, si un frame se retrasaba, podía provocar un círculo vicioso, en el que para compensar el tiempo perdido, ejecuta la física dos veces en el siguiente frame, lo que a su vez retrasa dicho frame y así continuamente.

Otro enfoque es teniendo en cuenta la coherencia del juego de frame a frame. Si se está en una escena costosa para un solo frame por tener muchos objetos costosos, es probable que la siguiente escena también sea igual de costosa por estar en el mismo sitio o similar. Un buen método es promediar los tiempos del frame sobre un pequeño número de frames para calcular lo que tardaría el siguiente frame. Ello permitiría al juego ajustar el ratio de frames bajando la calidad de los gráficos en esa escena por ejemplo. A mayor intervalo de frames a promediar, menor respuesta del juego para adaptarse.

Un último enfoque, es garantizar la duración del frame en vez de sugerir el tiempo del siguiente frame. Se realiza el frame y si al final, el tiempo del frame no es el mínimo que ha de tener para cumplir los fps, se le hace esperar. Si supera mucho el tiempo se debería esperar al frame siguiente. Este enfoque se le denomina gobierno por ratio de frames (*frame-rate governing*). Obviamente este enfoque sólo funciona si el ratio de frames es cercano al ratio del juego. Si el juego fluctúa mucho entre 30 y 15fps por frecuentes frames lentos, la calidad del juego baja mucho. Por ello es todavía buena idea diseñar el motor para ser capaz de enfrentar duraciones de frames arbitrarios.

Mantener el ratio de frames constante es importante ya que varios subsistemas trabajan mejor con ratios constantes además de verse mejor y evitar el *tearing* (11) que puede ocurrir cuando actualizamos el buffer del video a un ratio distinto al de refresco de la pantalla. Ese efecto ocurre cuando mandamos la nueva imagen cuando el refresco está a la mitad, por lo que una mitad tiene la imagen anterior y la otra mitad la nueva. Para ello varios motores esperan a dicho refresco para actualizar. Es otra forma de gobierno por ratio de frames.

2.7.2 Timelines

En un juego es muy útil pensar en términos de línea de tiempo abstracto. No necesitamos limitarnos al tiempo real de forma exclusiva y usar tantas líneas de tiempo como queramos. Podemos entonces definir la línea de tiempo del juego como una línea temporal independiente del tiempo real.

En circunstancias normales, el tiempo del juego coincide con el tiempo real. Si pausamos el juego, simplemente podemos dejar de actualizar temporalmente la línea temporal del juego. En caso de que queramos ir en *slow motion*, podemos actualizar el reloj del juego más despacio que el reloj real y análogamente para ir más deprisa.

El manejo del reloj del juego es también una herramienta útil de depuración, sobre todo si el motor de renderizado y la cámara sigue en marcha mientras el juego está pausado, cosa que ocurre si ambos están gobernados por un reloj distinto. Lo único a tener en cuenta, es que aunque el juego esté pausado, el bucle de juego continúa (sólo el reloj se ha parado) y no es lo mismo ir paso a paso depurando que simplemente añadir más tiempo a cada frame para observar detalles.

Hay dos tipos de líneas temporales, globales y locales. Por ejemplo un video o audio tiene su propia línea temporal, una línea temporal local cuyo origen es el comienzo del video/audio. Eso permite hacer muchas cosas como puede ser reproducir un video a la mitad de la velocidad a la que fue grabado, tan sólo aplicando un factor de tiempo y con ello meterlo en la línea temporal global. Incluso reproducirlo al revés aplicando un factor negativo.

2.8 Animaciones y gráficos

2.8.1 Gráficos

El sistema de renderizado del *engine* dibuja todo en pantalla lo más rápido que puede y es seguramente lo que más CPU pide de todos los componentes del juego, además de tener que lidiar con las diferentes CPUs y GPUs y hasta con las diferentes configuraciones de hardware y sistemas operativos.

Considerad como ejemplo un simulador de vuelo. Cuando el avión está en el suelo, se pueden ver algunos edificios, aviones, pistas… y alguna escena a lo lejos como una montaña. Todo con buen detalle.

Una vez el avión está en el aire, las cosas cambian, el área de visión es varias veces mayor y se ve muchas más cosas. Si se intenta dibujar todo ello sin más, enseguida se verá que es imposible a costa de cargarse los fps del juego.

La forma de arreglar el problema es usar diferentes niveles de detalle para dibujar, dependiendo de la distancia a la que está de la cámara. Lo difícil es conseguir una buena transición entre dichos niveles de detalles.

El otro principal problema en el dibujado, es evitar el sobredibujado. No tiene sentido dibujar objetos para que luego sean sobrescritos por otro objeto que estaba delante y tapa el primer objeto. (9)

Los triángulos de dibujo más baratos son aquellos que nunca se dibujan. Es importante filtrar para saber que objetos se renderizan o no. El proceso de construir la lista de instancias visibles se conoce como la determinación de visibilidad. Las principales técnicas existentes para ello son los siguientes:

- Tronco *culling*: Todos los objetos que no están dentro del tronco (13) son excluidos de la lista. Dado el objeto, se comprueba con simples test entre el volumen del objeto y los 6 planos del tronco. El volumen es normalmente una esfera, por lo que su cálculo es muy fácil.

  Una estructura de grafo de escena puede ayudar a la optimización del tronco *culling* permitiendo ignorar objetos que siquiera están cerca del tronco.

- Oclusión *culling*: Incluso aunque los objetos estén dentro del tronco, pueden taparse unos a otros. Quitar de la lista aquellos objetos ocultos por otros se denomina oclusión *culling*. En entornos llenos de gente a nivel del suelo es una buena oportunidad para hacer buen uso del método. En casos con menos posibilidades de tener muchos casos de objetos tapados, el método puede ser contraproducente.

- Sets potencialmente visibles: En entornos muy grandes, la occlusion *culling* se puede hacer precalculando un set potencialmente visible (PVS – *potentially visible set*). La diferencia es que esa técnica en vez de quitar de una lista de objetos, pone en una lista aquellos potencialmente visibles.

- Portales: Otra forma de determinar que partes de la escena son visibles, es usar portales. El mundo se divide en regiones semi-cerradas que están conectadas unas a otras por agujeros, como ventanas y puertas. Dichos agujeros se llaman portales y se suelen representar por polígonos que describen sus límites.

  Para renderizar una escena con portales, se empieza renderizando la región donde está la cámara y luego, por cada portal, se extiende un volumen parecido al tronco, que extienden desde la cámara y a través de los lados del portal.

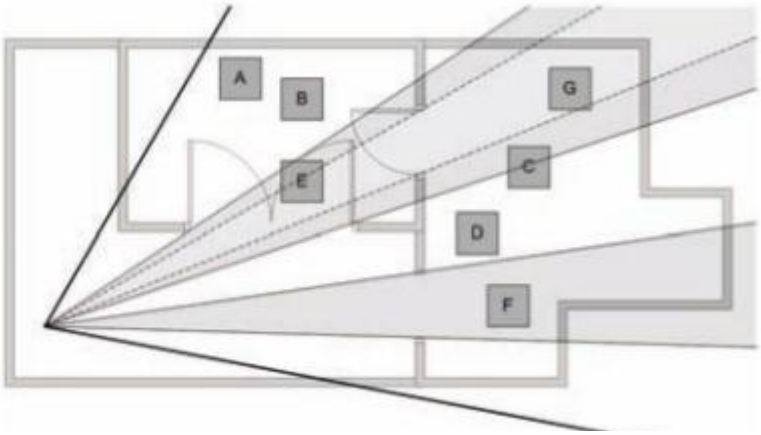


Figura 2-1. Ejemplo de portales. Fuente: Game Engine Architecture (1)

- Los anti-portales (volúmenes de oclusión) siguen un concepto similar a los portales. Se pueden usar volúmenes piramidales que van desde la silueta del objeto y se extienden al lado contrario a la cámara, formando un volumen en el que cualquier objeto dentro de él no se ve. La imagen ilustra mejor dicho concepto.

  Los portales son buenos para entornos cerrados como habitaciones, y los antiportales resultan mejor para entornos abiertos y grandes.

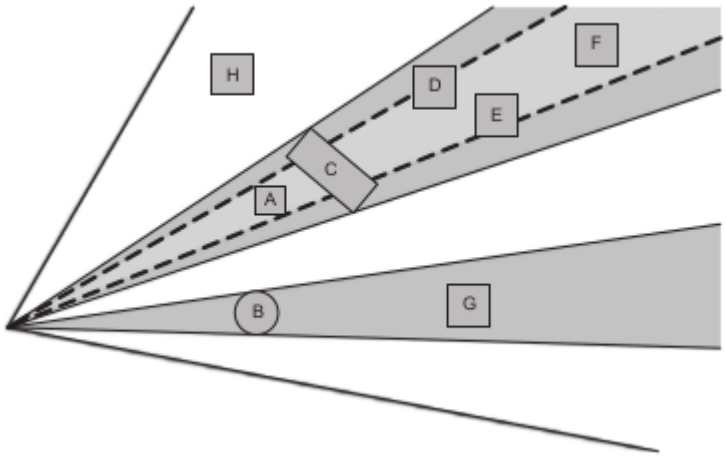


Figura 2-2. Ejemplo de antiportal. Fuente: Game Engine Architecture (1)

2.8.1.1 Grafo de escena

Como los mundos pueden ser muy grandes, muchos de los objetos no se van a ver dentro de la cámara y hacer un tronco puede ser una pérdida de tiempo. Por ello es bueno usar una estructura de datos que maneja la geometría de la escena para descartar rápidamente grandes partes del mundo sin pasar por el tronco.

Idealmente, dichas estructuras ayudan a ordenar la geometría de la escena. Dichas estructuras se llaman grafo de escena, en referencia a los grafos que suelen usar herramientas como Maya. Sin embargo no tiene porqué ser un grafo, normalmente es un árbol.

La principal idea es partir el espaciado de forma que ayude a descartar regiones. Ejemplos incluyen *quadtrees*, k, árboles BSP, k-árboles y técnicas de *hashing* espacial. Estructuras de datos que también se usan en las colisiones como se verá más adelante.

2.8.2 Animaciones

Dentro de la historia de la animación del personaje, sacamos las tres técnicas más comunes:

- Animación *Cel*: El precursor de toda animación conocido por animación tradicional o animación a mano. La ilusión de movimiento se hacía con rápidas secuencias de imágenes (frames). La animación *cel* es un tipo específico de animación tradicional. Un *cel* es una lámina transparente de plástico que se superponía a otras formando el conjunto de la imagen, y la secuencia de *cels* producía la animación de ese personaje sin cambiar el fondo. El equivalente electrónico es la animación de *sprites*. Muchas veces, la secuencia de frames/*sprites* se diseñaba de forma que la animación era fluida incluso si se repetía indefinidamente. Esto es conocido como animación cíclica. Los personajes suelen tener varias animaciones cíclicas como es caminar, correr, incluso estar "quieto" y otros.

- Animación jerárquica rígida. Con los juegos 3D, la animación de *sprites* empieza a caer en desuso. En Doom usan una animación basada en *sprites*. Los monstruos están formados por cuadrados, los cuales muestran una secuencia de bitmaps texturizadas (conocidos como texturas animadas) para producir la ilusión de movimiento. Hoy en día se sigue usando para objetos lejanos y/o de baja resolución.

  El enfoque más moderno para la animación de personajes 3D es una técnica conocida como animación jerárquica rígida. En este enfoque, un personaje es modelado como una colección de piezas rígidas unidas entre sí en una jerarquía. Eso permite movimientos naturales. El mayor problema son las uniones propiamente dichas.

- Animación por vértices y transformaciones objetivos: la animación jerárquica rígida tiende a verse antinatural por ser tan rígido. La forma de solventar esto es mover individualmente los vértices de forma que los triángulos puedan estirarse y parecer más natural. Una forma de llevarlo a cabo es con la técnica de la fuerza bruta, en el que los vértices son animados por un artistas, y los datos se exportan al motor para que este sepa cómo mover los vértices. Sin embargo es una técnica con gran intensidad de datos.

  Una variante es conocida como animación con transformaciones objetivo. Los vértices se mueven por un animador en un pequeño set de posturas y expresiones "extremas" y por extrapolación y mezcla de las expresiones, el motor calcula la posición de los vértices y renderiza con ello muchas más poses. Suele ser usado para expresiones faciales al ser una parte muy compleja.

- Animación *skinned*: Con la mejora de tecnologías, aparece esta. Tiene muchas de las ventajas de animación por vértices y transformaciones objetivos, pero con más eficiencia. Usada por primera vez en Super Mario 64 y el más prevalente hoy en día. En la animación *skinned*, un esqueleto

está formado por huesos rígidos los cuales permanecen escondidos. Una continua superficie lisa de triángulos llamada piel y que va unida a las uniones de los huesos. Sus vértices siguen el movimiento de las uniones. Cada vértice se puede ponderar a varios puntos por lo que la piel se puede ajustar de forma bastante natural a los movimientos de las uniones.

En las películas, cada aspecto de las escenas es planeada cuidadosamente. Sin embargo, en los juegos, debido a ser tan interactivos no se puede saber de antemano cómo van a moverse y comportarse los personajes. Incluso las decisiones hechas por el computador y no por el jugador ya que están muy ligadas al azar. Es por ello que los movimientos en los juegos no son una sola y larga secuencia de frames, sino que se dividen en varios movimientos distintos. Esos movimientos individuales se les llaman *animation clips* o simplemente animaciones.

Cada clip suele ser una acción bien definida, como correr, andar, saltar... Y algunas están diseñadas para ejecutarse en bucle, otras ejecutarse una sola vez, otras sólo afectan a una parte del personaje mientras que otras afectan a todo el cuerpo.

La única excepción sucede en las escenas no interactivas del juego llamadas cinemáticas de juego (*in-game cinematic* IGC), secuencias no interactivas (*noninteractive sequence* NIS) o vídeos de movimiento completo (*full-motion video* FMV). La diferencia entre estos tres términos está en que FMV se suele aplicar a las secuencias prerenderizadas y reproducidas como video en el juego, mientras que los otros dos se renderizan en tiempo real por el motor.

Otra variación son las secuencias semi-interactivas conocidas como *quick time event* (QTE) donde a una acción del jugador, se muestra otra secuencia no interactiva de forma complementaria. Como ejemplo, en el juego Metal Gear Solid: Snake Eater, durante sus videos donde lo estás viendo en tercera persona, puedes a veces pulsar un botón y ver el mismo video pero desde otra perspectiva, la del jugador.

2.9 La física

Tener en un motor/juego capacidades de simulación física siempre añade puntos de realismo a un juego, aunque no tiene porqué incrementar su diversión. Sin embargo, añadir físicas a un juego va añadido a un coste. Algunas de las cosas que se pueden pedir a un sistema de física son:

- Detección de colisiones entre objetos dinámicos y la geometría del mundo estático
- Simulación de cuerpos bajo la influencia de la gravedad y otras fuerzas
- Sistemas de masa y resorte
- Estructuras y edificios destruibles
- Ray and shape casts (to determine line of sight, bullet impacts, etc.).
- Volúmenes como disparadores, que determinan cuando los objetos entran, dejan o están dentro de regiones predefinidas.
- Permitir a los caracteres recoger objetos sólidos/rígidos
- Máquinas complejas, trampas.

- Vehículos con suspensiones realistas.
- "Muñeca de trapo" (*Rag doll*) del personaje y su muerte
- Simulación de ropa
- Propagación de audio
- Simulación de la superficie del agua y flotabilidad
- Y mucho más

El impacto que puede tener la física en el juego puede ser variado, algunos ejemplos son los impactos en el diseño:

- Previsibilidad: el caos inherente y las variables que caracterizan a una simulación física es también fuente de imprevisibilidad. Si se está seguro de que ha de ocurrir siempre de la misma forma, es mejor hacerlo como animación y no forzar la simulación para que realice de forma predecible dicho comportamiento.

- El control y la personalización/ajuste: las leyes de la física son fijas, pero en el juego podemos personalizarlos y cambiar variables como la gravedad o el coeficiente del rozamiento, lo que nos da cierto control. Pero al mismo tiempo, el cambio de esas variables suelen tener efectos indirectos y difíciles de ver. Es más fácil cambiar la animación del personaje moviéndose que cambiar las variables y forzar la física para que el personaje se mueva a la dirección requerida.

- Comportamientos emergentes: A veces la física introduce comportamientos inesperados, como puede ser ver la tabla de surf volando en el juego de PsyOps.

También tiene su impacto en la ingeniería, entre otros:

- Herramientas pipeline: Un buen pipeline de físicas o colisiones toma tiempo para construirlo y mantenerlo.

- Interfaz de usuario. ¿Cómo controla el usuario la física de los objetos en el mundo? ¿Puede dispararlos, caminar en ellos, cogerlos...?

- Detección de colisiones: Los modelos de colisiones destinados a simulaciones físicas quizás necesitan más información y ser construidos con más cuidado.

- IA: el trazado de la trayectoria a seguir puede no ser predecible en presencia de objetos físicamente simulados. El motor necesita tener en cuenta los puntos que se pueden mover o volar. ¿Puede la IA aprovecharse de la simulación física?

- Grabar y reproducir. Útil para depurar y como una divertida característica del juego. El problema es cuando está presente la simulación dinámica debido a comportamientos caóticos o al azar, que pueden hacer tomar caminos muy distintos con muy pocas diferencias del estado original.

Debido a que las físicas y colisiones son difíciles y consumen mucho tiempo para programarlos, siendo un porcentaje muy significativo del motor y por tanto mucho código a mantener, es usual usar SDKs físicos de terceros, como puede ser ODE "Open Dynamics Engine" (12), que es open source y similar a su contraparte comercial Havok, otra de las más importantes librerías es Havok, un importante estándar en SDKs físicas comerciales. Otros nombres son PhysX, TrueAxis, Bullet, PAL y DMM.

Un dato curioso acerca de la psicología humana y el realismo de los juegos. Cuando jugamos a juegos viejos o de baja calidad, tendemos a olvidar la figura tosca que se mueve de forma antinatural porque es una figura abstracta. Sin embargo, al jugar a un juego de última generación, al ver el personaje tan realista, no realizamos esa abstracción y se nos hace difícil aceptar cuando hace algo que se ve poco natural, aunque sea una cosa muy pequeña. (9)

2.9.1 El sistema de detección de colisiones

Su principal objetivo es la detección de colisiones, es decir, determinar si algún objeto está en contacto con otro. Para poder resolver la pregunta, los objetos se representan por uno o varias formas geométricas, normalmente sencillas como esferas, cubos o rectángulos, y comprobar si hay intersecciones entre dichas formas en un momento dado.

En adición, el sistema también proporciona información sobre la naturaleza del contacto, que se puede usar para evitar situaciones en el que dos objetos están interpenetrados el uno en el otro y corregir dicha anomalía visual. Otras situaciones son permitir uno o varios contactos que juntos permiten que el objeto quede en una posición de reposo, en equilibrio con las fuerzas que actúen contra él. También tiene otros usos como desencadenante para objetos como misiles que explotan al entrar en contacto con otro objeto, o curar al personaje al pasar por encima de un kit médico.

Para lograr todo esto, se necesita proporcionar una información, una representación de colisión en el que se describe las formas del objeto y su posición y orientación en el mundo. Dicha representación es independiente de la representación del objeto en el juego, es decir, de lo que define su comportamiento, su rol, su representación visual.

En el motor de Havok, usan el término *collidable* para referirse a los objetos colisionables, mientras que PhysX los llama actores. En ambos casos, el objeto contiene básicamente dos tipos de información, la forma y la transformación, el primero describe la forma geométrica del objeto, preferentemente formas geométricamente y matemáticamente simples, y la transformación describe la posición y orientación del objeto.

La razón de usar las transformaciones es que las formas son definidas con su centro en el origen, por lo que hay que colocarlo luego en sus coordenadas reales. También, muchos de los objetos en los juegos son dinámicos, por lo que mover un objeto relativamente complejo puede ser costoso, y en vez de actualizar vértices, planos y otras características, es más eficiente hacerlo a través de las

transformaciones, pudiendo mover el objeto sin apenas costes, dando igual su complejidad.

Una última razón para esta forma de proporcionar la información colisionable del objeto, es la cantidad de información necesaria para describir formas más complejas, que pueden usar mucha memoria, por lo que puede ser beneficioso poder usar la misma información para varios objetos. Por ejemplo, un coche de carreras necesita mucha información, información que seguramente sea idéntico en el resto de coches de una carrera y se pueda compartir entre todos. De forma análoga, un objeto puede tener varias instancias de información "colisionable", una por cada parte de su cuerpo.

El sistema mantiene todos estos objetos colisionables, bajo una estructura de datos conocido como mundo de colisión, una estructura bajo el patrón de Singleton. Dicha estructura es una representación del mundo del juego y diseñado para el sistema de colisión. Mantener toda esa información en una estructura de datos propia tiene ventajas sobre mantenerla en cada objeto, como es saber qué objetos pueden llegar a colisionar con otros, evitando tener que comprobar objetos irrelevantes.

2.9.2 Estructuras de colisiones

Uno de los principales problemas del sistema de colisiones es que en cada interacción tenemos que comprobar si cada objeto está colisionando con el resto de objetos de nuestro juego. Por ejemplo si tenemos 100 objetos en nuestro juegos nos daría 100x100=10.000 comprobaciones de colisión en cada bucle de nuestro juego, lo que es una barbaridad para 100 objetos. Divide y vencerás.

La idea básica tras la mayoría de las estructuras de datos es particionar el espacio de forma que se pueda descartar fácilmente regiones que no van a interseccionar, para que no se tenga que testear los objetos dentro de dichas regiones. Ejemplos de estructuras son los *quadtrees*, árboles BSP, *kd-trees* y técnicas de *hashing* espacial.

Los *quadtrees* son una estructura de datos tipo árbol en el que cada nodo tiene a su vez cuatro nodos hijos. Con ello se consigue dividir el escenario en cuadrantes y comprobar colisiones sólo dentro de un cuadrante. Si hay muchos objetos en un cuadrante, dicho cuadrante se subdivide a su vez en más cuadrantes.

Un caso especial aparece cuando un objeto queda a caballo sobre la línea de un cuadrante. Una forma es dejarlo en el nodo padre y comprobar colisiones con los nodos hijos. Una variante de *quadtrees* son los *octrees*, de ocho hijos en vez de cuatro y en la misma idea, puede ser *k-trees*. (13)

2.9.3 Las figuras colisionables

Las figuras tienen mucha base y teoría matemática. De forma simple, las figuras son las regiones del espacio limitado en los que se puede reconocer un interior y un exterior. Las figuras en 2D pueden estar limitadas por una curva o tres o más rectas, en cuyo caso pasa a ser un polígono. En el caso de 3D, la figura tiene

además un volumen también limitado por una superficie curva o por composición de polígonos, en cuyo caso son los poliedros.

En ocasiones, algunos objetos son definidos como superficies, incluso en el 3D donde es una figura geométrica 2D que tiene un frente y un detrás, pero no tienen el concepto de dentro/fuera. Es frecuente en las librerías de colisiones que permitan a las superficies a tener un volumen con un parámetro opcional, indicando cuán fino es la superficie. Con ello se evitan algunos problemas como perder colisiones entre objetos pequeños y muy rápidos, que pueden llegar a atravesar la superficie sin detectar la colisión. Ese es el problema del túnel y más adelante se ofrecen algunas soluciones. (14)

Un sistema de colisión ofrece información sobre la colisión tal como la naturaleza del contacto, que permite poder separar los objetos de forma eficiente, saber que dos objetos han colisionado, que figuras de los objetos son los que están en colisión, detalles individuales de esas figuras e incluso la velocidad de los cuerpos.

Un concepto importante en las colisiones es la convexidad y la no-convexidad (por ejemplo la concavidad) de las figuras. De forma técnica, la convexidad de una figura se define como aquél que ningún rayo originado desde dentro atraviesa la superficie más de una vez. Unos ejemplos más visuales son que figuras como el triángulo, el círculo o el rectángulo son convexos, pero la figura del Pac-man no.

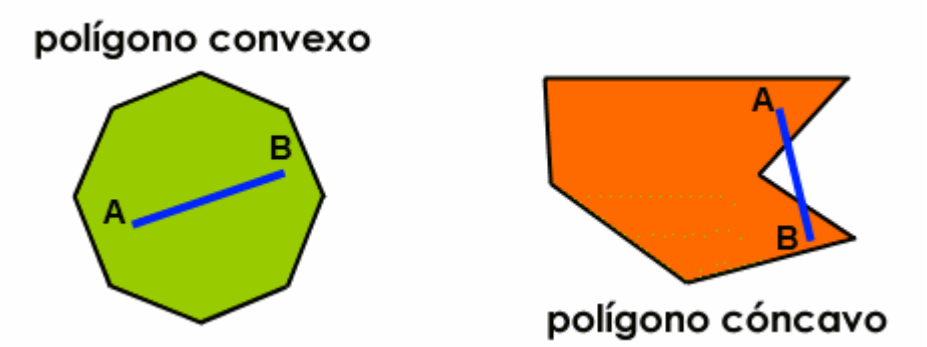


Figura 2-3. Ejemplo de cóncavo y convexo

La importancia de este concepto viene dada por la simpleza y facilidad de computación que tiene detectar las colisiones entre las figuras convexas frente a las no convexas.

2.9.4 Primitivas de colisión

Los sistemas de detección de colisiones normalmente funcionan con un conjunto limitado de figuras sencillas y se les suele denominar las primitivas de colisión. A partir de estas primitivas forman figuras más complejas. Estas figuras son:

- Esferas: el más simple y eficiente en el cálculo de la colisión. Representado por un centro y un radio.

- Cápsulas: figuras en forma de píldoras alargadas. Se forman por dos esferas en sus extremos y un cilindro de lado a lado. Suelen representarse por dos puntos (los centros de las esferas) y un radio. Son más eficientes que los

cilindros o cajas, por lo que suelen usarse en modelos muy similares a cilindros.

- *Axis-Aligned bounding boxes* (AABB): un volumen rectangular con sus caras alineadas en los ejes del sistema de coordenadas. No necesariamente tiene que estar alineado con todos los ejes. Se definen principalmente por dos puntos, uno con las coordenadas mínimas de cada eje, y otro con las coordenadas máximas de cada eje. Muy eficientes en colisiones, pero con la gran desventaja de tener que estar alineado siempre y se ha de recalcular el rectángulo cuando la figura gira, además de que si el objeto deja de estar alineado, el rendimiento y el cálculo de la colisión es muy pobre.

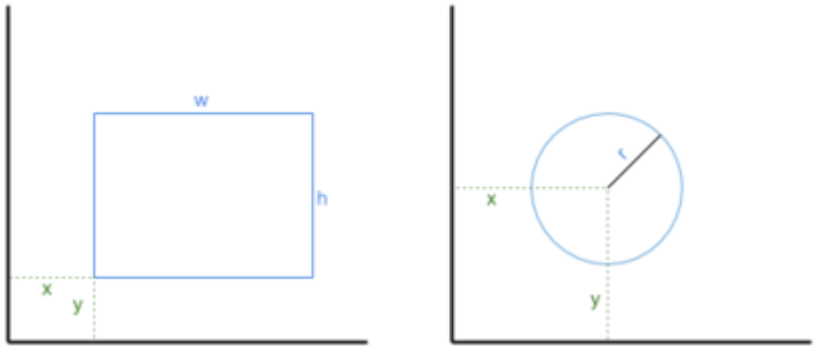


Figura 2-4. Esferas y AABB. Fuente: Genbeta Dev

- *Oriented Bounding Boxes* (OBB): Si al anterior se le permite rotar relativamente a sus ejes, tenemos esta figura que suele representarse por tres medias-dimensiones (la mitad de la altura, de la anchura y de la profundidad) y la transformación que posiciona el centro de la figura y su rotación de forma relativa a los ejes. Muy usados al hacer mejor trabajo detectando colisiones y siendo todavía una figura simple.

- Politopos discretos orientados (*Discrete oriented Polytopes* - DOP): es un caso más general que el AABB y el OBB. Un politopo (15) es la generalización a cualquier dimensión de un polígono bidimensional o poliedro tridimensional. Los DOP son politopos convexos que se aproximan a la figura del objeto, pueden ser construidos a partir de planos que se deslizan a lo largo de sus vectores hasta que entran en contacto con el objeto a ser aproximados. Un AABB es un 6-DOP en el que las normales de sus planos son paralelas a los ejes de las coordenadas y un OBB es un 6-DOP cuyas normales son paralelas a los ejes del objeto. Pueden ser de más planos (k-DOP) y una forma común de hacerlo es con un OBB para el objeto al que van biselando los bordes o vértices.

- Volúmenes convexos arbitrarios. Muchos sistemas de colisión permiten volúmenes arbitrarios construidos de herramientas como Maya. Dichos volúmenes se construyen con base de triángulos o rectángulos y se pasan por una herramienta que confirma que es una figura convexa. La figura entonces es convertida a una colección de planos (básicamente un k-DOP)

representados por k ecuaciones de planos o k puntos y k vectores de normales. Estos volúmenes son más costosos en las colisiones, pero hay algoritmos que resuelven el problema.

- Sopa de polígonos (*Poly soup*): Algunos sistemas de colisión soportan incluso figuras no convexas y totalmente arbitrarias. Normalmente son construidas por triángulos u otras figuras simples y de ahí su nombre (en inglés *polygon soup* or *poly soup*), no necesariamente representa un volumen y superficies. El problema es que no ofrecen información suficiente para diferenciar entre superficie y volumen cerrado para separar los objetos de una colisión por ejemplo, por lo que tienen que guiarse por el frente y detrás de los triángulos (la orientación del triángulo). Ello implica que el objeto ha de diseñarse con cuidado para que los triángulos miren en la dirección correcta. Son usados normalmente para geometrías complejas como terrenos y edificaciones. Es el más costoso de todos en cuanto al cálculo de las colisiones ya que tiene que testear cada uno de los triángulos de los que se compone y tener en cuenta las colisiones falsas entre bordes de triángulos que son compartidos. Por este motivo, los juegos tratan de limitar el uso de estas figuras a objetos que no forman parte de las simulaciones dinámicas.

- Figuras compuestas: algunos objetos se aproximan mejor con una colección de figuras y suele ser la alternativa más eficiente a la sopa de polígonos para objetos no convexos. El sistema de colisiones puede tomar ventaja y testear el volumen convexo que cubre las figuras del objeto y si no hay colisión, no hay necesidad de testear cada una de las subfiguras del objeto.

2.9.5 Cálculo de las colisiones

Como apunte, hay un truquito muy útil: El cálculo entre dos esferas es sencillo, dado por la distancia entre los dos centros, si dicha distancia es menor a la suma de los radios, colisionan. Sin embargo, hay un simple truco que para simplificar la operación de raíz cuadrada en el cálculo de la distancia, se hace lo siguiente:

$$s = c_1 \cdot c_2$$

$$s^2 = s \cdot s$$

$$|s^2| \leq (r_1 + r_2)^2$$

Para el cálculo de las colisiones, se usa mucho el teorema de los ejes separados (16). En este teorema se define el eje de separación entre dos objetos convexos como aquella recta tal que las proyecciones ortogonales de los objetos sobre ella no se solapan. Nuestras OBBs no colisionarán entre sí, si nuestro algoritmo consigue encontrar un eje de separación entre ellas (17). No se aplica para las figuras cóncavas.

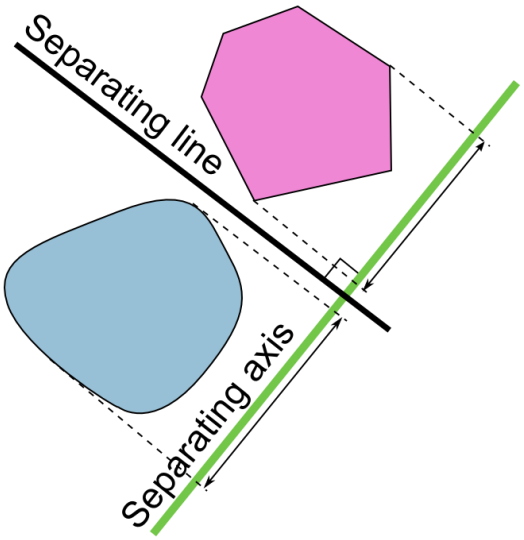


Figura 2-5. Teorema de los ejes separados

Se puede aplicar este teorema muy bien entre AABB y AABB, ya que se garantiza que las caras de ambos están paralelas a los ejes y sólo se ha de comparar los intervalos del eje que guardan.

En el caso de colisiones entre politopos, existe un algoritmo bastante eficiente, el algoritmo de GJK, llamado así por sus inventores E. G. Gilbert, D. W. Johnson y S. S. Keerthi. El algoritmo se basa en una operación geométrica llamada diferencia de Minkowski. Se coge cada punto de una figura y se le resta su contraparte en la otra figura. El set de puntos resultantes es la diferencia de Minkowski.

Lo útil de esa diferencia es que aplicado a dos figuras convexas, la figura resultante (también convexa) contendrá el origen si, y solo si las figuras colisionan.

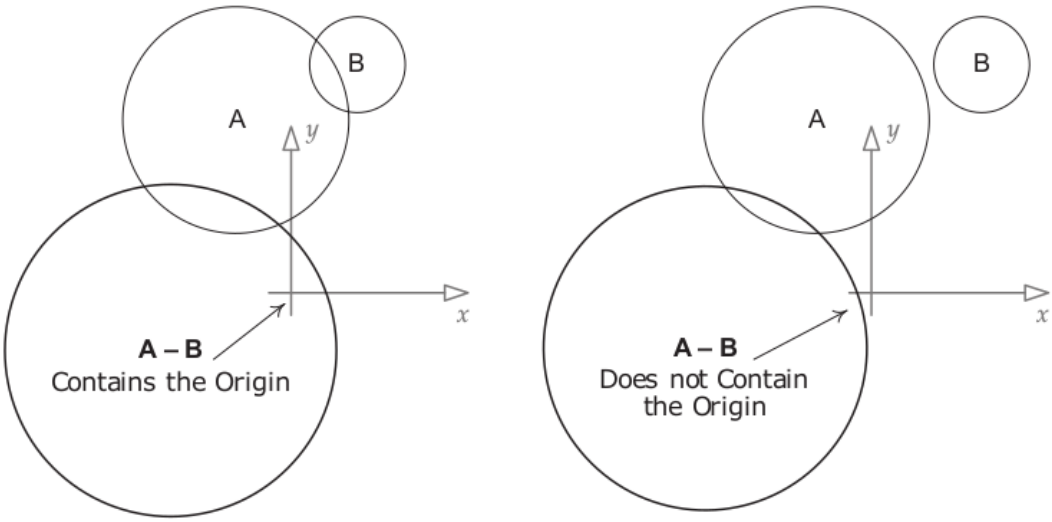


Figura 2-6. Algoritmo de Minkowski

El algoritmo de GJK consiste en encontrar un tetraedro que se halle en la figura resultante de la diferencia de Minkwoski y tenga el origen. Si se encuentra, se tiene colisión.

Para ello empieza con un punto en cualquier sitio de la figura de Minkowski y en cada iteración del algoritmo determina hacia donde crecer para tirar hacia el origen y se añade vértices si es necesario. Cuando se tiene el tetraedro conteniendo el origen, se termina sabiendo que colisionan. Si no es capaz de avanzar hacia él, no encontrando un vértice de soporte para ello, las figuras no colisionan.

Hay un punto a tener en cuenta. Y es el movimiento, que añade complejidad al cálculo de las colisiones. Como normalmente el movimiento se simula en pasos de tiempo discretos, un enfoque simple es calcular las colisiones del objeto en cada paso. Funciona para objetos que no se mueven muy rápidos en relación con su tamaño, sin embargo, falla cuando el objeto es pequeño y se mueve rápido, pudiendo provocar el problema del túnel, que ya mencionamos anteriormente, ya que de un paso a otro, puede haber un salto grande en el que se pasa un objeto y su colisión no se detecta.

Para ello, hay una solución que es el barrido de figuras (*Swept shapes*). Se forma una nueva figura formada por el movimiento de la figura de un punto a otro a lo largo del tiempo y se comprueba la colisión con ella. Este enfoque es de forma técnica, una interpolación lineal y puede no ser una buena aproximación ya que si el objeto gira durante el movimiento, puede quedar una figura poco realista e incluso no convexa. Es decir, que en general no es apropiada para figuras que giran. Es una técnica que puede ser útil para asegurarse de que no se pierden colisiones.

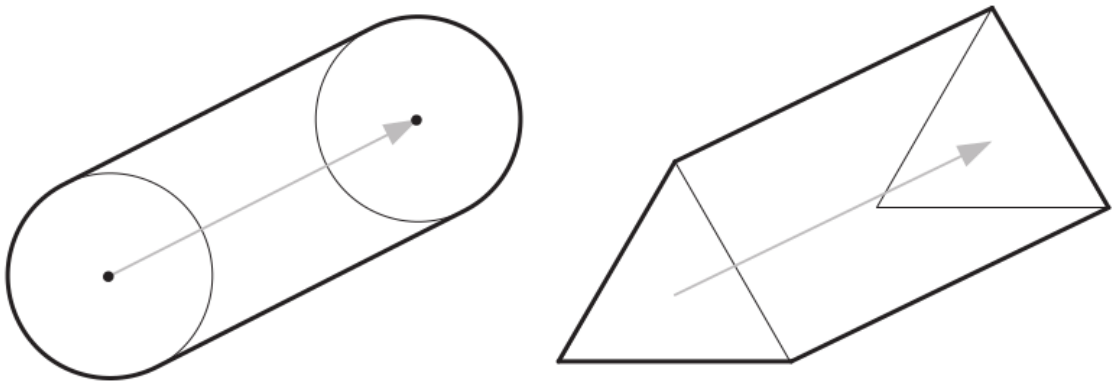


Figura 2-7. Barrido de figuras

Otra técnica para evitar el problema del túnel es el CCD (*Continuous Collision Detection*), detección continua de colisión. Su meta es encontrar el tiempo de impacto más “temprano” (TOI *Time of Impact*) entre dos objetos en movimiento en un intervalo de tiempo.

Es un algoritmo iterativo en el que para cada colisionable se mantiene la posición y orientación del paso anterior y las del paso actual y busca el TOI más temprano a lo largo de la ruta de movimiento. Hay varios algoritmos de búsqueda para ello.

En general, la comprobación de colisiones es una tarea intensiva para la CPU debido a que la comprobación de colisión entre dos objetos es en sí misma no trivial y la mayoría de mundos de juego tienen un gran número de objetos colisionables. Por ello se usan optimizaciones como subdivisiones espaciales para reducir el número de comprobaciones, la coherencia temporal entre frames y las fases que havok usa (fase ancha, fase media y fase estrecha).

En esas fases se comprueba de forma inexacta hasta una más exacta, permitiendo descartar rápidamente los objetos que no colisionan con simples cálculos:

- Fase ancha: se usa los test AABB para determinar que colisionables tienen potencial de colisión. En la mayoría de motores de colisión usan un algoritmo denominado *sweep* and *prune* (18)

- Fase media: donde se comprueba con el volumen general de los objetos si están cerca y pueden estar en colisión.

- Fase estrecha: si resulta colisión con los volúmenes generales, se comprueba en detalle cada figura individual que componen los objetos para determinar finalmente la colisión.

2.9.6 Otras tareas de las colisiones

Además de la detección de colisiones en sí, el sistema de detección de colisiones tiene otras tareas como contestar a hipotéticas preguntas como por ejemplo a qué daría una bala, si puede un coche conducir del punto A al B sin chocar y todos los enemigos que están alrededor del personaje. Esas operaciones son conocidas como consultas de colisión.

Las preguntas más usuales son de "*collision cast*" (consulta de lanzar), a veces llamados simplemente "*cast*", "*trace*" o "*probe*". Un *cast* determina si un hipotético objeto, puesto en el mundo y moviéndose a lo largo de un rayo o línea, impactaría contra algo y qué algo. Son ligeramente distintos a las detecciones de colisiones normales ya que los objetos no están realmente en mundo de colisión y por tanto no puede afectar al resto de objetos que están. De ahí que sean preguntas hipotéticas.

La forma más simple de *collision cast* es con un *ray cast* aunque no es un nombre muy adecuado. Es una línea recta dirigida desde un punto de partida y uno final (la mayoría de sistemas de colisión no admiten rayos infinitos porque lo calculan con una ecuación paramétrica). La recta es testeada en el mundo de colisión y si colisiona con algún objeto, devuelve los puntos de contacto.

Son muy usados, por ejemplo, para determinar si un objeto/personaje tiene visión de otro personaje simplemente trazando una línea entre ambos personajes y determinar si hay colisiones entre medias. Se usa mucho también para los sistemas de armas para ver si la bala impacta, en las mecánicas del jugador (por ejemplo, para determinar si tiene suelo sólido bajo sus pies), sistemas de IA para los

movimientos, búsqueda de un objetivo y otros, sistemas de vehículos (para localizar y pegar las ruedas al suelo) y más.

Otra pregunta común es cuán lejos puede llegar una figura convexa a través de una recta antes de colisionar (*shape cast*). Normalmente se describen con un punto inicial y final, y la figura con sus dimensiones y orientaciones. Al testearlo hay dos casos, que de entrada la figura ya esté en colisión impidiéndole moverse del punto inicial o que pueda avanzar de su punto inicial al no colisionar. En el primer caso puede existir interpenetración y normalmente se reporta los contactos y todos los colisionables. En el segundo caso, cuando haya colisión, puede colisionar con más de un objeto al mismo tiempo, pero nunca interpenetración.

La información que devuelve es un poco más compleja que con los "*ray cast*", ya que de entrada debe ser capaz de reportar contactos múltiples, y se suele informar de los puntos de contactos y otros datos como qué objetos han colisionado con él.

Muy útiles, sirven para determinar cuando la cámara colisiona con objetos e incluso para mover al personaje por un terreno desigual.

A veces se quiere testear que objetos quedan dentro de un determinado volumen, como puede ser ver todos los enemigos que están alrededor del jugador. Havok permite un tipo especial de colisionable conocido como fantasma para ello.

Se parece mucho a un *shape cast* con vector de distancia cero y devuelve su misma información. Pero a diferencia de ella, el fantasma es persistente en el mundo colisionable y por ello puede además tomar ventaja de las optimizaciones de coherencia temporal del sistema de colisión. En realidad, la única diferencia entre el fantasma y el resto del mundo colisionable, es que el fantasma es invisible al resto de colisionables (y no toma parte de las simulaciones dinámicas), pudiendo preguntarle sobre los objetos que colisionarían con el real y se garantiza que no tendrá efecto en el resto de colisionables (incluido otros fantasmas).

Algunos sistemas de colisión permiten otro tipo de preguntas, como Havok que soporta preguntas del tipo que puntos están más cerca, usado para encontrar los puntos de otros colisionables que están más cerca del colisionable dado.

2.9.7 Filtro de colisiones

Es muy común que se quiera deshabilitar colisiones, por ejemplo, cuando el personaje se mete en el agua. Se puede simular que flota, que hay más resistencia, pero no que el agua sea sólido. Por ello muchos sistemas de colisión permiten especificar que colisiones son aceptadas o rechazadas.

Una forma de filtrar dichas colisiones se puede hacer categorizando los objetos y en una tabla comprobar que categorías se permite colisionar con otra categoría. Es el filtro de colisiones por defecto en Havok, donde cada objeto puede tener sólo una categoría.

Normalmente, los desarrolladores necesitan categorizar los objetos ya no solo para controlar como colisionan, sino para controlar efectos secundarios como

puede ser reproducir el sonido de salpicadura cuando el personaje entra en el agua u otro sonido si está pisando sobre madera, metal u otros materiales.

Para ello, muchos juegos implementan una categorización muy similar al sistema de materiales que usa el motor de renderizado. De hecho, muchos juegos usan el término de colisión de materiales para esta categorización. La idea es asociar a cada superficie colisionable una serie de propiedades que define como debe comportarse en caso de colisiones. Esas propiedades pueden incluir sonidos y efectos de partículas, propiedades físicas como coeficientes de fricción, información de filtro de colisiones y cualquier otro tipo de información.

3 Gameplays

Un juego no se define por su tecnología, sino por su *gameplay*, sus mecánicas de juego. Usualmente es definido como el conjunto de reglas que gobiernan las interacciones entre las entidades en el juego, los objetivos, los criterios de éxito y fracaso, las habilidades del personaje, el número y tipos de entidades no jugables (*non-player entities*) y el flujo general del juego como un todo.

Muchos motores proveen un framework con el que construir las reglas del juego, objetivos y elementos dinámicos del mundo virtual. No hay un nombre estándar para cada uno de ellos, pero nos referimos de forma colectiva a ello como el sistema de fundamentos de juego (*engine's gameplay foundation system*). Es la zona donde trazaríamos la línea entre el juego y el motor si se pudiera.

En teoría se puede hacer un sistema que no sepa nada acerca del juego, pero en la práctica, casi siempre tienen detalles específicos a un género o juego concreto. Las diferencias entre motores quedan más patentes en el diseño e implementación de sus componentes *gameplays*, aunque sorprendentemente hay muchos patrones de diseño comunes entre los motores.

Normalmente proveen los siguientes subsistemas de alguna forma:

- Modelo de objetos de juego en tiempo de ejecución: una implementación del modelo de objetos de juego que se muestra a los diseñadores a través del editor de mundos. Este es probablemente el más complejo de todos.

- Gestión de niveles y *streaming*: este sistema carga y descarga los contenidos del mundo virtual, los cuales en muchos motores se hace mediante *streaming* durante el juego, dando la impresión de un mundo virtual muy grande aunque internamente se encuentre dividido.

- Actualización en tiempo real del modelo de objetos: para permitir que los objetos actúen con autonomía en el mundo virtual, cada objeto debe ser actualizado periódicamente. Aquí es donde todos los sistemas del motor trabajan juntos como un todo.
- Manejo de mensajes y eventos: muchos objetos necesitan comunicarse con otros. Normalmente se hace mediante un sistema de mensajería abstracto. Algunos mensajes llevan aparejado cambios en el estado del mundo virtual,

que son denominados eventos, por lo que varios estudios se refieren a este sistema de mensajería como el sistema de eventos.

- Scripting: programar la lógica del juego en C o C++ puede ser pesado, por lo que para mejorar la eficiencia y permitir el manejo a gente no programadora del equipo, se ofrece un lenguaje de scripting integrado, el cual puede estar basado en texto, como Python o Lua, o ser un lenguaje gráfico como Unreal's Kismet.

- Gestión de los objetivos y el flujo de juego: El subsistema maneja los objetivos y el flujo del juego, descrito normalmente como una secuencia, un árbol o un grafo de objetivos del jugador. Los objetivos normalmente se agrupan en capítulos, especialmente cuando el juego es guiado con una historia. El sistema maneja el juego, sigue el cumplimiento de los objetivos y lleva al jugador de un área al siguiente cuando los objetivos son cumplidos. Muchos diseñadores consideran esto la espina del juego.

3.1 Anatomía del mundo del juego

La mayoría de juegos suceden en un mundo virtual llamado *virtual game* compuesto por varios elementos. Estos elementos se suelen clasificar en elementos estáticos (terrenos, caminos, edificaciones... y elementos dinámicos (personajes, vehículos, armas, objetos coleccionables, regiones invisibles para detectar eventos importantes, emisiones de partículas, luces dinámicas...). Aunque a veces esa distinción es un poco borrosa, por ejemplo, en juegos arcade nos podemos encontrar con cascadas que se considerarían elementos dinámicos al tener texturas animadas y efectos, y pueden ser colocados por el diseñador independientemente del terreno y del agua. Sin embargo, desde el punto de vista del ingeniero, se tratarían como estáticos al no interaccionar con el resto del juego.

Un ejemplo de distinción borrosa se ve en los juegos cuyo entorno puede ser destruido. En ese caso, se puede definir tres versiones de cada elemento estático, versión intacta, versión dañada y versión destruida. Esa distinción se traza principalmente por cuestiones de optimización, ya que podemos hacer menos trabajo si sabemos que el estado del objeto no va a cambiar. En realidad, los elementos estáticos y dinámicos son dos extremos de un rango de posibles optimizaciones.

El *gameplay* se suele centrar en los elementos dinámicos del juego ya que aunque los elementos estáticos sean importantes, el software está más preocupado actualizando localizaciones, orientaciones y estados internos de los elementos dinámicos. El término de *game state* (estado del juego) se refiere al estado actual de todos los elementos dinámicos del juego como un todo.

3.2 Niveles

Cuando un juego tiene un mundo virtual muy grande, se suele dividir en varias regiones jugables, cada una de las cuales se llama "*world chunks*", también conocidos como niveles, mapas, etapas o áreas. El jugador normalmente ve sólo uno al mismo tiempo o como mucho un conjunto de ellos durante el juego, y el jugador progresa entre los niveles a medida que el juego se despliega.

Originalmente el concepto de niveles se inventó como un mecanismo para dar más variedad al *gameplay* con las limitaciones de memoria que se tenía. Sólo se podía tener un nivel al mismo tiempo en la memoria, pero el jugador podía igualmente progresar en niveles para una experiencia mucho más rica. Desde entonces, el diseño de los juegos se ha ramificado en muchas direcciones y ya no es tan frecuente los juegos lineales ni es tan evidente las delineaciones entre niveles para el jugador.

Los niveles también sirven como mecanismo para controlar el flujo del juego y para repartir las tareas entre los trabajadores, formando pequeños grupos uno para cada nivel.

El flujo de alto nivel de un juego define una secuencia, árbol o grafo de los objetivos del jugador, objetivos que a veces son llamados tareas, etapas, niveles u ondas (si el juego es principalmente de derrotar oleadas de enemigos). También proporciona la definición de éxito de cada objetivo y la penalización por fracaso. En un juego guiado por una historia, el flujo también incluye varias cinemáticas (cinemáticas, secuencias no interactivas, videos...) para el entendimiento del jugador en cómo se desarrolla la historia.

Al inicio, se mapeaba los objetivos del jugador a los niveles en una relación uno-a-uno. Hoy en día ya no es tan popular y cada objetivo puede estar asociado a uno o más niveles, desacoplándose de los niveles del mundo virtual y permitiendo más flexibilidad. Muchos juegos agrupan sus objetivos en secciones del *gameplay*, llamados normalmente capítulos o actos.

3.2.1 Editor de niveles

La analogía a las herramientas creadoras de contenido como Maya, Photoshop y otros en el *gameplay* es el *game world editor* (editor de niveles). Es una herramienta o suite de herramientas que permite definir trozos de mundo virtual y llenarlo con elementos. Todos los motores de videojuegos comerciales vienen con su editor de mundos.

El editor de mundos de videojuegos permite generalmente inicializar el estado de los objetos.

3.3 Objetos de juego

Los elementos dinámicos del juego normalmente están diseñados como orientados a objetos, siendo un enfoque natural e intuitivo tanto para programadores como para los diseñadores desde el editor del mundo. Aquí los llamaremos objetos de juego para referirnos a cualquier elemento dinámico del juego, pero en la

industria no usan ese término, sino entidades, actores o agentes, entre otros términos.

Un objeto de juego es una colección de atributos (el estado del objeto) y comportamientos (cómo cambia el estado a través del tiempo o en respuesta a los eventos). Normalmente se clasifican por tipo, en el que cada tipo tiene un esquema de atributos y comportamientos y todas las instancias de determinado tipo tienen el mismo esquema. La mayoría de motores soportan también la herencia de objetos, lo que permite reutilización del código y diseño.

El término de modelo de objeto tiene dos significados:

- Conjunto de características dadas por un lenguaje de programación o diseño formal del lenguaje

- Una específica interfaz de programación orientada a objetos.

Aquí utilizaremos el término modelo de objeto de juego para referirnos a las facilidades del motor de videojuego para la modelación y simulación de elementos dinámicos en el mundo virtual del juego. Lo que viene a tener parte de las dos definiciones anteriores:

- Es una interfaz de programación orientada a objetos específica hecha para resolver el problema particular de simular los conjuntos específicos que forman un juego.

- Suele ofrecer adicionalmente la característica de extender el lenguaje en el que el motor ha sido escrito, por ejemplo, si está escrito en C, los programadores pueden añadir características de orientación a objetos.

Recordando que un motor consta de dos partes, la suite de herramientas y el componente en tiempo de ejecución, el modelo de objetos de ambos no tienen por qué ser el mismo, por lo que un juego puede tener dos modelos distintos pero interrelacionados, uno que es el modelo de objetos del lado de las herramientas definido por el conjunto de tipos de objetos de juego y que ven los diseñadores, y otro, el modelo de objetos en tiempo de ejecución es definido por el conjunto de lenguajes y software usados por los programadores para implementar el modelo dado por la herramienta.

3.3.1 Arquitecturas del modelo de objetos

Ofrece típicamente casi todas las siguientes características:

- Lanzar/destruir objetos del juego dinámicamente
- Enlazar a los sistemas más básicos del motor, algunos objetos pueden generar sonidos, otras animaciones, colisiones, algunos de los cuales mediante el motor de físicas... el sistema tiene que asegurarse de que cada objeto tiene acceso a los servicios del sistema del motor del que depende.
- Simulación del comportamiento de los objetos en tiempo real
- Definir nuevos tipos de objetos
- Ids únicos para los objetos
- Búsqueda de objetos de juego
- Referencias a los objetos del juego

- Soporte para máquinas de estados finitos. Algunos juegos se modelan mejor de esa forma, con objetos en uno o más estados posibles, cada uno de los estados con sus atributos y características de comportamiento.
- Replicación en red: para juegos multijugador en el que se necesita replicar el estado del juego para cada máquina aunque sólo sea controlado por uno solo.
- Guardar y cargar los juegos/persistencia de objetos

En el editor de mundos, el diseñador tiene un modelo de objetos de juego abstractos, en el que define los tipos de los elementos dinámicos que pueden existir, cómo se comportan y otros atributos. En tiempo de ejecución, el sistema de fundamentos del juego tiene que proveer una implementación concreta de ese modelo.

Como se ha comentado antes, el modelo de objetos de juego en tiempo de ejecución no tiene por qué ser el mismo que el modelo de objetos de la suite de herramientas. El modelo en tiempo de ejecución es la manifestación en el juego del modelo de la suite de herramientas mostrada en el editor de mundos. Pueden seguir uno de las dos arquitecturas básicas:

- Centrado en objetos: cada objeto del lado de las herramientas se representa en tiempo de ejecución como una simple instancia o pequeña colección de instancias interconectadas. Cada objeto tiene atributos y comportamientos encapsulados en la clase/s del objeto. El mundo es una colección de objetos.

- Centrado en propiedades: cada objeto del lado de las herramientas se representa por un único id y las propiedades de cada objeto se encuentra distribuidas entre varias tablas cuya llave es el id. Las propiedades en sí suelen estar implementadas como instancias de clases *hard-coded*. El comportamiento del objeto es definido implícitamente por la colección de atributos. Por ejemplo, si tiene un atributo vida, puede entonces perder vida y morir, si tiene la propiedad MeshInstance, entonces puede renderizarse en 3D.

Las propiedades del objeto se tienden a guardar en estructuras de datos personalizadas cuya eficiencia puede ser muy buena o mala. Ultima Online (un juego) usa texto para definir las propiedades, flexible, fácil y con un cierto coste de memoria. Thief: Deadly Shadows usa un sistema de propiedades bastante complicado que es en realidad orientado a objetos: en él puede definir propiedades en un prototipo (*archetype*) y sobrescribirlas o crear unas nuevas para un objeto particular que se coloca sólo una vez en el juego. El sistema es eficiente en memoria al no copiar datos de las propiedades, pero tiene más coste de CPU al ser esencialmente una estructura en árbol. (9)

3.3.1.1 Centradas en objetos

Con este enfoque, se puede usar varios diseños posibles. No tiene por qué ser en un lenguaje orientado a objetos. Por ejemplo, Hydro Thunder está enteramente en C y usa estructuras, con unos pocos tipos de objetos y usando un modelo simple.

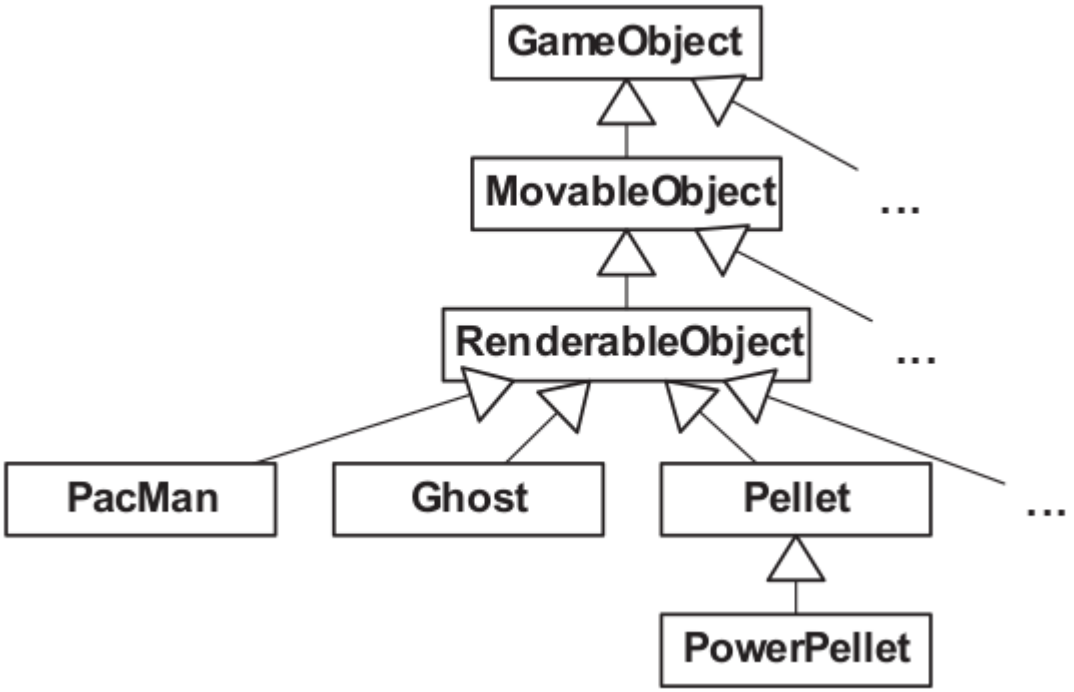


Figura 3-1. Jerarquía del Pac-Man

Jerarquía de clases monolítica: Una jerarquía de clases es la forma más intuitiva y sencilla de representar una colección de tipos de objetos interrelacionados. Un ejemplo de jerarquía aplicable al juego de Pac-man es la figura 3.1.

Se puede ver que todo hereda de una clase común, la clase objeto de juego que da facilidades comunes como la serialización u otros. La clase movable representa cualquier objeto que tiene posición y orientación. Asi mismo, la clase renderizable permite al objeto ser renderizado. Finalmente, de esa clase se derivan las clases para los fantasmas, Pac-man, bolitas y bolitas de fuerza que hacen el juego.

La idea básica es tener las funciones genéricas y comunes arriba en la jerarquía, y a medida que se va descenciendo, se van especificando e incrementando funcionalidades.

El problema que puede surgir con este enfoque es con la profundidad y anchura de las jerarquías. Cuanto más extensos, más problemas dan al ser más difícil de entender, mantener y modificar clases. La jerarquía de clases, lo más plano posible. Es un error común que se hace al sobrediseñar o al contrario sus clases y su jerarquía. Es algo que requiere práctica.

Una buena regla a seguir es que cada clase tenga una única responsabilidad y sus árboles de herencia no deberían tener más de dos o tres niveles de profundidad. Siempre hay excepciones pero mejor evitarlo.

Por el otro extremo, un problema común es la clase Blob bien descrito por los anti patrones (19), una clase que contiene un poco de todo y suele venir dado por la vagancia del programador de hacer nuevas clases más pequeñas y separadas. (9)

En cada nivel, se utiliza un criterio para dividir la clasificación de forma más refinada, sin embargo, una vez elegido el criterio, se vuelve difícil e incluso imposible usar otro criterio distinto. Por ejemplo, en la taxonomía de clasificación animal, se clasifica por los rasgos genéticos, pero nada sobre el color. Si quisiéramos clasificar por colores, tocaría hacer un árbol distinto.

Esta limitación se puede ver reflejada en la programación al examinar un árbol que trata de combinar varios criterios distintos de clasificación en un sólo árbol, haciéndolo confuso. En ocasiones, esas concesiones se hacen para incluir un tipo nuevo de objeto no planificado en el árbol.

Con esta última situación, el programador puede optar por hacer algunos “trucos” no muy buenos. Por ejemplo, una taxonomía que incluye vehículos, de la cual heredan vehículos de agua y vehículos de tierra. El problema llega con un vehículo anfibio y el programador decide hacer una clase anfibio que herede de tierra y de agua. Esa situación puede terminar en el diamante de la muerte, en el que un objeto tiene múltiples copias de su clase base.

Muchos estudios prohíben la herencia múltiple o lo limitan mucho por los problemas que conlleva eso.

Otra forma de resolver el problema es la mezcla de clases, una forma limitada de herencia múltiple. En ella, una clase puede heredar de una sola clase base, pero heredar también cualquier número de clases de mezcla (clases independientes sin clase base). Esto permite meter funcionalidad común factorizada en esas clases de mezcla y parsearlos en la jerarquía donde fuera necesario. En estos casos, normalmente es mejor usar un patrón de composición o agregación en vez de herencias.

Al principio, cuando se diseña una jerarquía de clases monolítica, las clases raíces suelen ser muy simples. Sin embargo, a medida que se añade funcionalidad al juego, el enfoque de compartir código entre clases no relacionadas hace que las características suban en la jerarquía (*bubble-up effect*). Si eso se permite, tiene la desventaja de no poder encapsular bien las funcionalidades de los subsistemas del motor.

Quizás la mayor causa del sobre-uso de la jerarquía monolítica es el abuso de la relación “es un” en el diseño orientado al objeto, siendo en varias ocasiones una relación de “tiene un”. Esa relación es una composición. Convertir las relaciones “es un” a “tiene un” puede ser muy útil para reducir el tamaño y las restricciones de la jerarquía.

Se puede pasar de una jerarquía como la de la figura 3.2 a la figura 3.3. La cual nos permite crear nuevos objetos seleccionando sólo las características deseadas y sin estar obligados a heredar de alguna clase que pueda tener algo que no nos interese o no sea necesario. Esas clases aíslan las características del objeto de juego y son a veces llamados componentes o servicios de objetos. También facilita el mantenimiento, extensión o refactorización de cada componente sin afectar al resto y facilitan el entendimiento y testeo, al igual que se facilita el mapeo directo con el subsistema concreto del motor.

Un pequeño problema que puede tener este enfoque, es que la clase que actúa como *hub*, en este caso el objeto de juego, es tradicional que "posea" los componentes de los que se compone, significando que maneja el tiempo de vida de sus componentes. Sin embargo, ¿cómo sabe el objeto qué componentes ha de crear?

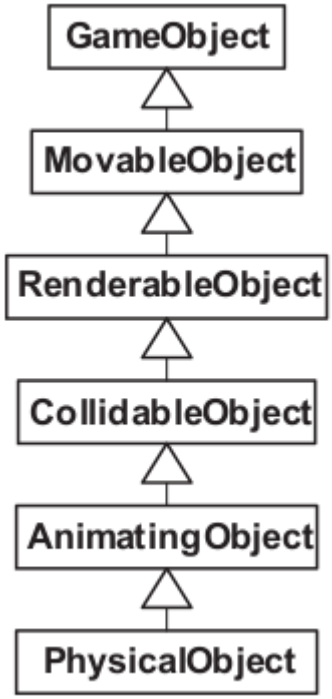


Figura 3-3. Jerarquía monolítica

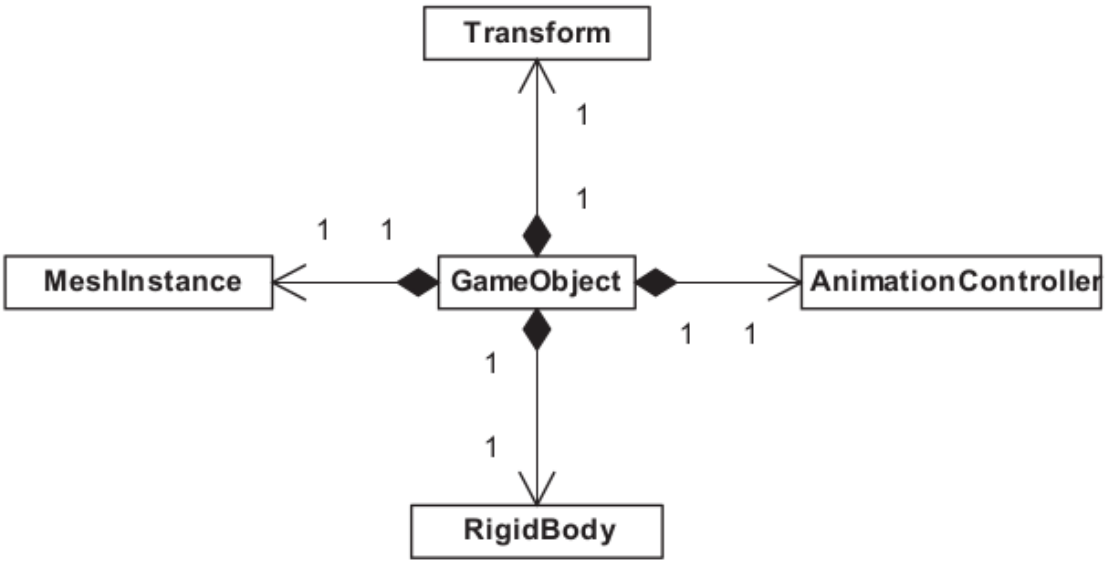


Figura 3-2. De composición

Una de las soluciones más simples es proveer al objeto raíz de punteros hacia todos sus posibles componentes, los cuales en su constructor inicializa a null. Cada tipo de objeto de juego deriva de esa raíz e inicializa aquellos componentes que usa.

Otro enfoque más flexible aunque más complejo, es proveer a la raíz de una lista genérica de componentes, los cuales derivan de una clase base común. Esto permite crear nuevas clases sin afectar a la raíz y tener un número arbitrario de instancias de cada tipo de componente.

Si intentamos llevar el modelo de componentes al extremo, nos encontraríamos con la funcionalidad del objeto de juego totalmente fuera, convirtiéndolo en un simple contenedor, por lo que se podría eliminar y usar únicamente el id del objeto para identificar los componentes. Este tipo de arquitectura es llamado modelo de componentes puro.

No es tan simple como suena y sigue teniendo algunos problemas. Por ejemplo, se sigue necesitando alguna forma de definir los tipos concretos de los objetos que el juego necesita e instanciar los componentes correctos en el orden correcto cuando se crea una instancia de ese tipo de objeto.

En vez de usar los constructores para ello, es bueno usar en su lugar un patrón de factoría (*pattern factory*), en el que se define una clase por cada tipo de objeto, con un constructor virtual el cual es sobrescrito para crear adecuadamente cada tipo de objeto.

Otra forma de hacerlo es con el *data-driven model*, donde los tipos de objetos son definidos en un fichero de texto parseable por el motor y que es consultado cuando el objeto es instanciado.

Otro problema del modelo de componentes se encuentra en la comunicación entre los componentes, ya que anteriormente, ese problema lo manejaba el “hub”, la clase que unificaba los componentes. Una forma de hacerlo es que la clase busque al resto de componentes por su id. Pero seguramente se quiera una forma más eficiente, como tenerlos previamente guardados en una lista circular. También es preciso que conozcan a priori el componente con el que quieren comunicarse o hacer *multicast*. Ninguna de las opciones es la ideal.

3.3.1.2 Centradas en propiedades

Otro punto de vista, en vez de pensar en una serie de objetos que tienen una serie de propiedades, podemos pensar en una serie de propiedades, por ejemplo la posición, y bajo ella, una serie de objetos que poseen esa propiedad y los valores correspondientes.

Es más similar a una base de datos relacional que a un modelo de objetos. Cada atributo funciona como una columna en la base de datos siendo el id del objeto la clave primaria. El comportamiento, según el motor, se puede implementar en las propiedades propiamente dichas y/o por código script.

Si se hace a través de las propiedades, cada tipo de propiedad se puede implementar como una clase, siendo tan sencillos como un *boolean* o un número,

y tan complejos como un objeto renderizable o un "cerebro" de IA. El comportamiento puede ser provisto por los métodos de dichas clases y el comportamiento final se compone de la agregación de comportamientos de las propiedades del objeto.

Por ejemplo, un objeto con la propiedad vida, que puede ser dañado y/o destruido. El objeto de vida puede reaccionar ante los ataques realizados decrementando la vida del objeto e igualmente comunicarse con otras propiedades del mismo objeto para comportamientos colaborativos como puede ser a la propiedad de animación para renderizar la reacción.

Si se hace a través de scripts, todo objeto de juego puede tener un atributo que guarde la id del script que maneje el comportamiento.

El enfoque en las propiedades suele ser más eficiente en memoria al guardar sólo los datos necesarios, y fácil de construir un modelo al estilo *data-driven*, definiendo nuevos atributos fácilmente y sin recompilar el juego al no cambiar definiciones. Sólo cuando se definen nuevos tipos de atributos.

Igualmente, es más eficiente con la caché al tener todos los datos del mismo tipo de forma contigua en memoria.

Las desventajas que tiene es que es más difícil depurar, y es más difícil aplicar relaciones entre las propiedades al igual que implementar un comportamiento a gran escala simplemente uniendo los comportamientos de cada parte.

Llegados a este punto, hemos hablado de propiedades y componentes. Por componente nos referimos a un subobjeto en un diseño centrado en objetos, que no es lo mismo que propiedad de un objeto. Sin embargo, están bastante relacionados, por lo que la distinción entre uno y otro es irrelevante en unos cuantos casos.

3.4 Formatos de dato de niveles

Hay que guardar datos sobre los objetos del juego. Una escena se compone de elementos estáticos y dinámicos, los cuales tienen unas posiciones y atributos iniciales. También se tiene que guardar qué tipo son esos objetos para su instanciación a la hora de cargar el mundo.

Una forma de guardar una colección de objetos del juego en disco es con imágenes binarias de los objetos (*binary object images*), es decir, cómo se ven exactamente en memoria en tiempo de ejecución, por lo que lanzar los objetos es trivial, ya que al cargar el nivel del juego, ya están preparados los objetos en sí.

Sin embargo no es tan simple, guardar las instancias de las clases puede ser problemáticas, entre ellas, el tener que manejar de alguna forma los punteros y la posibilidad del cambio de orientación de los bytes en memoria. Igualmente, no es tan flexible, gran contra al ser los niveles de un juego lo que más cambia y más rápido se ha de cambiar.

Otra opción es serializar los objetos. La serialización es soportada nativamente por varios lenguajes de programación. C++ no es el caso, pero hay varios sistemas que lo han construido de forma exitosa. Se han usado XML por ser un formato popular y fácil de leer y entender, además de soporte de jerarquías. El problema

es la lentitud del parseo, lo que ha llevado a usar un formato binario propietario para compensar.

La desventaja de ambas formas de guardar los datos, es que se basan en el objeto en tiempo de ejecución del juego y requiere que el editor de mundos lo conozca y trabaje de forma íntima con la implementación en tiempo de ejecución. Una forma de evitar esa acoplación es abstrayendo la descripción de los objetos, de forma que cada objeto del juego guardado en el fichero, sea un pequeño bloque de datos, a veces llamado *spawner*, que representa sólo los datos del objeto a instanciar e inicializar. Incluye el id del objeto del lado de las herramientas y una tabla de clave-valor para describir los atributos iniciales.

Cuando el objeto es lanzado, la clase apropiada es instanciada (determinada por el id de los datos) y usa dichos datos para inicializarse. Dicho *spawner* puede ser configurado para lanzar inmediatamente el objeto al cargar el nivel, o esperar un tiempo hasta que sea requerido.

Dichos *spawner* se pueden implementar como clases principales y tener una interfaz adecuada, al igual que usarse para otros propósitos como definir puntos del mapa importantes (*position spawners* or *locator spawners*) para un personaje IA, donde originarse un efecto de partículas, puntos a lo largo de una pista de carreras...

Al poder crecer mucho los atributos de un objeto de juego, es realmente de ayuda definir valores por defecto del objeto. Esto tiene un pequeño problema y es que si hemos puesto varios objetos y luego cambiamos uno de los atributos para que por defecto sea otro valor, las nuevas instancias quedan con ese nuevo valor, pero no los que ya están puestos.

De forma ideal, el sistema del *spawner* actualizaría todos los objetos ya instanciados con los cambios del valor por defecto en aquellos que no haya sido sobrescrito. Una forma sencilla es omitir la clave-valor de sus atributos. Cuando un atributo es omitido, se usa el de por defecto.

Los beneficios del *spawner* son básicamente la simplicidad, flexibilidad y robustez. Es más sencillo manejar una tabla de claves-valores que un objeto binario o serializado. Dicha tabla ofrece robustez además de la flexibilidad al poder ignorar sin más alguna clave-valor que no espere o tenga error, usando en su lugar el valor por defecto.

Igualmente simplifica el diseño e implementación del editor de mundos al sólo tener que conocer el manejo de las claves-valor y los esquemas de los tipos de objetos, sin tener que compartir código con el motor del tiempo de ejecución.

3.5 Guardado de juegos

Es similar a la carga y descarga de los niveles del juego en el sentido de ser capaz de cargar el estado del juego desde disco o *memory card*. Sin embargo los requisitos son distintos. Los niveles guardan las condiciones iniciales de los objetos dinámicos en el mundo, pero también toda la información de los estáticos, como los suelos, los datos de las colisiones y demás, los cuales ocupan mucho espacio.

Un juego guardado tiene que almacenar también el estado de los objetos en el mundo, pero no le hace falta un duplicado de toda la información ya determinada por el nivel, por lo que algunos objetos se pueden omitir, y de otros guardar sólo parte de la información.

Por ello, los juegos guardados tienden a ser ficheros más pequeños que los niveles y se centran en comprimir la información y omitir la que pueden. El tamaño es importante sobre todo cuando se permiten guardar varios juegos distintos.

Un enfoque para el guardado de partidas son los *check points*. Los *check points* son puntos específicos del juego donde se permite guardar o se guarda automáticamente. Los beneficios de esto es conocer el estado del juego de antemano y su vecindad, por lo que los datos son exactamente los mismos, dando igual que jugador está jugando, con lo que ya no es necesario incluirlo.

Como resultado, los ficheros pueden ser muy pequeños, con el nombre del último *check point* y alguna información del jugador como los puntos, cantidad de vida, armas y similares. Algunos juegos ni eso, comenzando en el punto con un estado conocido. El único punto en contra puede ser la frustración del jugador sobre todo si los puntos son pocos y están alejados.

El otro enfoque es permitir guardar en cualquier momento/lugar. Para permitirlo, el fichero guardado es más grande, ya que han de guardar las localizaciones y estados de los objetos relevantes. Con este diseño, el fichero contiene lo mismo que un nivel menos los objetos estáticos y alguna cosa omisible como puede ser el tiempo exacto de animación y velocidad, confiando en la poca memoria de los jugadores.

3.6 Carga y *streaming* de mundos

Un último retazo entre la separación del editor de mundos y el modelo de objetos de la parte de tiempo de ejecución, es la forma de cargar los niveles y mundos en memoria y descargarlos cuando no es necesario.

El sistema de carga de niveles/mundos tiene dos responsabilidades: gestionar la I/O necesaria para cargar el mundo y otros recursos necesarios del disco a memoria y gestionar la asignación y deasignación de memoria para dichos recursos. El motor necesita además gestionar el lanzado y destrucción de los objetos del juego en el sentido de asignar y deasignar memoria.

Una forma sencilla de hacerlo es permitir que uno y sólo un nivel (un *world chunk*) sea cargado en memoria al mismo tiempo. Cuando el juego empieza por primera vez y entre niveles, se muestra al jugador una pantalla estática o con una simple animación de carga para que espere mientras se carga el nivel.

El manejo de la memoria aquí es muy sencillo. El mejor para estos casos es el basado en pila. Al inicio del juego se carga todo recurso necesario a lo largo de todos los niveles en el fondo de la pila. Esos datos son denominados datos *load-and-stay-resident* (LSR) y se guarda la situación del puntero una vez terminado. A partir de ahí, todo recurso asociado a un nivel, es cargado justo encima y cuando se completa o cambia de nivel, se libera toda la memoria devolviendo el puntero en la parte superior del LSR.

El único problema es que sólo se puede tener niveles discretos y no mapas grandes y continuos, y al sólo poder tener un nivel cargado en memoria, entre cambio y cambio se tarda mientras se cargan recursos y no hay mundos cargados, por lo que se muestra al jugador una pantalla de carga.

Otra forma de evitar cargas de pantalla es ir cargando el siguiente nivel mientras se está jugando el actual. Una forma simple de conseguirlo es dividir la memoria en dos partes iguales y mientras se tiene cargado el nivel actual en un bloque A, se carga el siguiente nivel en el bloque B usando librerías E/S asíncronas (por ejemplo, en otro hilo diferente).

Una variante es usar dos bloques no iguales, uno grande con un nivel completo (*full game world chunk*) y otro más pequeño, lo justo para un pequeño *chunk* (*air lock*). El jugador está jugando en el full y cuando entra en el air *lock*, en la que hay garantía de que el jugador no puede volver atrás al full *chunk*, ya sea porque se cierra la puerta o algún otro impedimento. En ese momento, el *full chunk* es descargado y se carga el siguiente mientras se entretiene al jugador en el air *lock*.

El único momento en el que se presenta una pantalla de carga es al principio del juego, cuando todavía no se ha cargado ni el full ni el air *lock*.

Para un efecto superior, el de sentir que está jugando en un mundo grande y contiguo, sin transiciones y de forma ideal, sin meter al jugador en regiones *air lock* periódicamente se usa la técnica de *streaming*. El objetivo es cargar los datos mientras se está jugando en tareas regulares y gestionar la memoria de forma que elimine la fragmentación mientras carga y descarga los datos según se necesite.

Con los avances, se tiene más memoria por lo que se puede tener varios niveles al mismo tiempo. Por poner un ejemplo, se divide la memoria en tres partes iguales, en los que al principio del juego se cargan los niveles A, B y C, siendo A donde el jugador empieza. Cuando el jugador se adentra en B y está lo suficientemente lejos de A como para que no se pueda visualizar, se descarga y en su lugar se carga el nivel D y así de forma continua.

El problema que conlleva es que implica grandes restricciones al tamaño de los niveles, obligando a que sean de tamaño similar, de forma que llenen todo lo posible su trozo de memoria pero nunca más de la cuenta.

Una forma de evitarlo es granular más la subdivisión de la memoria, dividiendo cada recurso del juego, desde los mundos hasta las texturas, animaciones y otros, en bloques de datos iguales. De esta forma, se puede usar a *chunky*, *pool-based memory allocation system* sin tener que preocuparnos de la fragmentación en memoria.

La pregunta que llega es cómo determinar qué recursos cargar. El método que utiliza Uncharted: Drake's Fortune es dividir los mundos en *chunks* geográficamente adyacentes. Un simple volumen convexo conocido como región engloba cada uno de los *chunks* y las regiones a veces se solapan entre sí.

Cada región contiene una lista de los *chunks* que deberían estar en memoria cuando el jugador está en esa región. Con ello, se puede determinar los *chunks* que deben estar haciendo una unión de las listas de todas las regiones que solapan en el lugar del jugador. El sistema de carga de niveles compara esa lista con la

lista que guarda y carga y descarga según si ha aparecido/desaparecido algún *chunk*.

Los niveles son diseñados de forma que el jugador no vea desaparecer un *chunk* al ser descargado y que haya tiempo suficiente para cargar el siguiente antes del primer contacto visual del jugador con esa zona.

3.7 Gestión de la memoria en los *spawning*

Una vez cargado el nivel, se necesita manejar el *spawning* de los objetos. Muchos motores tienen algún sistema que maneja su creación y destrucción cuando es necesario. Al ser la instanciación dinámica un tanto lenta, han de tener cuidado y asegurarse de realizarlo de la forma más eficiente posible, vigilando la desfragmentación.

Algunos motores lo resuelven cortando de raíz los problemas deshabilitando la asignación de memoria dinámica durante el juego, permitiéndolo sólo en los niveles. Todos los objetos son creados inmediatamente después del nivel. Esto evita desfragmentación y se conoce a priori la memoria usada. El problema es la limitación a los diseñadores del juego, teniendo que conocer todos los objetos que han de aparecer y es un problema si quieren dar un suplemento infinito de vida, munición, enemigos y similar.

En el caso de permitir la asignación dinámica, el principal problema es la desfragmentación. Debido a distintos tamaños y vida, no se puede usar la asignación pool ni siquiera la de pila. Algunas formas de resolverlo son:

- Usar una memoria *pool* para cada tipo de objeto: en el caso de que se garantice que cada objeto del mismo tipo sea del mismo tamaño. Resuelve la fragmentación aunque con el coste de mantener varios pools por separado y saber cuántos objetos van a haber en cada pool, con el riesgo de si se asigna mucho espacio para un pool se desperdicia y si se asigna poco, puede quedarse corto.

- Con la misma idea de las memorias *pool*, se crean pools por tamaños y se guarda el objeto en el pool más pequeño que entre, permitiendo guardarlo en un pool cuyo espacio sea mayor que el requerido, reduciendo con el número necesario de pools y evitando el posible desperdicio de memoria en cada pool.

- Otra forma de eliminar la fragmentación es atacándolo directamente, enfoque conocido como recolocación de memoria. Que viene a ser mover los bloques de memoria para rellenar los huecos. Es fácil mover los bloques pero se ha de tener cuidado con los punteros que apuntan a los bloques que se mueven.

3.8 Referencias y consultas de objetos

Todo objeto del juego necesita de un ID único que le permita identificarse en tiempo de ejecución para, entre otras cosas, permitir la comunicación entre objetos. El id es igualmente útil para las herramientas.

Una vez encontrado el objeto se necesita una referencia hacia él, como simples punteros o algo más sofisticado como *handlers* o punteros inteligentes.

Los punteros son simples, fáciles, eficientes, flexibles y potentes. Sin embargo si no se tiene cuidado aparecen problemas como los objetos huérfanos (objetos no necesarios pero que no son referenciados por otro objeto, se pierden y nunca se liberan), *stale pointers* (punteros a bloques de memorias de un objeto válido que se ha liberado y si se intenta escribir o leer el programa casca) o punteros inválidos (que apuntan a direcciones inválidas, como null o datos malinterpretados).

Aún con sus ventajas, los programadores son cautelosos con ellos sobre todo ante prácticas peligrosas para los punteros como puede ser el desplazamiento de memoria para evitar la desfragmentación. Para ello, hay otros tipos de punteros, los punteros inteligentes.

Los punteros inteligentes son objetos pequeños que actúan como punteros y mantienen algunos datos más que permite llevar un conteo de referencias (objetos que tienen el puntero) tras el cual si queda a cero, automáticamente borra el objeto. Puede detectar si el objeto es borrado, retornando null.

Los punteros inteligentes también tienen sus problemas. Fáciles de implementar pero difíciles de usar correctamente, ya que hay varios tipos de punteros inteligentes. El por qué no usarlo para todo viene dado por su ligero mayor uso de CPU y memoria, el cual se nota si se tiene miles de objetos a manejar y procesar en cada frame. Por otro lado, quitan algo del control sobre la memoria del puntero. Un consejo personal del autor del libro Game Engine Architecture (1) es alejarse de los punteros inteligentes o usar una solución madura como boost. (9)

Escoger uno u otro depende del contexto. Un puntero desnudo está bien para guardar dentro de la clase que lo usa y sólo él usa. En el caso del sistema de eventos por ejemplo, se usan los punteros inteligentes ya que no está claro quiénes pueden tener esa referencia y tener la seguridad de destruirlo sin problemas.

El Conteo de referencias es un mecanismo normalmente usado para el manejo de memoria. Hay que tener cuidado de los sistemas con el conteo de referencia, pues algunos borran el objeto automáticamente cuando el conteo es cero, y otros dejan esa tarea al programador, pudiendo crear una fuga de memoria (memory leak).

El problema es que sumar y restar referencias al objeto cuando se usa es fácil de olvidar, por lo que existen plantillas(*templates*) de C++, como shared_ptr del estándar TR1 C++ que lo hacen de forma automática sobrescribiendo los operadores y el constructor copia. (9)

Un par de cosas a tener en cuenta en el uso de punteros inteligentes es que no se puede tener nunca dos objetos distintos manejados por punteros inteligentes y

apuntando uno al otro y viceversa. Y cada vez que se crea un punto inteligente, asegurarse de crearlo desde un *raw pointer* con el operador new.

Hay una tercera forma, con los *handle*, los cuales actúan como punteros inteligentes en muchos casos, pero siendo más simples de implementar y por ello, menos propensos a errores. Un *handle* es básicamente un índice *int* en una tabla de *handles* global. La tabla contiene punteros a los objetos que referencia el *handle*. Para crear un *handle*, se busca en la tabla la dirección del objeto en cuestión y se guarda su índice en el *handle*.

Para desreferenciarlo, indexa el slot apropiado en la tabla y dereferencia el puntero contenido ahí.

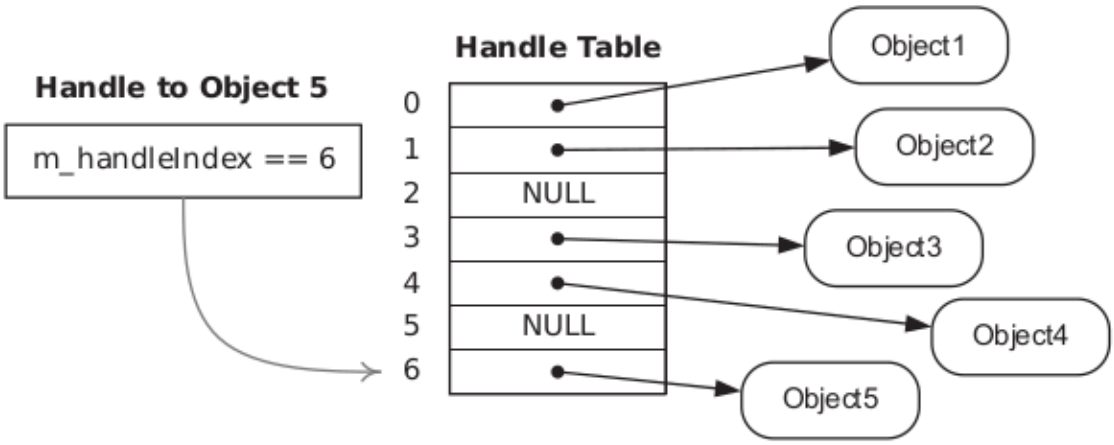


Figura 3-4. Tabla de referencias

Con este enfoque, si un objeto es borrado, simplemente se pone a null en la tabla y todas las referencias a él se actualizan. Igualmente, soportan recolocación de memoria al poder buscar la dirección en la tabla y actualizarlo. Otras ventajas es poder iterar rápida y eficientemente todos los objetos del juego y usarlo para consultas de otro tipo.

En tiempo de ejecución se necesita alguna forma de buscar objetos con determinados criterios y normalmente se necesita también consultas basadas en cercanía. De forma ideal, sería una combinación de flexibilidad y rapidez que por lo general no es posible. Por ello los desarrolladores suelen determinar qué tipo de consultas necesitarán e implementar estructura de datos especiales para acelerar dichas consultas. Si se necesitan de más, utilizan las existentes para su implementación o inventan unas nuevas si se necesitan.

Ejemplos de estructura de datos especializadas son tablas hash o árboles binarios para buscar objetos por su id, objetos pre ordenados en listas enlazadas basados en criterios varios para cuando se precise buscar un objeto por determinados criterios e incluso guardarlos en una estructura de datos hash espacial para encontrar objetos dentro de una región o radio dados.

3.9 Actualización de objetos en tiempo real

El estado de un objeto de juego se puede definir por los valores de todos sus atributos en un determinado tiempo. Es aconsejable pensar en los estados como cambiantes continuamente y siendo muestreado por el motor de forma discreta.

Una simple forma que no suele funcionar para actualizar los objetos, es hacer que todos implementen una función *virtual update()* al que se le puede pasar algún dato como el tiempo transcurrido desde la última llamada y llamarlo en cada turno en todos los objetos, normalmente en una iteración del bucle del juego. Algunos objetos pueden personalizar las implementaciones de *update* para avanzar cálculos para el siguiente turno.

El problema puede entrar en cómo mantener la colección de los objetos a actualizar y de qué debe encargarse el método *update*. El método, además de actualizar su estado, podría necesitar de otros subsistemas que a su vez también tienen estados internos que necesitan ser actualizados. Una opción es que el método *update* también actualice los subsistemas. Sin embargo, podría resultar en muchos objetos actualizando muchas veces los mismos subsistemas. Amén a las restricciones que tienen de rendimiento por la cantidad de datos que manejan.

Muchos motores usan la técnica de actualizaciones en lotes. Es más eficiente actualizar muchas animaciones en un lote que ir objeto a objeto y actualizarlos junto a otras operaciones no relacionadas. Se suele actualizar los subsistemas en el bucle del juego directamente.

Cuando un objeto requiere el servicio de un subsistema concreto, pregunta por una simple instancia de información específica, por ejemplo, una instancia de triángulo a renderizar, el cual maneja y coloca como quiere, pero nunca controla cómo ha de renderizarlo. En su lugar, el subsistema maneja una lista de todas esas instancias dadas y una vez se han actualizado todos los objetos, el subsistema realiza su trabajo mediante una actualización de lotes eficiente.

Varias de sus ventajas incluyen máxima coherencia en caché, redución al minimo de las computaciones necesarias evitando duplicados, reduce la recolocación de los recursos (ya que durante las actualizaciones pueden necesitar memoria de forma temporal) y mayor eficiencia en *pipelining*.

3.9.1 Dependencias entre objetos y subsistemas

Incluso sin considerar el rendimiento, el enfoque inicial falla cuando un objeto depende de otro. Por ejemplo, un humano sosteniendo un gato. Para calcular la pose del gato, es preciso calcular la pose del humano primero, poniendo en relieve la importancia del orden de actualización. Lo mismo sucede cuando un subsistema depende de otro subsistema.

Una forma de resolverlo es escribir de forma explícita en el código el orden de actualización de los subsistemas. En los objetos es más peliagudo, ya que al actualizar un objeto, puede pasar que dependan de resultados intermedios entre subsistemas. Por ejemplo, un objeto puede requerir que se reproduzca la animación antes de actualizar el subsistema de animación. Pero puede necesitar también ajustar la pose intermedia dada por la animación para ser usada en otro

subsistema. Esto hace que dicho objeto tenga que actualizarse dos veces, una antes del sistema de animación y otra después.

Varios motores permiten actualizar los objetos en varios puntos distintos durante el frame. Se puede conseguir estableciendo distintos métodos *update* virtuales, uno para cada punto, como puede ser al inicio, al final o en el medio.

Se ha de usar con cuidado, al poder ser costoso iterar todos los objetos y llamando a funciones virtuales en cada fase de actualización y no todos los objetos requieren todas las fases, siendo un desperdicio de CPU. Por ello, se pueden usar distintas listas enlazadas para cada fase.

Otra variante, los *bucketed updates*, pueden mejorarse para permitir hilar mejor las reglas de actualización. Imaginemos que un objeto A necesita que un objeto B esté totalmente actualizado, algo que chocaría con el uso de actualizaciones en lote de la animación, que espera a que estén todos listos para actualizar todas las animaciones al mismo tiempo.

Las interdependencias se pueden visualizar como un bosque de árboles. Una solución para evitar esos problemas de conflicto es recolectar esos objetos en grupos independientes que podemos llamar *buckets* a falta de mejor nombre. El primer *bucket* se compondría de todas las raíces, el segundo de todos los objetos que están en el segundo nivel de todos los árboles, el tercero los del tercer nivel, y así.

Para cada *bucket*, se ejecuta una actualización completa de los objetos y los subsistemas, con todas las fases de actualización y se repite el procedimiento con los siguientes *buckets* hasta terminar. Huelga decir que la profundidad de los árboles cuenta, por lo que muchos motores limitan ésta y algunas desventajas es que puede que algunos subsistemas no puedan ser actualizados con este enfoque.

Centrándonos en el objeto en sí, nos podemos encontrar que antes y después del bucle de actualización, los estados de todos los objetos del juego son consistentes, pero durante, pueden ser inconsistentes. Dicha inconsistencia puede ser fuente de confusión y bugs y se complica cuando algún objeto consulta a otro objeto (dependencia) cuyo estado es inconsistente. Si el objeto precisa del estado anterior, no hay mucho problema, pero si necesita el nuevo estado, el objeto puede no estar actualizado y se presenta el problema del *one-frame-off lags* (lags de un frame fuera), siendo el estado del objeto A un frame por detrás del resto de objetos.

Una forma de solucionarlo es con la técnica del *bucket*, pero impone restricciones. Por ejemplo, si el objeto A quiere información actualizada de B, B es una dependencia, pero si también quiere la información previa de B, no podrá por las restricciones del *bucket*. Para añadir consistencia, se puede "cachear" el estado del objeto, es decir, guardar el estado anterior, y escribir el nuevo estado en lugar de sobrescribir, permitiendo con ello consultas seguras y consistencia. Otro beneficio con ello es que se puede realizar interpolación linear.

La desventaja principal es la duplicación de memoria además de resolver la mitad del problema, pues todavía puede sufrir de inconsistencia en el estado nuevo.

3.10 Eventos y paso de mensajes

Los juegos son inherentemente dirigidos por eventos. Un evento es cualquier cosa interesante que pasa durante el juego: una explosión, un enemigo que daña al jugador, un pack de medicinas que se ha recogido y más. El juego generalmente necesita una forma de notificar a los objetos interesados cuando ocurre algo y organizar la respuesta de dichos objetos ante el evento.

Una forma simple de notificar es llamar a un método del objeto interesado. Son funciones enlazadas de forma estática. El problema es que es poco flexible y obliga a casi todos los objetos del juego a implementar el método virtual que responde a los eventos tanto si les interesa o no e incluso puede obligar a implementar varios métodos distintos según los eventos. Dificultando además añadir nuevos eventos al sistema.

Lo que realmente se necesita son funciones enlazadas de forma dinámica. Se puede conseguir encapsulando la llamada a una función en un objeto, en el que guarda principalmente el tipo de evento y sus argumentos. Algunos motores los llaman mensajes o comandos y sus principales ventajas son reducir los métodos necesarios para los eventos al encapsular el tipo de eventos y sus argumentos, la persistencia y el reenvío de los eventos.

El tipo de los eventos puede definirse en una enumeración, pero implica conocer todos los posibles eventos del juego, además de su centralización y consecuente rotura de la encapsulación. También impide que nuevos eventos puedan definirse mediante el enfoque de dirigido por datos.

Otra forma usando *strings* permite nuevos eventos pero nuevos problemas como posibles conflictos de nombres, que no funcionen por un error de tipeo, más memoria requerida y mayor coste de comparación en relación con las enumeraciones. Algunos se pueden resolver con ids de *string hasheados* pero sigue habiendo posibles problemas por conflictos de nombres.

Los argumentos de los eventos proveen información acerca del evento y se puede implementar de varias formas:

- Heredar de la clase Evento para cada tipo único de evento y los argumentos guardarlos como atributos de clase.

- Guardarlos como una colección de variantes. Los variantes son datos de objetos que pueden guardar más de un tipo distinto de dato en una variable. En C++, *union* es una variante.

Uno de los problemas que surgen con los argumentos, es que tanto emisor como receptor deben conocer el orden de los argumentos pasados, fuente de bugs. Para solventar ese problema se puede implementar los argumentos como pares clave-valor. Cada argumento es conocido por un id único.

Una vez implementado los eventos y recibidos, se necesita responder a ellos de alguna forma, el cual suele estar implementado en un trozo de código script o función. A menudo la función es un método virtual que acepta cualquier tipo de evento y dentro del método incluye un *switch* o un *if/else* en cascada para manejar aquellos tipos de evento que le interesan.

Teniendo en cuenta que los objetos tienen dependencias, nos podemos encontrar con un vehículo que reciba un evento, y probablemente tenga que pasar dicho evento a los pasajeros del vehículo e incluso en los pasajeros podría ser conveniente pasarlo a los objetos de sus inventarios. O simplemente hacer *multicast* como puede ser en una explosión pasar dicho evento a todos los que estaban en su radio.

La técnica de pasar los eventos es un patrón de diseño común, denominado *chain of responsibility*. Normalmente el orden en el que el evento es pasado, es determinado por los ingenieros. El evento es pasado al primer objeto de la cadena de responsabilidades y el método devuelve un *boolean* o un código indicando si ha consumido el evento y no se pasa a otros, o se pasa a otros.

Hacer *multicast* o *broadcast* puede ser muy ineficiente ya que la mayoría de objetos no están interesados en la mayoría de eventos que hay, por lo que resulta buena idea permitir a los objetos registrar su interés en determinados eventos.

Para ello se puede mantener una serie de listas para cada tipo de evento o hacer algo similar. Con ello podemos evitar llamadas a los métodos virtuales, que pueden suponer operaciones no triviales.

Por último, una última característica de los eventos es permitir que los eventos sean encolados para mandarlos en un futuro arbitrario. Tiene sus ventajas pero aumenta la complejidad del sistema y conlleva algunos problemas.

Entre los beneficios se encuentran el control de los eventos, de la misma manera que el orden importaba en la eficiencia cuando actualizábamos los objetos, pasa lo mismo en ciertos eventos durante el bucle de juego. Igualmente, postergar los eventos puede ser una característica interesante. Esto último se consigue marcando el tiempo en el evento. Para ello es conveniente ordenar la pila.

Incluso habiendo ordenado los eventos, el orden es ambiguo si dos de ellos coinciden, algo más frecuente de lo que se piensa. Una forma de resolverlo es asignando prioridades a los eventos y ordenar la pila por tiempo y por prioridad.

Los problemas que se pueden encontrar con la pila es el incremento de la complejidad, necesidad de hacer una copia profunda del evento y por tanto asignación dinámica de memoria y dificultades para la depuración.

Pero no por ello la entrega de eventos inmediata carece de problemas. Pueden crear pilas de llamadas muy profundas.

4 Desarrollo

4.1 Estudio de C++

Cuando se decidió por realizar este proyecto, se estableció el uso de C++ como principal lenguaje de programación debido a su amplio uso en los motores de videojuegos y en los videojuegos en sí mismo, y su buen futuro en cuanto a uso de dicho lenguaje en el campo orientado a los motores.

Por ello, se procedió a aprender el lenguaje y aplicarlo en el mini proyecto del tetris que serviría también para un primer contacto con SDL y posteriormente con

SFML. Los tutoriales y recursos leídos y utilizados para tal fin fueron muy variados, incluyendo aquí los principales y omitiendo algunos esporádicos:

- Tutorial de la página oficial de C++, página también utilizada para la consulta de métodos y apis del lenguaje. (20)
- Stackoverflow para alguna consulta concreta. (21)
- Tutorial de SDL versión 1.2 hecho para C++ (22) en Open Libra (23)
- Como programar en C/C++ y Java, 4ta Edición, Deitel, Editorial Pearson, Prentice Hall (24).
- Un cursillo de coursera, el cual no se completó satisfactoriamente por tener un nivel más elevado de lo esperado (25).
- Programación y diseño en C++. Introducción a la programación y al diseño orientado a objetos James P. Cohoon Jack W.Davidson (27).
- C++ por Bjarne Stroustrup (27)

Para poner en práctica los conocimientos de C++ al mismo tiempo que se familiarizaba con las librerías que se fueran a utilizar y la estructura genérica de los juegos y los motores de videojuegos, se decidió hacer un simple juego de Tetris y reproducir el proyecto ofrecido por SFML.

Inicialmente se empezó con el Tetris y la librería SDL (28), y posteriormente, con el Tetris casi terminado, se cambió a la librería SFML por motivos que se detallarán más adelante. Para el estudio y profundización de la librería SFML (29) y perfeccionamiento del C++ en su uso en el ámbito de los juegos, se siguió el libro SFML Game Development, por Laurent Gomila (30), el cual permitió ahondar en características más avanzadas de C++ en su estándar C++11 mientras se realizaba de forma guiada un juego completo desde cero, con lo que al finalizar el aprendizaje de C++ se tenía dos juegos en los que se usaban diferentes características de C++.

4.2 Uso de SDL y SFML

Para la realización del motor, se precisaba una librería o framework multimedia sobre el que apoyarse para que se encargara de tareas de hardware como la comunicación con los dispositivos de entrada, el audio y los gráficos. Y preferentemente multiplataforma.

La librería SDL (*Simple DirectMedia Layer*) es una librería multiplataforma que permite acceso a bajo nivel al audio, dispositivos de entrada como teclados, ratones y joysticks, y al hardware gráfico ya sea a través de la librería o usando liberías como OpenGL y Direct3D. Igualmente, está escrito en C y trabaja de forma nativa en C++, lo que le da un punto más por la integración con el lenguaje seleccionado para trabajar. Se ha usado para realizar juegos. (28)

El motivo principal de escoger dicha librería es que ya se disponía de conocimientos sobre la librería al haberlo utilizado en otras ocasiones y es una librería con el que se tenía comodidad y confianza.

Posteriormente, hacia el final de la realización del juego Tetris, se realizó un cambio. La constante aparición de la librería SFML junto a la librería SDL en comparativas, opiniones y preguntas hacían ver que ambas librerías eran muy similares, siendo más ventajosa SFML por estar en C++, más actualizado y tener mayor enfoque a juegos.

Por ese mismo motivo, se ofreció a SFML una oportunidad que se le había negado para no tener que aprender otra librería conociendo SDL. Al ser librerías similares, se pudo aprender con rapidez y adaptar en apenas dos días o menos el código del Tetris realizado en SDL a SFML, con un comportamiento similar y más sencillo. Igualmente, se ha podido comprobar su ventaja en tener mayor enfoque en los juegos y la orientación a objetos.

La comunidad y los códigos y ejemplos disponibles para SFML junto a lo anteriormente mencionado, hacen que se decida cambiar de librería y usar para el proyecto la librería SFML. (29)

4.3 Herramientas y librerías usadas

Para la realización del proyecto, se utilizaron una serie de herramientas y librerías. Entre ellas, podemos contar con las herramientas del entorno de trabajo:

- Netbeans versión 8 (31) con su plugin para C/C++. Un completo IDE que originalmente se usaba para Java y actualmente soporta varios lenguajes más, entre ellos C/C++. Es la herramienta personal preferida para la programación y su correcto soporte para C/C++ hace innecesario buscar herramientas adecuadas para la codificación, compilación y ejecución del proyecto.

- Git y Gitlab. (32) Aunque el proyecto se ha realizado en solitario, el disponer de más de un ordenador y programar en más de un sitio, hace ideal el uso de un repositorio para mantener en todo momento la última versión del proyecto a desarrollar. Junto con las bondades de disponer del control de versiones, historiales y control de bugs entre otras cosas, se ha instalado un servidor personal y un gestor de repositorios para git llamado Gitlab. No hay un motivo especial en usar esta solución más que el aprovechar el servidor del que se disponía y el control que supone ser el administrador del gestor de repositorios.

- GCC. En el entorno de trabajo se usa Linux, por cuestiones de preferencia personal, el cual incluye GNU Compiler Collection, más conocido como GCC, un compilador estándar entre los cuales incluye a C y C++, por lo que no se precisó de nada más. (33)

- Doxygen. Para la generación de documentación del código fuente. (34)

Disponiendo ya del entorno de trabajo configurado, se procedió al resto de herramientas específicas al proyecto, las cuales se incluyen:

- SFML. Librería multimedia sobre el que se basará el motor.

- Tiled Map Editor. Una aplicación que permite elaborar mapas y definir polígonos a partir de una imagen o más a ser usada como tileset (conjunto de tiles (35), es la parte gráfica de cada videojuego que puede ser usada para completar partes de un fondo). Para su aprendizaje se ha usado el siguiente tutorial http://gamedevelopment.tutsplus.com/tutorials/introduction-to-tiled-map-editor--gamedev-2838 (36) Con este editor se pretende realizar los mapas necesarios para probar el motor. (37)

- TinyXML 2. Ante la decisión de querer hacer el motor dirigido por datos, se iba a tener que leer de entrada. XML es un lenguaje bastante adecuado y se seleccionó esta librería para el parseo de ficheros xml debido a su sencillez, ligereza y sobre todo a que ya se tenían conocimientos de dicha librería al usarlo en otros proyectos con éxito. (37)

4.4 Planificación y estudio económico

Independientemente del tipo de proyecto, es aconsejable realizar una planificación temporal para controlar el avance del proyecto, los riesgos y los posibles contratiempos que surjan. De igual manera, realizar un estudio económico para comprobar su rentabilidad y controlar los costes. En este proyecto, dada su envergadura no es una excepción.

4.4.1 Planificación

Para este proyecto, al disponer de una fecha límite en el que obtener los resultados del proyecto, se ha decidido seguir el siguiente enfoque:

- Desglosar el proyecto y su objetivo en varias tareas y objetivos.

- Establecer la importancia de los objetivos y sus interrelaciones (ej. un objetivo precisa que otro objetivo sea cumplido para poder iniciarse)

- Seleccionar los objetivos más importantes para establecer una fecha límite

- Desglosar el tiempo disponible para el proyecto y establecer el tiempo necesario para cada objetivo.

- Asignar las fechas límites a los objetivos seleccionados

- Al principio del proyecto y en cada fecha límite, establecer una pequeña planificación para obtener el siguiente objetivo a realizar teniendo en cuenta el éxito/fracaso de los objetivos anteriores y la fidelidad de la estimación realizada, gestionando todos los posibles riesgos que puedan surgir.

Con ello en mente, las partes en las que se puede dividir el proyecto inicialmente en la siguiente tabla y considerando un día como cuatro horas. También se tiene en cuenta que varias de esas tareas se solapan.

Tabla 1. Planificación inicial

Tarea	Especificación	Duración
A	Estudio e investigación de la teoría necesaria para el acometimiento del proyecto	43 días
B	Preparación del entorno de trabajo	3 días
C	Análisis de los requisitos	11 días
D	Diseño de la codificación	24 días
E	Codificación	60 días
E.1	Estados y menús	10 días
E.2	Gráficos	34 días
E.3	Entidades	35 días
E.4	Colisiones	24 días
F	Diseño gráfico y de audio	16 días
G	Codificación de datos	22 días
H	Testeo de la aplicación	12 días
I	Lanzamiento (prueba final, empaquetado y documentación)	26 días

Calculando tiempos y colocando las tareas y fechas límites, se ha optado por un simple diagrama de Gantt para visualizar la planificación realizada, la cual sufrirá modificaciones y puntualizaciones cada vez que se empiece o termine alguna de las tareas que se han establecido.

La planificación hecha inicialmente ha sido la mostrada en la figura 4.1. Hacia el final del proyecto, el tiempo invertido ha sido al final la que se observa en la figura 4.2 y tabla 4.2, en el que se pueden ver ciertas modificaciones sustanciales sobre todo a la hora de dividir el proyecto en partes y la duración de cada una de ellas.

Se debe sobre todo al desconocimiento de los detalles del proyecto y el esfuerzo que realmente requerían. Por otro lado, se había seguido un esquema muy clásico de ir por partes e ir finalizándolos, cuando en realidad están muy mezclados y se realizan de forma conjunta en general, hasta el punto de que algunas partes empezaban pronto, realizando algunas cosas y se pausaban hasta más adelante cuando se hacía la otra mitad del trabajo.

Tabla 2. Planificación final

Tarea	Especificación	Duración
A	Estudio e investigación de la teoría necesaria para el acometimiento del proyecto	54 días
B	Preparación del entorno de trabajo	3 días
C	Análisis de los requisitos	11 días
D	Diseño de la codificación	22 días
E	Codificación	121 días
E.1	Codificación estados	14 días
E.2	Codificación gráfica	65 días
E.3	Codificación de entidades y propiedades	45 días
E.4	Codificación de parseo de datos	108 días
E.5	Codificación de animaciones	45 días
E.6	Codificación de colisiones	44 días
F	Testeo	122 días
F.1	Realización de niveles	14 días
F.2	Diseño gráfico	14 días
F.3	Realización de niveles II	14 días
F.4	Diseño gráfico II	14 días
F.5	Diseño y codificación de datos	122 días
G	Lanzamiento (prueba final, empaquetado y documentación)	15 días

Por otro lado, considerando que en este proyecto se incluyen las horas de estudio y preparación, en un proyecto similar posterior, dichos conocimientos y práctica ya se tienen, por lo que podría reducirse el tiempo en tranquilamente dos meses del total.

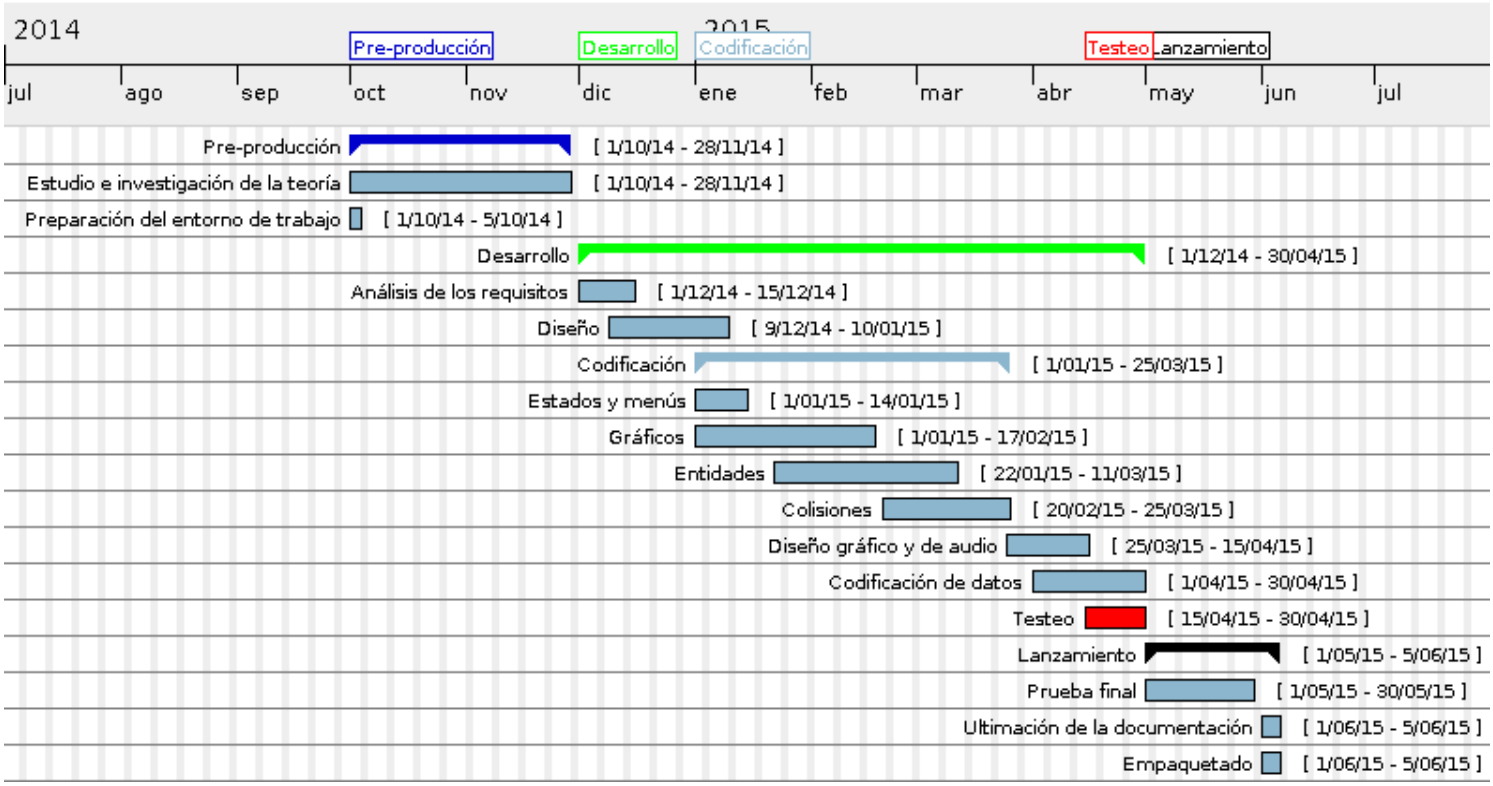


Figura 4-1. Planificación inicial

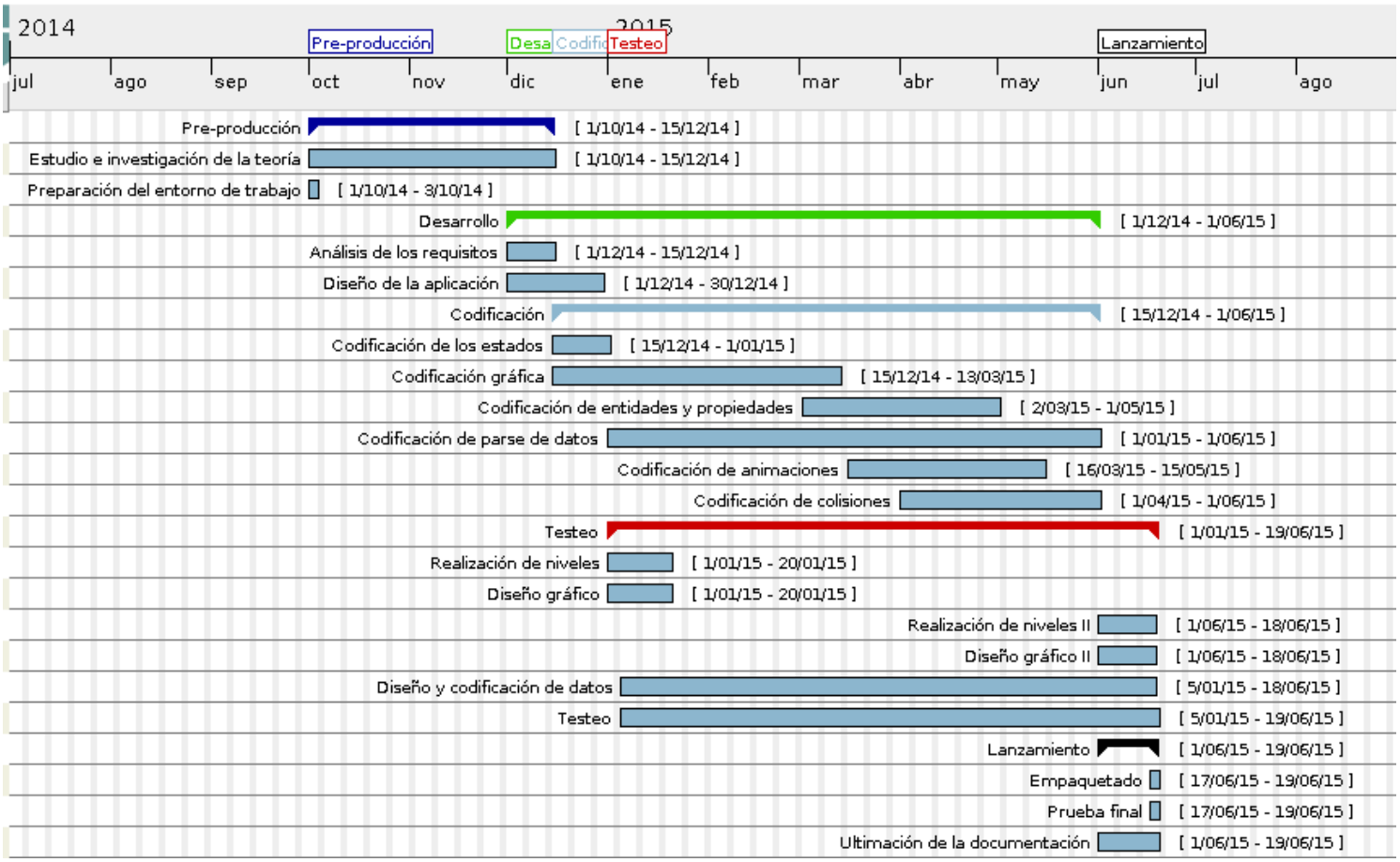


Figura 4-2. Planificación final

4.4.2 Estudio económico

Se han hecho unos cálculos aproximados de lo que costaría llevar a cabo el proyecto, incluyendo materiales y tiempo de aprendizaje que posteriormente en proyectos similares no incurrirían en gastos extras al ser ya amortizados.

Tabla 3. Tareas de desarrollo del proyecto y costes

Código	Unidad	Nombre unidad	Nº unidades	Precio(€)	Importe(€)
1.1	Horas	Estudio e investigación	45	70	3.150
1.2	Horas	Análisis de los requisitos	30	70	2.100
1.3	Horas	Codificación	1.335	70	93.450
1.4	Horas	Testeo	30	70	2.100
				Total	101.160

Tabla 4. Materiales empleados

Código	Unidad	Nombre unidad	Nº unidades	Precio(€)	Importe(€)
2.1	Ud.	PC (Procesador Intel i7 a 4.0Ghz, 6GB RAM y tarjeta gráfica a 2GB)	2	900	1800
2.2	Ud.	Distribución Linux	2	0	0
				Total	1800

Tabla 5. Presupuesto

Capítulo	Unidad	Resumen	Precio(€)	Precio (Letra)
1	Ud.	Mano de obra	101.160	Ciento un mil ciento sesenta euros
2	Ud.	Material empleado	1800	Mil ochocientos euros

Total presupuesto de ejecución material	102.960 €	Ciento dos mil novecientos sesenta euros

4.5 Codificación

4.5.1 Requerimientos

Los requisitos funcionales para la aplicación no se definen de forma estricta, debido al carácter de estudio que se tiene en este proyecto. Sin embargo, se han definido unos requisitos ideales a conseguir, de forma que se pueda tocar varias características básicas de un juego, que los motores han de implementar. Dichos requisitos ideales son:

R1. Disponer de una pantalla inicial donde escoger si empezar un nuevo juego, cargar un juego, ir a opciones o salir de la aplicación.

R2. Disponer de una pantalla para modificar opciones.

R3. Disponer de una pausa con menú durante el juego.

R4. En la pausa del juego se debe permitir guardar el juego y salir del juego al menú principal.

R5. El jugador debe tener una serie de misiones y poder visualizar durante el juego el estado de las mismas.

R6. Se debería poder determinar detalles del juego a través de ficheros externos a la aplicación, eliminando la necesidad de recompilar cada vez que se haga un cambio.

R7. Se pueden reproducir sonidos en diversas ocasiones

R8. Durante el juego, se tienen los siguientes requisitos:

R9. El jugador debe poder moverse y el mapa o vista moverse acorde al jugador.

R10. El jugador debe poder interactuar con los elementos, idealmente, debería poder hablar, colisionar, ser dañado y dañar, entre otras acciones.

R11. El jugador puede moverse entre niveles ante una determinada acción (por ejemplo, entrar a una determinada zona)

R12. Pueden suceder determinados eventos, como una nueva misión, cuando el jugador entre en alguna zona clave.

4.5.2 Casos de uso

Al ser un motor de videojuego, cuya interacción con el usuario es jugar, los casos de uso del proyecto son bien sencillos, reducidos a la imagen 4.3.

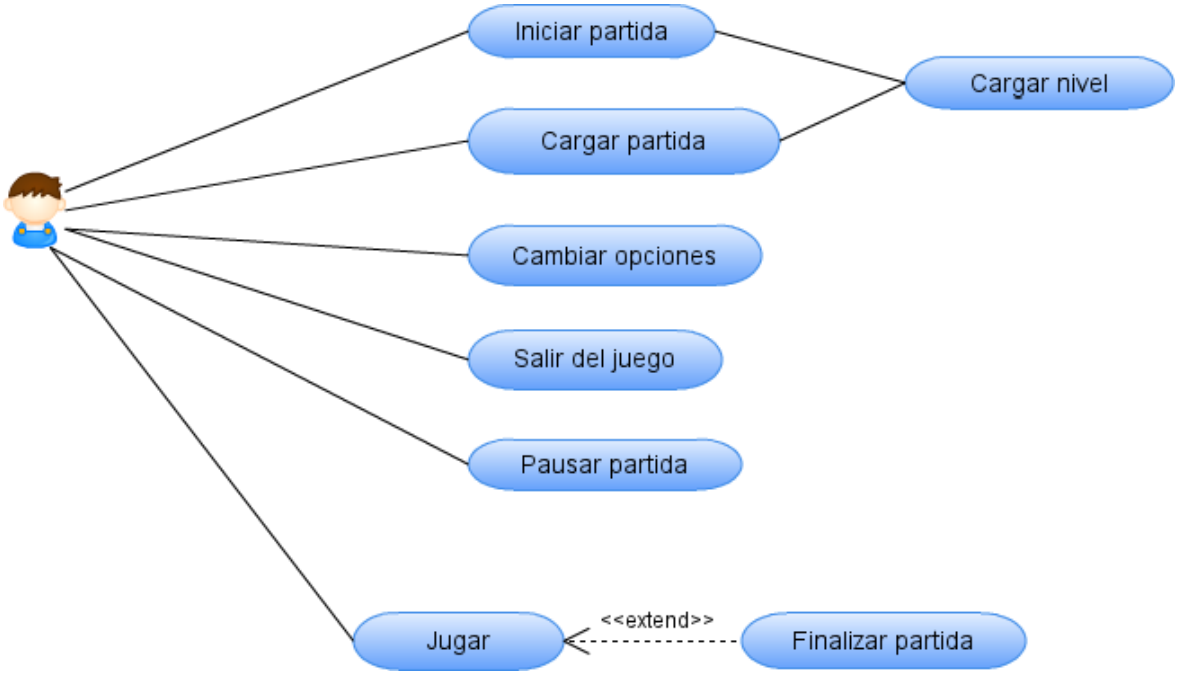


Figura 4-3. Caso de uso

4.5.3 Datos guardados

Entre las pautas dadas, están el uso de ficheros externos para guardar datos. En la estructura de la aplicación, los datos se guardan bajo la carpeta Media. También se consideran datos los recursos multimedia, que quedan guardados bajos las carpetas *Textures*, *Fonts*, *Music* y *Sound* según el tipo de recurso, todas bajo la carpeta principal Media. Dichos recursos se han obtenido de la web http://opengameart.org/ (39) con licencias libres o en su caso con reconocimiento de la autoría de las imágenes.

Uno de los formatos aconsejados para guardar datos, es el xml, por lo que el resto de datos se han guardado en dicho formato y referencian al resto de recursos, ya sean otros xml o recursos multimedia. Dichos xml quedan bajo la carpeta Data.

Una excepción son los xml de niveles, que quedan en una carpeta aparte llamada *Levels*.

4.5.3.1 Mapas

Para generar los datos del mapa y guardarlos en un xml, se ha usado el programa Tiled (37), un editor de mapas de recuadros, en el que a partir de imágenes con los sets de recuadros, se puede generar un xml con el mapa creado.

Para ello, se han seguido algunos tutoriales (36) (40) (41) acerca de cómo guardar y manejar mapas en información xml.

Siguiendo el esquema xml que genera el programa, se crean dos capas, una con el fondo principal y otro con algunos detalles más del suelo. Igualmente se crea otra capa en el que se guardan los objetos que actúan como entidades colisionables, dando un nombre que distingue entre distintos tipos.

En el mismo fichero, se incluye una relación de nombres xml referenciando tanto al personaje como al resto de entidades del mapa.

Ejemplo de xml para un nivel/mapa.

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
	<character id="Character.xml" />
	<villager id="Aldeano.xml" />
	<villager id="Enemy.xml" />
</people>
<map version="1.0" orientation="orthogonal" renderorder="left-up" width="50" height="50" tilewidth="32"
tileheight="32" nextobjectid="2">
	<tileset firstgid="1" name="TileSet" tilewidth="32" tileheight="32">
<image source="Media/Textures/TileSet.png" width="1024" height="1024"/>
	</tileset>
	<layer name="Underground" width="50" height="50">
	<data>
	<tile gid="0"/>
	<tile gid="0"/>
	<tile gid="0"/>
	<tile gid="0"/>
	<tile gid="354"/>
	<tile gid="0"/>
	</data>
	</layer>

	<layer name="Background" width="50" height="50">
	<data>
	<tile gid="293"/>
	<tile gid="293"/>
	<tile gid="293"/>
	<tile gid="293"/>
	<tile gid="639"/>
	</data>
	</layer>

	<objectgroup name="Capa de Objetos 1">
	<object id="86" x="698" y="343.75" name="Hole" type="Hole">
	<polyline points="0,0 -3.75,-23.25 -13.75,-42.5 -22,-47 -39,-48.75 -61.75,-45.5 -78.25,-25 -
	78.5,4.25 -61.25,26.25 -18.75,26.75 -0.25,0.75"/>
	</object>
	<object id="87" x="663.333" y="1216.33" name="Block" type="Block">
	<polyline points="0,0 -114.333,-118.667 -221,-119.667 -236.667,-99.3333 -237.667,-54"/>
	</object>
	<object id="113" x="1599.67" y="1450.33" name="ChangeLevel" type="ChangeLevel"
level="level2.xml">
	<polyline points="0,0 -32,0 -32.3333,77 0,77.3333"/>
	</object>
	</objectgroup>
</map>
```

De todo el código generado, y que se encuentra retocado para dejarlo con esta estructura, lo importante son los atributos *width*, *height*, indicando cuántos

recuadros de ancho y alto tiene el mapa y los atributos *tilewidth* y *tileheight* indicando el tamaño de un recuadro de imagen.

Los nombres *Underground* y *Background* distinguen las capas de detalle y de fondo respectivamente, en el que cada tile es uno de los recuadros del mapa generados por el programa.

Por último, todos los objetos del mapa quedan bajo la etiqueta *objectgroup*. Cada objeto en la etiqueta *object*, con una posición absoluta, un nombre y un tipo que distingue qué tipo de objeto es y una lista de puntos relativos a la posición, generados por el programa.

Los tipos de objetos que hay, son, por el momento, *Hole*, *Block* y *ChangeLevel*, representando respectivamente un agujero, un bloque que no se puede atravesar y un cambio de nivel cuando se activa.

4.5.3.2 Entidades

A la hora de definir una entidad, se define en un xml los siguientes datos:

- Propiedades de la entidad. En él se define el identificador numérico, el cual debe ser único entre las entidades, y una serie de propiedades identificadas por el nombre de la propiedad. Según la propiedad, puede requerir atributos extras.

```
<Entity id="3">
<Properties>
        <Property id="Position" x="500" y ="500" />
        <Property id="MaxVelocity" value="100" />
        <Property id="Drawable" numberAnimations="8"/>
        <Property id="Collision" />
        <Property id="Talk" value="Hola chico, que te trae por aquí."/>
</Properties>
</Entity>
```

- Colisiones. Si dispone de la propiedad de colisión, se añaden los mismos datos que en las colisiones del mapa.

```
<Collision>
        <object type="Block" x="0" y="0"><polyline points="2,16 2,32 30,32 30,16"/></object>
</Collision>
```

- Máquina de estados finitos. Consta de varias transiciones las cuales definen una máquina de estados finitos para las animaciones de la entidad si éste tiene la propiedad de ser dibujado. Es condición necesaria que los estados comiencen en 0 y coincidan con el orden en el que se ponen las animaciones. Igualmente, la entrada coincide con el orden de la enumeración Actions.h.

```
<StateMachine numberStates="8">
        <Transition state="0" entry="1" newState="7"/>
        <Transition state="0" entry="2" newState="6"/>
        …
</StateMachine>
```

- Si se define la propiedad de ser dibujado, es necesario disponer al menos de una animación. Se debe tener el mismo número que el especificado en los estados de la máquina finita. Cada una de las animaciones puede disponer de más de un frame.

```
<Animation id="character-down-idle" idImage="player.png" replay="false">
<frame>
        <vertice value="32"/>
        <vertice value="0"/>
        <vertice value="32"/>
        <vertice value="32"/>
</frame>
</Animation>
```

4.5.3.3 Vida

De forma más especial, se ha incluido un xml concreto para dar imagen a la barra de vida de los personajes con el mismo formato que las animaciones de las entidades.

4.5.4 Patrones y estrategias

Durante el desarrollo, se ha seguido las pautas y estrategias de código que se ven en la industria del videojuego, para ello se ha seguido tutoriales y códigos ofrecidos por diversas fuentes como son la wiki del proyecto SFML (42), el libro Game Engine Architecture (1), Genbeta Dev (43), Game Dev Stack Exchange (43) y Game Coding Complete (9) entre varios otros.

Pasamos a comentar aquí dichas estrategias y algoritmos. Una parte del código se encuentra inspirado o copiado de diversas fuentes, entre ellas la wiki oficial de SFML (42) y el trabajo de varias personas en sus códigos públicos (43) (44) (45), entre otros, que se encuentran debidamente referenciados en el código fuente.

4.5.4.1 Flujo general

La aplicación comienza en el main, que enseguida delega en la clase Application. Ahí, se crea el contexto de la aplicación, donde se mantendrán y obtendrán los recursos que la aplicación maneja. Igualmente se crean e inicializan los sistemas del motor.

Los estados del juego también son creados al crearse el contexto, cada uno de los estados es creado, y se comienza por el estado inicial. Durante el bucle de la aplicación, la clase mantiene un bucle controlando los tiempos para los fps, y llama en el siguiente orden, los métodos processInput, update y render de la pila, el cual delega dichos métodos a los estados de la pila desde la parte superior a la inferior hasta que se acaban o hasta que alguno indica que se ha finalizado y que no vaya al resto de estados.

A partir de ahí, cada estado define el comportamiento de la aplicación. Estados sencillos como el menú, se limitan a dibujar, moverse por los componentes y reaccionar cambiando opciones o de estado. Cada vez que se solicita cambios de estados, se guarda en una pila de cambios pendientes en la pila de estados y se aplican una vez se actualiza o se procesa la entrada.

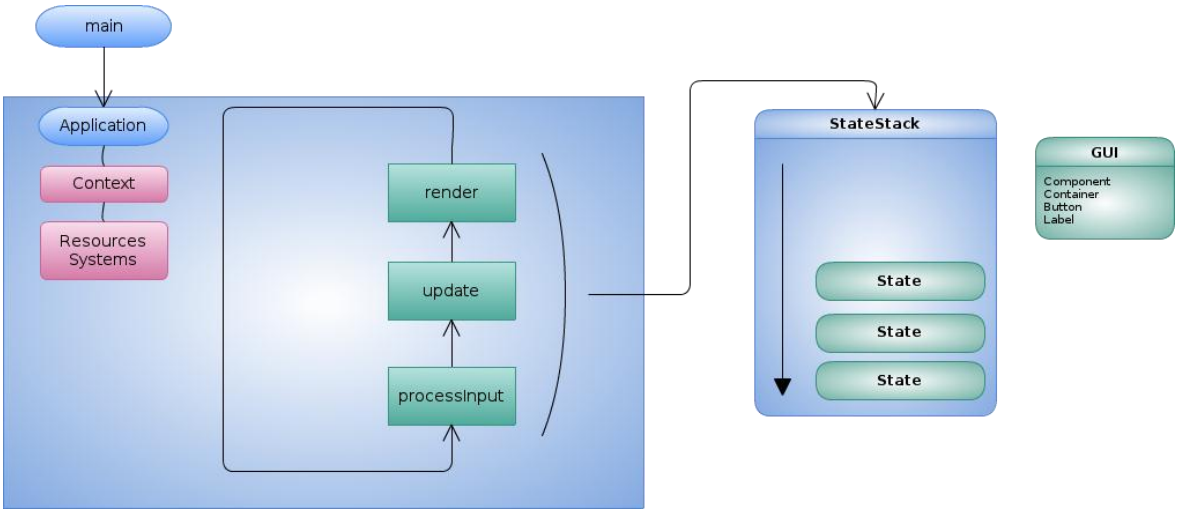


Figura 4-4. Flujo general de la aplicación

Para pasar al estado de juego, se pasa por el estado de cargando, es decir el LoadingState. En él, se crea una tarea paralela que haciendo uso de la clase LoadingLevel, se encargará de cargar en memoria los recursos necesarios y preparar en definitiva el nivel. Una vez finalizado, se pasa al estado de juego.

En el estado de juego, simplemente delega las funciones de actualización, procesado de entrada y renderizado, al nivel actualmente cargado en el contexto de la aplicación. El único control que realiza, es la finalización del nivel, tras el cual cambiará a otro estado.

El nivel obtiene del gestor de sistemas del contexto, del cual saca todos los sistemas que necesita. Para el renderizado, delega la función en el sistema de gráficos, para el procesado lo delega al subestado de juego si existe o hace uso de la clase Player, que controla la entrada y guardar el comando generado en una pila.

En la actualización, realiza las actualizaciones de los sistemas, controlando el orden y algunas acciones concretas como comprobar colisiones, llamando al método necesario en el sistema de colisiones.

El orden seguido es el siguiente:

- Si hay subestado, delega la actualización, sino:
- Realiza la ejecución de comandos del jugador delegándolo en el sistema de comandos.
    - Actualiza el sistema de movimientos
    - Actualiza el sistema de colisiones
    - Comprueba colisiones delegándolo en el sistema de colisiones

    - Resuelve las colisiones delegándolo en el sistema de colisiones
- Actualiza el sistema de gráficos
- Actualiza el sistema de gráficos en una segunda vuelta
- Actualiza el sistema de misiones
- Actualiza el sistema de objetos

Por último, si no hay subestado, captura la entrada en tiempo real del jugador y lo procesa.

En algún momento, alguno de los subsistemas puede modificar un subestado (ya sea añadiéndolo o quitándolo. De ser así, con el patrón observador, se ejecuta el método adecuado para controlar dichos cambios.

4.5.4.2 Estados del juego

La aplicación pasa por una serie de estados o pantallas diferentes, entre los que se pueden distinguir la pantalla inicial, la pantalla del juego propiamente dicho, la pantalla de las opciones, la pantalla de pausa con su menú asociado… El enfoque seguido para controlar y modular los distintos estados es mediante una pila de estados.

Un estado es una pantalla independiente, un objeto que encapsula la lógica y los gráficos de un determinado grupo de funcionalidad e información. Algunos de dichos estados ocuparán toda la pantalla, mientras que otros corren junto con otros estados y dibujando en la misma pantalla.

El manejo de los estados se va a parecer mucho a las máquinas de estados finitos. En la figura 4.5 se puede ver un simple ejemplo de cómo irían los estados. Decimos que se parece mucho debido a que estados como el de pausa, no ocupa toda la pantalla y se ejecuta junto con el estado de juego, con lo que la definición de Máquina de estados finitos no encaja perfectamente.

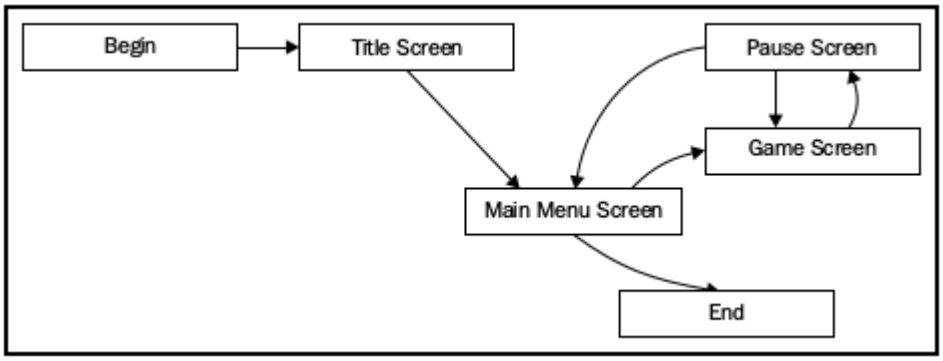


Figura 4-5. Máquina de estados finitos

La clase State es la clase base sobre el que se especializan los distintos estados, y permite ser dibujado en pantalla, procesar él mismo la entrada del usuario y ser actualizado en el tiempo. La clase StateStack apila los estados, y el estado activo

es el que se encuentra en la parte superior. Implementa los mismos métodos de dibujado, gestión de la entrada y actualización que los estados, de forma que se llaman en la pila como si de un estado se tratara, y la pila de estados se encarga de llamarlos a su vez en los estados.

Esto permite que si alguno de los estados no ocupa toda la pantalla o va conjuntamente con otros estados, los mismos métodos de dibujado y de procesado de la entrada del usuario notifican con los valores de retorno esta situación y la pila procede a llamar a los métodos del siguiente estado de la pila.

```
class State {
public:
	...
	virtual void draw()=0;
	virtual bool update(sf::Time delta)=0;
	virtual bool handleEvent(const sf::Event& event)=0;
	virtual void pushedAction() { //nothing by default
	};
	…

class StateStack : private sf::NonCopyable {
public:
...
	void StateStack::update(sf::Time dt) {
		// Iterate from top to bottom, stop as soon as update() returns false
		for (auto itr = stack.rbegin(); itr != stack.rend(); ++itr) {
			if (!(*itr)->update(dt))
			break;
		}
		applyPendingChanges();
	}

	void StateStack::draw() {
		// Draw all active states from bottom to top
		for (std::vector<State*>::iterator it = stack.begin(); it != stack.end(); ++it) {
			(*it)->draw();
		}
	}

	void StateStack::handleEvent(const sf::Event& event) {
		// Iterate from top to bottom, stop as soon as handleEvent() returns false
		for (auto itr = stack.rbegin(); itr != stack.rend(); ++itr) {
			if (!(*itr)->handleEvent(event))
			break;
		}
		applyPendingChanges();
	}
...
```

Por último, la pila tiene los mismos métodos que se esperarían de una pila normal. Cuando se crea un nuevo estado, se apila, y cuando el estado ha finalizado, se extrae de la pila.

Como detalle final, los estados son creados en el momento de arranque de la aplicación para una mayor rapidez en los cambios. Sin embargo, no todos los estados pueden crearse en primera estancia, por lo que disponen de un método para ser inicializados, el cuál es llamado desde la pila cuando son colocados.

De forma similar, se tiene otro método para cuando son desapilados, en caso de ser necesario alguna acción específica cuando el estado termina, como puede ser parar la música.

4.5.4.3 Patrón Observer

Dentro del estado del juego, existen unos subestados, como puede ser la conversación, misiones o inventario. Dichas acciones del juego, son pantallas bien definidas e independientes a excepción de una dependencia, los datos del juego.

Los estados quedan cargados al iniciar la aplicación y los datos que utilizan, son los datos compartidos en el contexto (clase Context) para acceder a recursos, a la ventana de la aplicación y a los sistemas del motor.

Son recursos genéricos y aunque se ha metido también una referencia al nivel actual en el contexto, los subestados necesitan de una interacción íntima con el nivel y sus datos, por lo que para no exponer todo el contenido del nivel al resto del código ni sobrecargar el uso del contexto, se decidió hacer como subestados dentro del estado de juego.

Además, la razón principal es que el cambio de los estados se realiza desde el propio estado, sin embargo, los cambios del juego se conocen y hacen dentro del propio nivel, no en el estado, por lo que para una mayor limpieza en los estados, se ha seguido otro enfoque.

Dichos subestados pasan a implementar otra interfaz muy similar, SubStateGame, y a estar controlados por el nivel (clase Level). Se tienen tres principales clases para implementar el patrón observer (48), Subject, Message y Observer.

El observer (observador) es el nivel, que vigila los cambios, y ante un cambio, reacciona cambiando, poniendo o eliminando un subestado.

El Subject es el objeto que se vigila. Mantiene una lista de sus observadores, y cuando sufre cambios, avisa a dichos observadores. Es decir, avisa al nivel.

El Message es una clase que guarda información sobre el nuevo subestado a cambiar y algún otro dato útil. Es la variable que cambia subject y la información que utiliza el nivel para reaccionar.

El sistema que controla los objetos añade el subject a todas las entidades, de modo que cualquier acción o reacción pueda provocar un cambio de subestado, como puede ser ante el botón del jugador o cuando el personaje golpea o entra en una zona.

4.5.4.4 Grafo de escena

Cuando se tienen varios elementos o entidades a dibujar en la escena, un simple dibujado secuencial de los elementos puede ser suficiente. La mayoría de las veces no es así, ya que dificulta tener elementos con una posición relativa a otro o incluso dibujar según la profundidad, tapando de forma parcial unos elementos encima de otros. (30)

Para ello se desarrolla un grafo de escena, una estructura de datos en árbol consistente en nodos llamados nodos de escena (clase SceneNode). Cada nodo puede guardar un objeto o entidad a dibujar.

Cada nodo puede tener cualquier cantidad de hijos, los cuales adaptan su posición, rotación y escalado en función de su nodo padre cuando son renderizados. Estos nodos, cuando guardan su posición, rotación y escalado, lo guardan de forma relativa al padre, no de forma absoluta.

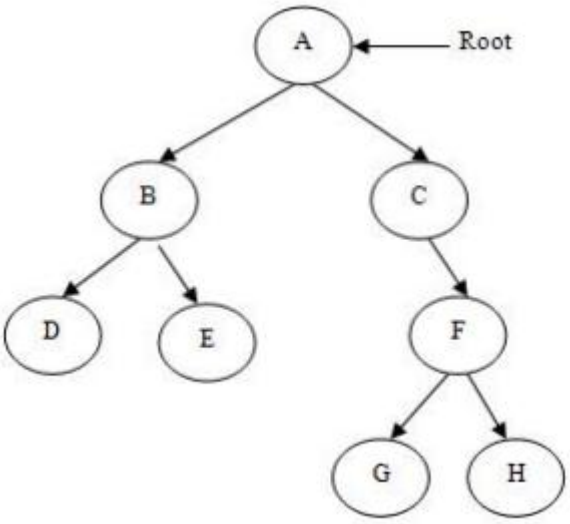


Figura 4-6. Estructura de árbol

Por último, dicho árbol contiene un único nodo raíz, que sólo existe una vez en el mundo virtual.

La clase SceneNode representa un nodo base para el resto de nodos, el cual es responsable de su propio tiempo de vida y destrucción y guarda una lista de nodos hijos. Cuando el nodo es destruido, todos los nodos hijos se destruyen.

Como muchas estructuras de datos en árbol, los nodos incluyen los típicos métodos para añadir nuevos nodos y borrarlos. Además, incluyen los métodos para dibujar el nodo y actualizar el nodo.

El método de actualizar el nodo se sobrescribe cuando es necesario actualizar el objeto o entidad a través del tiempo, como pueden ser las animaciones o la posición del nodo.

A partir de esa clase base, se tienen especializaciones para tener un nodo con texto, un nodo con una simple imagen o incluso un nodo que contiene una entidad propiamente dicha.

Formado el árbol, se recorre en *inorden* para ser dibujado.

4.5.4.5 Entidades y propiedades

En el motor se ha seguido el paradigma entidad-componente, la llamada centradas en propiedades en el apartado de objetos de juego de la teoría. Con ello, cada elemento del juego es una entidad independiente, ya sea jugador, enemigo, bala, objeto... y se representan como una entidad única, sin estructuras ni herencia.

Cada una de esas entidades está compuesta de una serie de componentes que definen sus características y comportamiento y que se pueden poner y quitar sin que afecten al resto de componentes al ser totalmente independientes entre ellos. (49)

Los componentes pueden ser Velocidad, Posición, Movimiento, Colisión... En el código se tiene la clase Entity como entidad, que hace uso de IdEntity para identificar cada una de las entidades, e implementa la clase PropertyManager la cual maneja todas las propiedades haciendo uso de las interfaces IProperty y Tproperty.

A partir de ahí cualquier clase o tipo de dato se puede usar como propiedad sin ningún tipo de herencia o implementación. Dichas propiedades son utilizadas y manejadas por los sistemas del motor.

Igualmente, se ha reutilizado las clases de PropertyManager y sus clases para el paso de parámetros arbitrarios de unas funciones a otras, como puede ser desde el parseado de datos xml a las clases que crean las entidades a partir de dichos datos.

4.5.4.6 Sistemas del juego

Teniendo las entidades y sus propiedades, el motor dispone de varios sistemas que se encargan de distintas tareas. Cada sistema maneja las propiedades que le corresponden, actualizando (en dos partes para un control más granulado) las entidades en el bucle del juego y permitiendo acciones específicas. Todos los sistemas están bajo una clase base ISystem. En la clase TypeSystem nos encontramos una enumeración de los distintos sistemas que tiene el motor.

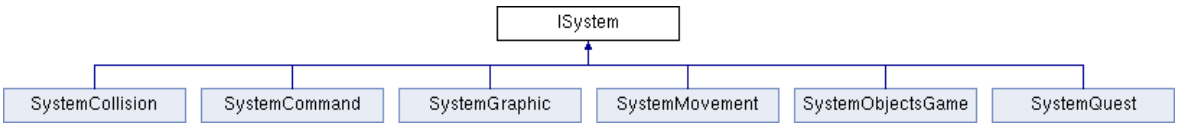


Figura 4-7. Jerarquía de sistemas

Lo importante de este apartado es la clase SystemManager, que controla y maneja el resto de sistemas. En este gestor es donde se realiza el registro y eliminación de las entidades, cuya vida es manejado por el sistema de objetos SystemObjectsGame. El registro y eliminación de entidades del resto de sistemas son métodos en los que delega el gestor de sistemas y no se llaman en ningún otro contexto.

Cuando una nueva entidad se crea, se registra en el sistema de entidades, el cual, si no lo encuentra en su sistema, lo añade, notifica inmediatamente al gestor de sistemas y éste se encarga de llamar a todos los sistemas (incluido el sistema de objetos) para que registren la nueva entidad si entra dentro de su rango de acción.

Cada sistema, cuando recibe la entidad, comprueba por medio de sus propiedades, si es una entidad que han de controlar ellos.

Igualmente, cuando el sistema de objetos determina que hay entidades cuya vida ha terminado, lo notifica al gestor de sistemas, el cual hace un llamado a todos los sistemas incluyendo al sistema de objetos.

Muchos de esos sistemas ofrecen métodos particulares relacionados con su campo, como la comprobación de colisiones del sistema de colisión, la formación de la escena en el sistema gráfico, o la petición de entidades concretas al sistema de objetos.

4.5.4.7 Colisiones

Para las colisiones, se tiene una clase Collision, que define una figura. A partir de esa figura, ofrece métodos para manejar los vértices de la figura, modificar su posición, rotación, redimensionado.

La parte más importante son los métodos que generan un rectángulo que contiene dichos vértices, de forma que se pueda testear de forma rápida y sencilla colisiones.

El sistema de colisiones mantiene un árbol, cuyos nodos son formados por la clase QuadTree y que ya se comentó anteriormente su funcionamiento básico. Dichos nodos ofrecen también métodos extras para consultar qué entidades colisionan o están en determinada zona, o con determinada entidad.

Dichos métodos son auxiliares a los principales métodos del sistema de colisiones, que son checkCollisions y resolveCollisions. El primer método realiza el checkeo en una determinada zona y guarda en una pila las colisiones, que luego son delegadas en las entidades en el método resolveCollisions.

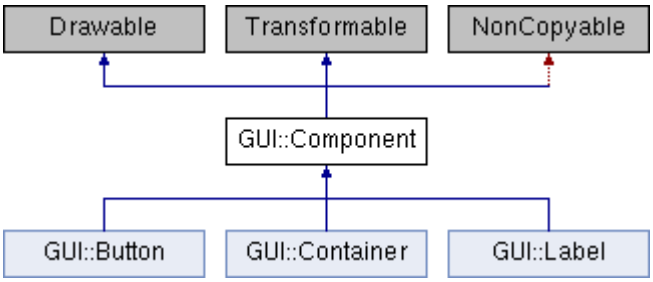
CheckCollisions obtiene en primer lugar las entidades de la región y comprueba entidades que puedan colisionar. Si los hay, comprueba con el método collisionDetection, que hace uso del test de rectángulos.

De haberlo, se continúa con el método firstTimeCollision, que intenta determinar el momento en el que se realizó la colisión, para pasar al método de triangulate, donde se triangula primero una figura y luego la otra, para afinar que efectivamente está colisionando dos triángulos y obtener el MTV, vector mínimo de traslado.

El cálculo de colisión entre triángulos es largo, por lo que se sacó a otro elemento y se tiene una secuencia de métodos para comprobar las colisiones. Si después de todo esto, se determina que hay colisión, se pone en la lista de colisiones.

4.5.4.8 GUI

Para los menús y similares, se hace uso de interfaz de usuario gráfica. Para hacer una distinción clara, se incluye el código bajo el espacio de nombres GUI, para evitar que nombres comunes como Component choquen con otros códigos. Toda clase de GUI hereda de la clase Component, clase base, que a su vez hereda de SFML para poder ser colocado y dibujado. (30)




```
namespace GUI {
class Component : public sf::Drawable, public sf::Transformable, private sf::NonCopyable {
public:
        Component();
        virtual ~Component();
        virtual bool isSelectable() const = 0;
        bool isSelected() const;
        virtual void select();
        virtual void deselect();
        virtual bool isActive() const;
        virtual void activate();
        virtual void deactivate();
        virtual void handleEvent(const sf::Event& event) = 0;
        private:
        bool selected;
        bool actived;
};
}
```

La clase Container actúa como la clase raíz en la jerarquía de componentes, introduciendo en él el resto de componentes GUI, incluyendo de nuevo otro container si se desea. El contenedor se encarga de la selección de los componentes seleccionables, selección que se realiza mediante el teclado.

El método de handleEvent realiza la acción sobre el componente seleccionado y es dependiente de cada componente que implemente la clase Component.

4.5.5 Tips programación

Durante la codificación y en bibliografías consultadas se han descrito una serie de tips y consejos, algunos de sentido común y común a cualquier ámbito de la programación, que pueden ser muy útiles para tener en cuenta mientras se programa y/o diseña el motor (9):

- Las funciones virtuales son poderosas, pero también peligrosas con fines malvados. Hay que tener en cuenta que una función virtual tiene un coste extra frente a los métodos normales.

También es una característica que suele ser abusada, creando funciones virtuales cuando no son necesarios o encadenando muchas funciones virtuales.

- Las funciones *get* triviales deben empezar con *get*, las no triviales con *find*, de esa forma permite alertar sobre la eficiencia del método.

- Los *string* siempre aparte, para la localización. Si es en caso de depuración, está bien *hardcodearlos*.

- Establecer unos prefijos determinados puede ayudar a identificar el tipo de variable o método que se tiene, o heredar sin miedo de varias clases al asegurarse de ser interfaces puras y duras. Un ejemplo de prefijos puede ser.
    - g para variables globales (g_Counter)
    - m para variables miembo (m_Counter)
    - p para punteros (m_pActor)
    - V para funciones virtuales (Vdraw())
    - I para interfaces (class Idrawable)

- La capitalización también contribuye a la claridad:
    - Variables y parámetros empezando en minúscula y las siguientes palabras compuestas con mayúscula inicial
    - Clases, funciones, typedefs y métodos empezando con mayúscula
    - Macros y constantes con mayúscula siempre
- Evitar esconder código y operaciones no triviales: Constructores copia, sobrecarga de operadores y destructores entra dentro de esos problemas. El mejor ejemplo es el destructor al no llamarlo nunca de forma explícita. El consejo es tratar de evitar toda operación no trivial en esos métodos, por ejemplo, si encuentras un simple igual o multiplicación, no vas a esperar que invoque una complicada fórmula como las series de Taylor.

De la misma forma, los constructores copia y operadores de copia entre otros, son creados por el compilador si no lo escribes de forma explícita, esos métodos se crean de forma pública y pueden dar efectos inesperados que sería mejor resolver creando dichos métodos aunque sea poniéndolos privados. Un buen ejemplo que se aplica es un objeto como un sonido de ambiente que puede ser de varios megas es obvio que se quiere evitar duplicaciones ocultas y se puede evitar poniendo dichos métodos privados.

El principal motivo es evitar siempre sorpresas, porque suelen ser sorpresas malas.

- Jerarquía de clases: mantenerlo lo más plano posible. Es un error común que se hace al sobrediseñar o al contrario sus clases y su jerarquía. Es algo que requiere práctica.

  Una buena regla a seguir es que cada clase tenga una única responsabilidad y sus árboles de herencia no deberían tener más de dos o tres niveles de profundidad. Siempre hay excepciones pero mejor evitarlo.

  Por el otro extremo, un problema común es la clase Blob bien descrito por los anti patrones (21), una clase que contiene un poco de todo y suele venir

dado por la vagancia del programador de hacer nuevas clases más pequeñas y separadas.

- Herencia vs composición: la herencia es usado para hacer "evolucionar" un objeto desde otro o cuando el hijo es una versión del padre. La composición es usado cuando un objeto se compone de varios componentes distintos.

- Usar interfaces: Clases con sólo funciones virtuales puros. Cada vez que sea posible, tenemos que hacer que el sistema dependa de las interfaces y no de la implementación de las clases. Dos sistemas diferentes no deberían nunca conocer las implementaciones de la otra, sólo interactuar con los métodos de la interfaz.

- Usar factorías: Los juegos tienden a construir complejos objetos. Por ello es aconsejable usar el patrón de diseño factoría. Un *abstract Factory* es un patrón de creación que define la interfaz para crear objetos. Diferentes implementaciones del *abstract factory* lleva a cabo las tareas concretas de construir objetos con varias partes y ensamblarlos.

  Es un buen patrón para escenas, animaciones, IA y otros. También ayuda a construirlos en el momento correcto, ya que se puede retrasar una instanciación. Se pueden crear objetos factoría, ponerlos en una pila y luego llamar al método construir cuando se esté listo. Algo útil cuando no se tiene memoria para instanciar hasta que no se destruya lo que ya hay de antes.

- Encapsular componentes que cambian: Las partes que más cambian del código es mejor encapsularla para no afectar al resto de los componentes cada vez que se realiza un cambio, como puede ser encapsular la parte de renderizado del juego, de forma que no afecte cuando se cambie de DirectX a OpenGL u otro sistema de renderizado.

- Usar *stream* para inicializar objetos: cualquier objeto persistente del juego debería implementar un método que admitiera un *stream* y leyera de él para inicializar el objeto.

4.6 Pruebas y resultados

Las pruebas realizadas han sido poco automatizadas y serias, consistiendo en simples ejecuciones del programa y probando a mano distintas funciones. El motivo de no realizar pruebas unitarias y similares se debe al desconocimiento de cuál sería la forma más adecuada o correcta de realizar los test en el lenguaje C++ y orientado a probar un motor de videojuegos.

A pesar de ello, los resultados no han sido malos. La aplicación, hasta donde se ha implementado, parece funcionar correctamente y se ha conseguido cumplir todos los requisitos a excepción de uno, el permitir guardar y cargar una partida previamente iniciada.

5 Conclusiones

En este trabajo se ha hecho un estudio sobre los motores de videojuegos. Inicialmente se realizó una toma de contacto con el lenguaje C++ y las librerías SDL y SFML con un clon del juego Tetris. Mientras, se realizaba una lectura y estudio de bibliografía recomendada para analizar la arquitectura de los motores, los diferentes enfoques con sus ventajas y desventajas y algunos motores de ejemplo que implementan dichos enfoques y estrategias. Posteriormente, con las bases establecidas y escogido la librería SFML para la realización del proyecto, se programó un motor sencillo aplicando algunas de las estrategias analizadas durante el estudio previo de los motores.

Tras todo este trabajo y los resultados obtenidos, hay mucho por donde mejorar, mucho por hacer y aprender. Pero se ha comprendido cómo es un motor de videojuegos, objetivo principal del proyecto. La cantidad de conocimientos que se han podido adquirir supera los plasmados en la memoria y forman una sólida base sobre el que continuar aprendiendo en el sector de videojuegos.

Aun siendo algo que ya se sabía antes de acometer el proyecto, la realización de un motor de videojuego es una tarea muy exigente en tiempos y conocimientos, nada adecuado para una sola persona y en plazos de tiempo relativamente cortos, hoy en día los motores actuales tienen grandes equipos de expertos detrás y pueden llegar tranquilamente a años de desarrollo.

Igualmente, la experiencia adquirida y la autorrealización que se llega a alcanzar, puede ser muy beneficiosa si se realiza como un reto personal a realizar durante un largo periodo de tiempo.

6 Líneas futuras

Entre las líneas futuras del presente trabajo podemos destacar las siguientes:

- Realizar pruebas automatizadas.
- Mejorar el procesamiento de datos externos.
- Añadir más elementos de juego para ofrecer más funciones al juego.
- Conseguir menor dependencia dentro del código.
- Controlar más la memoria, tanto la asignación como la liberación, y evitar fugas.

Por la naturaleza del proyecto, además de estas líneas, se pueden encontrar muchas otras líneas futuras que se pueden realizar.

Lista de referencias

1. Gregory, Jason. *Game Engine Architecture*. s.l. : A K Peters, Ltd., 2009. 987-1-4398-6526-2.

2. Wikipedia - Motor de videojuego. [En línea] http://es.wikipedia.org/wiki/Motor_de_videojuego.

3. Ward, Jeff. Game Career Guide - What is a game. [En línea] 29 de 04 de 2008. http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=2.

4. Wikipedia - First person shooter. [En línea] http://en.wikipedia.org/wiki/First-person_shooter.

5. Wikipedia - Platform game. [En línea] http://en.wikipedia.org/wiki/Platform_game.

6. Wikipedia - Fighting game. [En línea] http://en.wikipedia.org/wiki/Fighting_game.

7. Wikipedia - Real time strategy. [En línea] http://en.wikipedia.org/wiki/Real-time_strategy.

8. Wikipedia - Massively multiplayer online game. [En línea] http://en.wikipedia.org/wiki/Massively_multiplayer_online_game.

9. McShaffry, Mike y Graham, David. *Game Coding Complete*. s.l. : Cengage Learning, 2013. 987-1-133-77657-4.

10. Wikipedia - Cyclic redundancy check. [En línea] http://en.wikipedia.org/wiki/Cyclic_redundancy_check.

11. Rad Game Tools. [En línea] http://www.radgametools.com.

12. Wikipedia - Screen tearing. [En línea] https://en.wikipedia.org/wiki/Screen_tearing.

13. Open Dynamics Engine. [En línea] http://www.ode.org.

14. Genbeta Dev - Teoría de colisiones 2D, QuadTree. [En línea] http://www.genbetadev.com/programacion-de-videojuegos/teoria-de-colisiones-2d-quadtree.

15. Genbeta Dev - Teoría de colisiones 2D, conceptos básicos. [En línea] http://www.genbetadev.com/programacion-de-videojuegos/teoria-de-colisiones-2d-conceptos-basicos.

16. Wikipedia - Politopo. [En línea] http://es.wikipedia.org/wiki/Politopo.

17. Wikipedia - Separating axis theorem. [En línea] http://en.wikipedia.org/wiki/Separating_axis_theorem.

18. Matemáticas y videojuegos - volúmenes envolventes. [En línea] http://matesyvideojuegos.blogspot.com.es/2012/06/capitulo-4-volumenes-envolventes.html.

19. Wikipedia - Sweep and prune. [En línea] http://en.wikipedia.org/wiki/Sweep_and_prune.

20. SourceMaking - The blob. [En línea] http://sourcemaking.com/antipatterns/the-blob.

21. Cplusplus. [En línea] http://www.cplusplus.com/doc/tutorial/.

22. Stack Overflow. [En línea] http://stackoverflow.com/.

23. García Alba, Antonio. *Tutorial de libSDL para la programación de videojuegos.* 2008.

24. Open Libra. [En línea] http://www.etnassoft.com/biblioteca/.

25. Deitel. *Como programar en C/C++ y Java.* s.l. : Pearson, Prentice Hall.

26. Coursera - C++ para programadores en C. [En línea] https://www.coursera.org/course/cplusplus4c.

27. P. Cohoon, James y W.Davidson, Jack. *Programación y diseño en C++ Introducción a la programación y al diseño orientado a objetos.*

28. Stroustrup, Bjarne. *The C++ programming language*.

29. Simple DirectMedia Layer - SDL. [En línea] https://www.libsdl.org/ .

30. Gomila, Laurent. SFML. [En línea] http://www.sfml-dev.org/.

31. —. *SFML Game Development*. s.l. : Packt Publishing.

32. Netbeans. [En línea] https://netbeans.org.

33. Gitlab. [En línea] https://about.gitlab.com/.

34. GCC, GNU Compiler. [En línea] https://gcc.gnu.org/.

35. Doxygen. [En línea] http://www.stack.nl/~dimitri/doxygen/index.html .

36. Wikipedia - Tile. [En línea] https://es.wikipedia.org/wiki/Tile.

37. Game Development - Introduction to tiled map editor. [En línea] http://gamedevelopment.tutsplus.com/tutorials/introduction-to-tiled-map-editor--gamedev-2838.

38. Tiled. [En línea] http://www.mapeditor.org/.

39. TinyXML 2. [En línea] http://www.grinninglizard.com/tinyxml2/ .

40. Open Game Art. [En línea] http://opengameart.org/.

41. Game Development - Parsing tiled tmx. [En línea] http://gamedevelopment.tutsplus.com/tutorials/parsing-tiled-tmx-format-maps-in-your-own-game-engine--gamedev-3104 .

42. Genbeta Dev - Tiled map editor. [En línea] http://www.genbetadev.com/programacion-de-videojuegos/tiled-map-editor-el-editor-de-mapas-libre.

43. Wiki SFML. [En línea] https://github.com/SFML/SFML/wiki.

44. Genbeta Dev. [En línea] http://www.genbetadev.com/.

45. Game Dev Stack Exchange. [En línea] https://gamedev.stackexchange.com/.

46. Source Forge - box2dsandbox. [En línea] http://sourceforge.net/projects/box2dsandbox/.

47. Github - Project Mayhem. [En línea] https://github.com/simplerr/Project-Mayhem/tree/178acb0ff06d60c99b1891e4e40644ce8a056764.

48. SFML - Tutoriales. [En línea] http://www.sfml-dev.org/tutorials/2.0/.

49. Wikipedia - Patrón observer. [En línea] https://es.wikipedia.org/wiki/Observer_(patr%C3%B3n_de_dise%C3%B1o).

50. Genbeta Dev - orientado a entidades y componentes. [En línea] http://www.genbetadev.com/programacion-de-videojuegos/diseno-de-videojuegos-orientado-a-entidades-y-componentes.

Anexo A: Manual de usuario

Una vez se arranca el programa, la primera pantalla es la pantalla de bienvenida, en el que como bien indica, hay que pulsar cualquier tecla para continuar.

Figura A-1. Pantalla inicial

Al pulsar la tecla, nos encontramos con el menú inicial, el cual podremos navegar con las teclas de dirección y seleccionar con el botón Enter.

Desde aquí podemos:

- Salir del programa
- Ir a opciones
- Jugar una nueva partida / cargar una nueva partida: estas dos opciones realizan la misma acción, la de iniciar una nueva partida porque no se ha tenido tiempo de implementar el guardado y cargado de una partida.

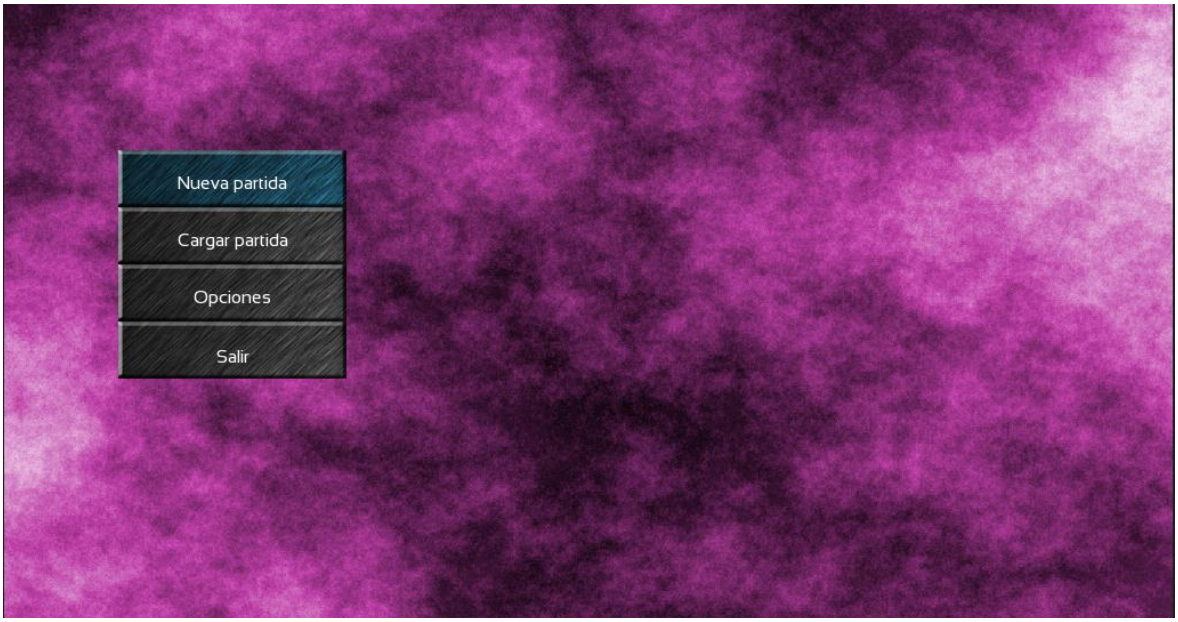


Figura A-2. Menú principal

Si accedemos a las opciones, vemos una lista de botones con las acciones disponibles durante el juego y la tecla al que está asociado.

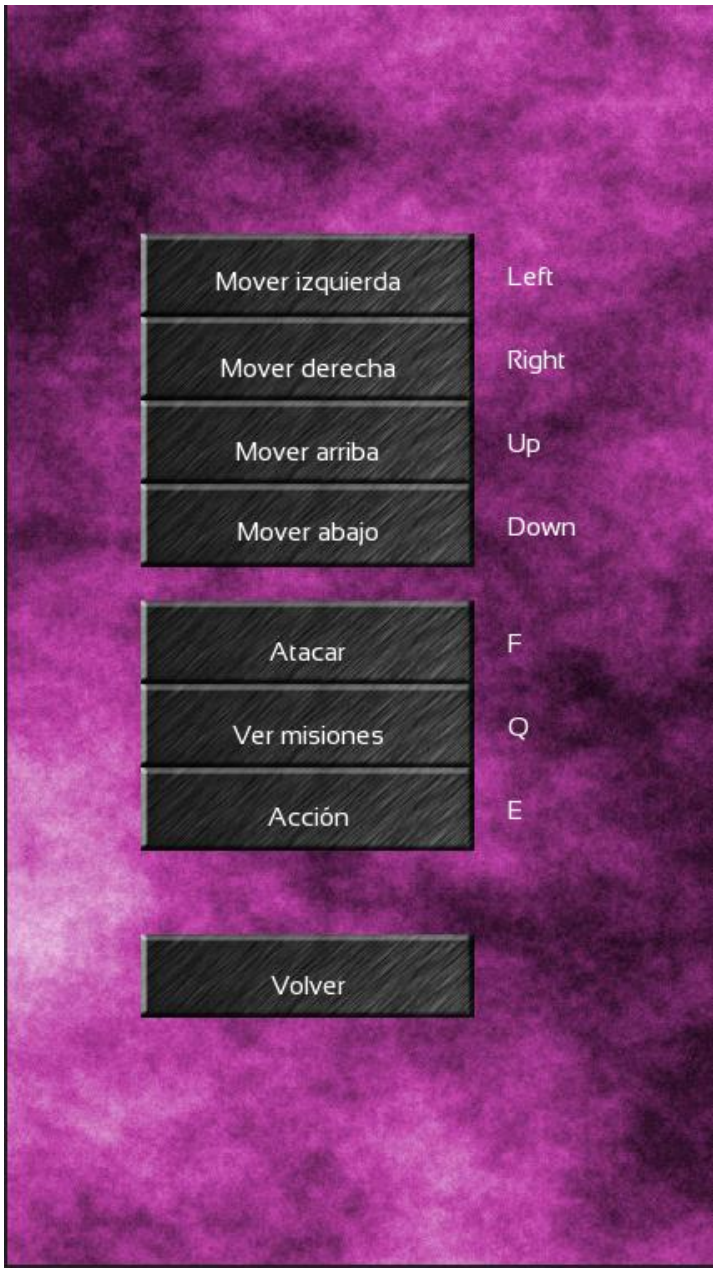


Figura A-3. Opciones

Al igual que en el menú principal, navegaremos con las teclas de dirección y seleccionaremos con la tecla Enter. Una vez seleccionado una acción, el botón queda marcado hasta que pulsemos una nueva tecla que se asociará a la acción seleccionada.

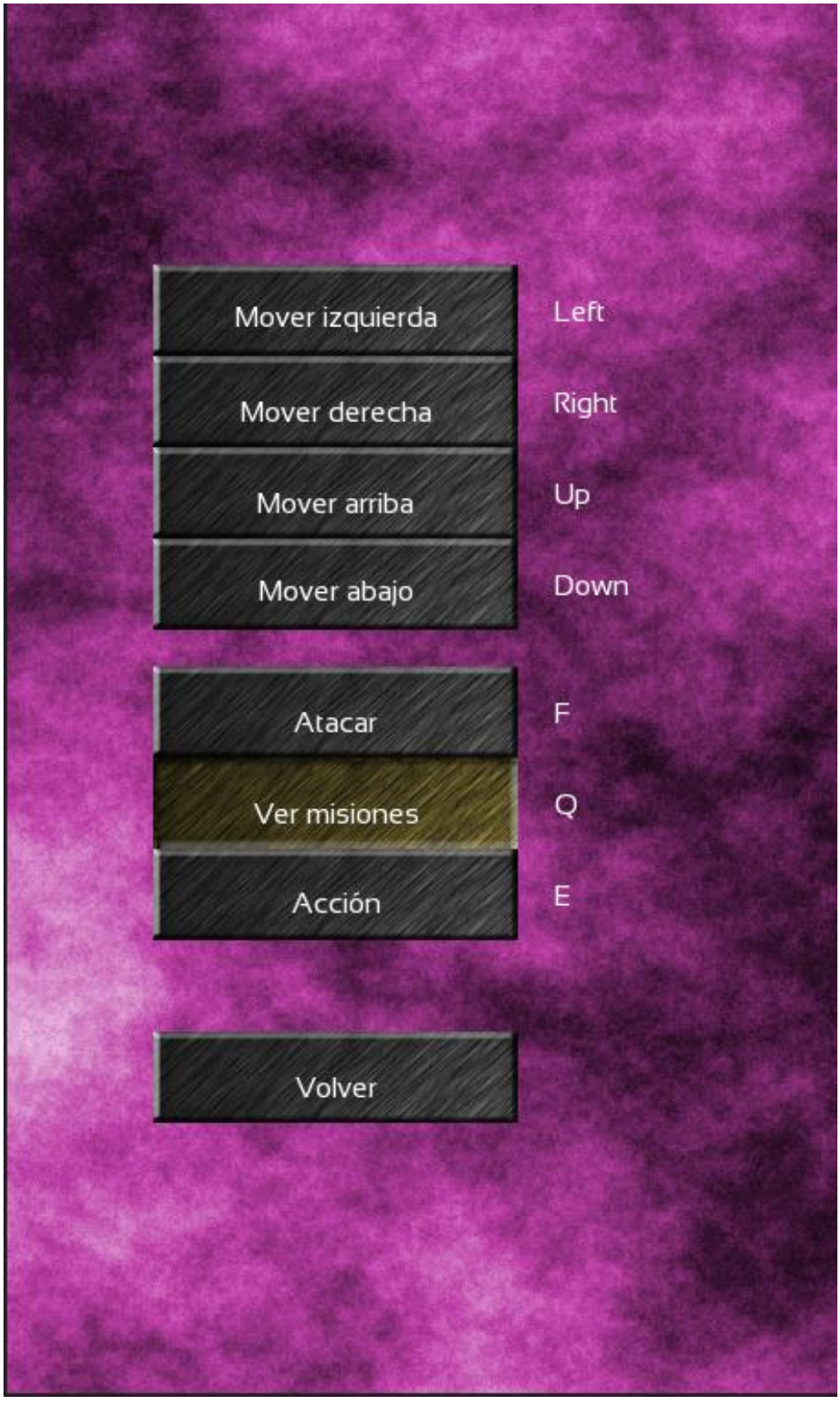


Figura A-4. Cambiando acción

Seleccionando el botón volver, se guardarán las opciones y se volverá al menú principal.

Al comenzar una nueva partida, a veces se nos pondrá una pantalla de cargando y se quitará al poco tiempo, según el tiempo que la aplicación tarde en cargar los datos. Nos encontramos en el juego en una pantalla similar a la siguiente.

Figura A-5. Pantalla de juego

En ella nos podremos mover e interactuar según los botones establecidos en la pantalla de opciones. Si quisieramos consultarlos o cambiarlos, con la tecla Escape se pausa el juego y se ofrece un menú.

Figura A-6. Menú de pausa

En dicho menú se puede volver al menú principal, perdiendo todo avance, continuar el juego, o acceder de nuevo a la pantalla de opciones.

Volviendo al juego, se puede hablar con los personajes con la tecla de acción. Si el personaje habla, se muestra un recuadro con un diálogo.

Figura A-7. Conversación larga con un aldeano

El círculo es indicador de que hay más texto para leer y se puede avanzar con cualquier tecla. Si la conversación ha finalizado, o no se quiere leer todo y salir de la conversación, pulsando la tecla Enter se da por finalizado.

Figura A-8. Conversación con aldeano

La barra encima de los personajes determina su vida y por tanto si se pueden dañar y matar atacándolos.

Figura A-9. Atacando al soldado

Figura A-10. Matando al soldado

Eso implica que el personaje principal puede morir, y entonces aparece la pantalla de juego finalizado.

Figura A-11. Ese agujero no es bueno...

Por último, el objetivo del juego es cumplir con unas misiones, podemos acceder a las misiones desde la tecla designada en las opciones. Para salir de la vista de misiones, se utiliza la tecla Enter.

Figura A-12. Visualizando las misiones

Anexo B: Manual de programador

B.1 Introducción al manual del programador

En este documento se da un repaso general a toda la estructura que compone la aplicación de forma que el usuario se familiarice con ella y sobre todo que sea capaz de orientarse a la hora de realizar modificaciones o ampliaciones.

En el documento se describirá brevemente cómo está organizado físicamente los ficheros y carpetas de la aplicación, se definirá la estructura lógica seguida en el código fuente, comentando cada módulo y cada fichero de código fuente de forma genérica intentando seguir un orden de menor a mayor dificultad y/o complejidad. Para cada uno de ellos se especifica los métodos que disponen y su signatura.

También se ofrece un apartado que muestra gráficamente la clasificación y estructuración de las clases para ofrecer una visión general. Finalmente, se dedica un apartado para clases y códigos más complejos que puedan requerir una explicación de apoyo adicional a los comentarios del propio código.

B.2 Estructura general de desarrollo

La estructura general del proyecto está muy influenciada por la estructura de proyecto del entorno de desarrollo de Netbeans. Por lo que tenemos las siguientes carpetas:

- build: Carpeta donde se realiza el proceso de compilado, es decir, donde se dejan los objetos fuente para la fase de enlazado.
- dist: Carpeta donde se deja el objeto ejecutable del proyecto
- doc: En él se encuentra la documentación del código en formato html y latex.
- etc: Carpeta donde se dejan recursos extras para el proyecto, como ficheros utilizados para la realización de pruebas automáticas.
- nbproject: Carpeta propia del entorno de desarrollo Netbeans
- include: Cabeceras y códigos fuentes incluidos en el proyecto
- Media: Todo recurso de la aplicación, como pueden ser imágenes, audio, ficheros varios...
- src: carpeta contenedora del código fuente.
- test: carpeta con el código fuente que realiza test

B.3 Estructura del código fuente

Físicamente el código fuente se encuentra en la misma carpeta, la carpeta src, sin organización alguna por no tener dominio del make (herramienta de gestión de dependencias) que permitiría establecer una jerarquía u organización al automatizar la compilación/generación.

Sin embargo, el código está organizado en una forma lógica, componiendo los siguientes módulos:

- Animations: Código fuente para manejar los recursos de animación
- Application: Código de inicio de la aplicación y de uso general en toda la aplicación
- Collision: Códigos referentes a las colisiones y sus cálculos
- Command: Códigos para el manejo de eventos/comandos entre entidades de la aplicación
- ConcreteStates: Estados concretos, son implementaciones del código que se encuentra bajo la carpeta de States
- Data: Para el manejo y uso de datos en bruto externos a la aplicación. Se subdivide en dos carpetas lógicas:
- DataStructure: estructuras de datos auxiliares al procesamiento de datos
- XMLParser: parseadores de datos xml
- Debug: Estructuras auxiliares a los propósitos de debug de la aplicación
- Entities: Estructuras bases para la implementación de personajes y otros objetos o entidades del juego. Contiene una carpeta:
- Properties: Estructura base para las propiedades de los objetos o entidades.
- GameObjects: Implementaciones concretas de las entidades del juego. Incluye también la fábrica para instanciar dichas implementaciones.
- GameStates: Código para los subestados de la aplicación durante el estado de juego.
- GUI: Códigos para la interfaz gráfica como botones o etiquetas.
- Music: Códigos encargados de la parte musical del motor.
- Properties: Implementaciones concretas de las propiedades que puede tener un objeto o entidad en el juego.
- Quest: Código referente al manejo de las misiones y objetivos del juego.

- Resources: Código para el manejo de recursos externos en la aplicación como las imágenes y fuentes.
- States: Estructuras base para los estados en los que se divide el juego/aplicación.
- Systems: Código para los sistemas del juego que llevan el control de una parte específica del juego.
- Utilities: Código de utilidades varias en la aplicación que no tiene cabida en una clase o dos ni en una categoría concreta.
- World: Código referente a los niveles y el manejo del terreno o mapa del juego. Se subdivide en dos carpetas:
- Level: Código para las cargas de nivel y su manejo y control.
- ScenneGraph: Código para la gestión y dibujado del nivel y/o mapa, implementa los nodos del árbol que maneja el sistema gráfico para el dibujado.

La detallación de los módulos se intentará hacer en un orden en el que se partirán de los módulos más independientes y/o sencillos a los complejos o con dependencias a otros módulos. Se indicará, si procede, algunas de las funciones más importantes de los ficheros.

B.3 Detalles de los módulos de código fuente

B.3.1 Módulo Utilities

CardinalPoints.h: Enumeración para indicar los cuatro puntos cardinales.

HashIdEntity.h: Implementan el hash de la clase IdEntity para poder usar dicha clase como clave en los hashmaps.

ParallelTask.h ParallelTask.cpp: Esta clase abstracta implementa el código necesario para realizar implementaciones que se pueden realizar la tarea en paralelo. Su uso principal en la aplicación es la carga de niveles en paralelo al juego. La implementación concreta realizada está en la clase LoadingLevel.

Para su implementación se ha de sobreescribir el método runTask(), donde va el código que se quiere realizar en paralelo.

void ParallelTask::runTask ( ) pure virtual

Método a sobreescribir. En él se codifica la tarea a realizar.

ConvexTest.h: Este fichero contiene los métodos auxiliares necesarios para comprobar si una estructura de colisión forma una figura convexa.

bool polygonIsConvex (StructCollision* collision)

Comprueba si el polígono es convexo

Parámetros

collision la estructura de colisión a comprobar

Devuelve

true si el polígono es convexo

ConvexHull.h: Implementación del algoritmo de la envolvente convexa, consistente en obtener una figura convexa que englobe todos los puntos dados. Inicialmente se desarrolló para la comprobación de colisiones, pero actualmente está sin uso alguno en la aplicación.

std::vector<Point> convertConvexHull(std::vector<Point> P)

Devuelve una lista de puntos de la convexa envolvente en el orden de las agujas del reloj.

Nota: El último punto del resultado es el mismo que el primero

Parámetros

P vértices de la figura

Devuelve

std::vector<Point> lista de puntos que engloba la figura

Utilities.h Utilities.cpp: Fichero que aúna una serie de métodos útiles en varios sitios o sin asociación a una clase. Algunos de los métodos están sin uso.

B.3.2 Módulo GUI

Component.h Component.cpp: Clase base para los distintos componentes de la interfaz gráfica.

Container.h Container.cpp: Clase que actúa de contenedor para los componentes gráficos.

Label.h Label.cpp: Clase que representa una etiqueta o label en la interfaz gráfica.

Button.h Button.cpp: Clase que representa un botón en la interfaz. Hace uso de la enumeración ButtonState para controlar el estado del botón en cada momento.

ButtonState.h: Enumeración para el estado del botón.

B.3.3 Módulo Application

main.cpp: Punto de partida de la applicación. Desde esta fichero se inicializan los hilos para su uso en ParallelTask y se instancia y corre la clase Application.

Context.h Context.cpp: Alternativa al patrón Singleton. En esta clase se instancia una vez en la aplicación y se mantienen los objetos que igualmente se instancian una vez en la aplicación y son de uso general en distintas partes. Mantiene punteros a dichos objetos.

Player.h Player.cpp: Su nombre aquí no es muy descriptivo, es la clase donde se realiza el mapeo entre la entrada que recibe la aplicación y los comandos o acciones que se han de tomar durante el juego.

Los métodos más importantes son los que permiten setear las teclas a alguna acción, y procesar la entrada.

void Player::asignKey (Actions action,
sf::Keyboard::Key key)

Asigna una tecla a una determinada acción

Parámetros

action acción a setear
key código de tecla asociada a la acción

void Player::handleEvent (const sf::Event& event,
CommandQueue& commands)

Procesa la entrada, creando si es necesario un comando a ejecutar que deja en la pila de comandos.

Parámetros

event evento
commands pila de comandos

Application.h Application.cpp: Esta clase es el encargado de inicializar todos los sistemas, estados, contexto y la ventana de la aplicación y controla el bucle del juego y la entrada de datos por parte del usuario (teclado, joystick…) encargándose de los fps del juego y delegando los eventos que recibe al sistema adecuado.

B.3.4 Módulo States

State.h State.cpp: Clase abstracta, base para un estado de la aplicación. Es una clase abstracta en el que se tienen que sobrescribir la mayoría de métodos, entre ellos, especial atención a los siguientes dos por su especial comportamiento.

void State::pulledAction ( ) virtual

Método ejecutado cuando es sacado de la pila.

void State::pushedAction ( ) virtual

Método ejecutado cuando es puesto en pila.

StateStack.h StateStack.cpp: La pila de estados. Quizás lo más importante no sea la pila en sí, de funcionamiento sencillo, sino la siguiente función, ya que si no se encuentran registrados los estados, la aplicación no funciona.

void StateStack::registerState (StatesID stateID)

Procesa la entrada, creando si es necesario un comando a ejecutar que deja en la pila de comandos.

Parámetros

stateID identificador del estado

PendingChange.h PendingChange.cpp: Cuando se añade o quita estados de la pila, no se hacen en el momento, por lo que se guarda el cambio a realizar en una lista de la pila de estados, y cuando se realiza una actualización en la pila de estados, se ejecutan todos los cambios que se detallan en esta clase (quitar o poner de la lista y qué estado se quita o pone).

ActionStack.h: Una enumeración de las posibles acciones que se pueden ejecutar sobre la pila de estados (apilar, sacar o limpiar).

StatesID.h: Enumeración de todos los estados de los que dispone la aplicación.

B.3.5 Módulo ConcreteStates

TitleState.h TitleState.cpp: Pantalla de título del juego

MenuState.h MenuState.cpp: Pantalla de menú inicial del juego, donde se pasan a las pantallas de opciones, juegos y otros.

LoadingState.h LoadingState.cpp: La pantalla de carga de nivel, el clásico cargando... mientras se ejecuta una tarea en paralelo que realiza la carga.

PauseState.h PauseState.cpp: Pantalla de pausa del juego. El estado captura los eventos, pero permite que se dibuje el estado anterior a él, mostrando el juego pausado por detrás del menú que muestra.

GameState.h GameState.cpp: Pantalla de juego.

GameOverState.h GameOverState.cpp: Pantalla de juego finalizado

SettingsState.h SettingsState.cpp: Pantalla de opciones del juego.

IncludeStates.h: Importaciones de todos los estados existentes.

B.3.6 Módulo Resources

IDFonts.h: Enumeración con las fuentes de la aplicación.

ResourceHolder.h: Es una clase template que se encarga de la gestión de un recurso con un identificador, es decir, carga los recursos a memoria y se accede a ellos a través de un identificador.

B.3.7 Módulo Music

MusicID.h: Enumeración de la música disponible.

MusicPlayer.h: MusicPlayer.cpp: Controla la reprodución de la música.

SoundEffectID.h: Enumeración de los efectos disponibles.

SoundPlayer.h: SoundPlayer.cpp: Controla la reprodución de los efectos especiales, incluso con efecto 3D.

B.3.8 Módulo Entities

Category.h: Distintas categorías en los que clasificar las entidades para refinar el envío de comandos a entidades.

Entity.h Entity.cpp: Una simple clase que representa una entidad en la aplicación/juego. Sólo contiene un identificador, una serie de propiedades heredadas de la clase PropertyManager y la categoría en la que se engloba.

IdEntity.h IdEntity.cpp: Un simple identificador. Mantiene una variable estática para asignar ids únicos en toda la aplicación.

B.3.8.1 Submódulo Properties

IProperty.h Iproperty.cpp: Define la interfaz de una propiedad de entidad. Al ser una template, tiene un nombre definido como string, y un valor que puede ser de cualquier tipo o clase.

PropertyManager.h PropertyManager.cpp: Gestor de propiedades, mantiene una lista de propiedades. Esta clase es la que se usa en sitios que requieran de un número variable de propiedades, como puede ser Entity.

TProperty: La versión en formato template de IProperty para su uso con clases propias, hereda de IProperty.

B.3.9 Módulo Systems

ISystem.h ISystem.cpp: Interfaz para los sistemas del juego. Tiene especial importancia los siguientes métodos, cuyas implementaciones definen el comportamiento del motor.

void ISystem::registerEntity (Entity* entity) pure virtual

Registra una entidad en el sistema si es de su incumbencia

Parámetros

entity entidad a registrar

void ISystem::update (sf::Time dt) virtual

Actualiza el sistema en una primera tanda

Parámetros

dt tiempo entre frame y frame

void ISystem::updateSecondPart (sf::Time dt) virtual

Actualiza el sistema en una segunda tanda

Parámetros

dt tiempo entre frame y frame

ImportSystem.h: Simple cabecera para poder importar todos los sistemas con un solo include.

TypeSystem.h: Enumeración con los distintos sistemas que hay en la aplicación.

SystemCollision.h SystemCollision.cpp: Sistema que se encarga de las colisiones. Se encarga de las entidades con la propiedad Collision y Position. Mantiene un quadtree con los objetos para una mayor eficiencia en sus procedimientos y queries y se encarga de los cálculos de colisión.

void SystemCollision::checkCollisions (sf::FloatRect region)

Chequea todas las collisiones que se producen en la región dada y las va guardando en una pila

Parámetros

region zona a consultar colisiones

void SystemCollision::newWorldCollision (sf::FloatRect region)

Crea un nuevo mundo colisionable

Parámetros

region nuevos límites del mapa

std::vector<Entity*>* SystemCollision::query (sf::FloatRect query)

Busca todas las entidades que colisionan con el cuadrado dado

Parámetros

query zona a consultar

void SystemCollision::resolveCollisions ( )

Resuelve las colisiones que han quedado en la cola después de comprobar. La resolución de colisiones son delegados en las entidades.

Para la detección de colisiones, se usan una serie de métodos relacionados entre sí.

MessageCollision* SystemCollision::collisionDetection (Entity* one ,
Entity* two)

Hace una prueba simple y genérica de si las dos entidades colisionan. Si es así, pasa a mayores comprobaciones

Parámetros

one una entidad
two otra entidad

Devuelve

Mensaje de colisión o nullptr si no hay colisión

MessageCollision* SystemCollision::firstTimeCollision (Entity* one ,
Entity* two)

Comprueba en que momento habrían colisionado las entidades y manda triangulizar para afinar la colisión

Parámetros

one una entidad
two otra entidad

Devuelve

Mensaje de colisión o nullptr si no hay colisión

MessageCollision* SystemCollision::triangulate (Collision* first ,
Entity* second ,
float time)

Comprueba en que momento habrían colisionado las entidades y manda triangular para afinar la colisión

Parámetros

first un colisionable
second otro colisionable
time tiempo en el que presumiblemente han colisionado

Devuelve

Mensaje de colisión o nullptr si no hay colisión

MTV SystemCollision::checkCollisions (Triangle& poly1 ,
Triangle& poly2,
float detailed)

Dado dos triángulos que en teoría colisionan, comprueba el colisionado y calcula el mínimo vector de traslado para separar ambos triángulos.

Parámetros

poly1 un triágulo
poly2 otro triángulo
detailed true si se desea tener dicho mtv o false si solo se quiere comprobar que colisionan sin hacer cálculos extras

Devuelve

El mínimo vector de traslado

SystemCommand.h SystemCommand.cpp: Sistema que se encarga de manejar los comandos o acciones. Trata con las entidades que tienen la propiedad Commandable.

void SystemCommand::onCommand (CommandQueue& queue,
sf::Time delta)

Ejecuta los comandos sobre las entidades

Parámetros

queue cola de comandos
delta tiempo entre frame y frame

SystemGraphics.h SystemGraphics.cpp: Encargado del dibujado en escena y actualizar las imagenes con el paso del tiempo. Mantiene un árbol con el que determina con rapidez que partes hay que dibujar.

void SystemGraphic::newScene (StructMap* infoMap)

Crea una nueva escena gráfica

Parámetros

infoMap información sobre el mapa que se pasa a la escena gráfica

SystemManager.h SystemManager.cpp: Sistema ligeramente distinto al resto. Mantiene una lista con el resto de sistemas e implementa sus mismas funciones, las cuales delega en los sistemas que gestiona.

SystemMovement.h SystemMovement.cpp: Sistema encargado del movimiento de las entidades, aquellos con la propiedad Velocity.

SystemObjectsGame.h SystemObjectsGame.cpp: Sistema que mantiene una referencia a todas las entidades de la aplicación

Entity* SystemObjects::getEntity (IdEntity id)

Devuelve la entidad solicitada

Parámetros

id identificador de la entidad solicitada

Devuelve

La entidad solicitada, si no lo encuentra, salta excepción

Entity* SystemObjects::getEntityXml (IdEntity id)

Devuelve la entidad solicitada

Parámetros

id identificador del xml de la entidad solicitada

Devuelve

La entidad solicitada, si no lo encuentra, salta excepción

SystemQuest.h SystemQuest.cpp: Sistema que maneja las quest o misiones del juego que se encuentran asociados a entidades (entidades per se, o entidades

creadas especificamente para contener la misión). Se encarga de las entidades con la propiedad Questeable.

MissionStatus SystemQuest::getStatus ( )

Devuelve el estado de las misiones, de forma que se sepa si ha ganado o perdido el juego, o está en marcha todavía.

Devuelve

Estado del juego

B.3.10 Módulo World

Se engloba aquí lo referente a los niveles y mundos. Se subdivide en dos módulos.

B.3.10.1 Submódulo Level

Level.h Level.cpp: Clase que representa un determinado nivel manteniendo referencia a los sistemas que controlan las distintas partes del juego y la entidad del jugador. Controla el orden de actualización, tratamiento de eventos y dibujado, delegando en los correspondientes sistemas y controla por sí mismo los subestados concretos del juego.

LoadingLevel.h LoadingLevel.cpp: Implementación concreta de ParallelTask que realiza la carga de un nivel en paralelo mientras muestra una pantalla de espera.

B.3.10.2 Submódulo SceneGraph

Layer.h: Enumeración con las distintas capas que se pueden distinguir en el dibujado del juego, las cuales se traducen en subraíces del árbol que forman los nodos del SceneGraph.

void Level::update ( )

Actualiza los subestados del juego cuando hay cambios. Implementa el patrón Observer

SceneNode.h SceneNode.cpp: Clase abstracta para los nodos del árbol. También actúa como un nodo contenedor vacío para otros tipos de nodos. Importa las siguientes funciones al definir las posiciones y orden donde ser colocados/dibujados, afectando al ordenamiento de los nodos en el árbol.

sf::Transform SceneNode::getWorldPosition ( ) const

Devuelve las transformadas absolutas (posición, rotación y escalado) del nodo

Devuelve

Las transformadas relativas

void SceneNode::setXOffset (float xOffset)

Setea el offset en el eje x por si la posición a tener en cuenta del elemento no coincide con la de la entidad.

Parámetros

xOffset offset en el eje x

void SceneNode::setYOffset (float yOffset)

Setea el offset en el eje y por si la posición a tener en cuenta del elemento no coincide con la de la entidad.

Parámetros

yOffset offset en el eje y

Debug.h Debug.cpp: Nodo utilizado para dibujar entidades que tienen la propiedad Debug, que les permite dibujar extras para facilitar la depuración.

EntityNode.h EntityNode.cpp: Nodo que dibuja una entidad concreta con la propiedad Drawable.

LifeNode.h LifeNode.cpp: Cuando una entidad tiene la propiedad Life (vida), entonces se le asocia este nodo, que a partir de la clase Life, controla la cantidad de vida y dibuja acorde a ello.

SpriteNode.h SpriteNode.cpp: Nodo que simplemente controla un sprite (imagen) a dibujar.

TextNode.h TextNode.cpp: Nodo para dibujar textos.

TileMapNode.h TileMapNode.cpp: Nodo que controla y maneja el fondo principal del mapa, mantiendo la lista de los tiles de los que se compone, controlando su escalado y las zonas que se ven.

void TileMapNode::prepareMap (const int* tilesMap)

Prepara el mapa después de inicializar el nodo con detalles como el número de columnas o filas, tamaño de ventana…

Parámetros

tilesMap los tiles del mapa

TileNode.h TileNode.cpp: Nodo que representa uno o varios tiles o recuadros del mapa. Se usa para añadir detalle al fondo general del mapa.

void TileNode::prepareMap (const std::vector<sf::Vector3i>& tileMap)

Prepara el mapa después de inicializar el nodo con detalles como el número de columnas o filas, tamaño de ventana…

Parámetros

tileMap vector con los tiles. Cada tile está representado en un vector de tres enteros indicando posición x,y y el tercer entero representa el tile a dibujar.

B.3.11 Módulo Properties

Behaviour.h Behaviour.cpp: Simple estructura contenedora de una función a ejecutar o usar cuando se precisa una reacción por parte de la entidad, un comportamiento. Las condiciones y momento de ejecución son usadas por el sistema que lo gestiona.

Life.h Life.cpp: Clase que maneja la vida de una entidad.

OnCollision.h OnCollision.cpp: Simple estructura contenedora de una función a ejecutar o usar cuando se produce una colisión. Las condiciones y momento de ejecución son usadas por el sistema que lo gestiona.

Position.h Position.cpp: Clase que guarda y controla la posición en la que se encuentra la entidad, al igual que el punto hacia el que está mirando. También guarda la posición anterior que se tenía y con ello sacar el vector de movimiento entre una posición y otra.

Questeable.h Questeable.cpp: Clase que guarda una referencia a una misión o parte de una misión asociado a la entidad.

Talk.h Talk.cpp: La clase guarda un simple string que representa la conversación que da una entidad en un momento determinado.

Velocity.h Velocity.cpp: Guarda y maneja la velocidad en relación a los ejes de coordenadas que tiene en un determinado momento una entidad.

B.3.12 Módulo GameState

GameStates.h: Enumeración de los posibles subestados del estado juego.

SubStateGame.h SubStateGame.cpp: Interfaz para los subestados del juego.

ConversationState.h ConversationState.cpp: Subestado del juego relativo a la conversación entre personajes.

QuestState.h QuestState.cpp: Subestado del juego relativo a las misiones.

Subject.h Subject.cpp: Patrón observer, clase que mantiene una lista de observadores y el elemento vigilado.

Observer.h Observer.cpp: El observador que mantiene el subject que está vigilando y del que obtendrá los datos cuando sea notificado de cambios.

void TileNode::update ( ) pure virtual

Método a ejecutar cuando haya cambios en el objeto vigilado

Message.h Message.cpp: El mensaje que se usa para ser vigilado en el Subject y dar información sobre cambios de subestados de juego.

B.3.13 Módulo Quest

TypeQuest.h: Enumeración con los distintos tipos de misiones que hay.

Quest.h Quest.cpp: Representa una misión, con las partes de las que se compone, si está disponible para ser realizado, si se ha finalizado o no y si tiene temporizador, al igual que si las partes de la misión se realizan en orden o no.

std::vector<int> Quest::onFinished ( )

Lista de misiones que se activarán cuando esta misión se termine

Devuelve

Lista de misiones a activar al finalizar

PartQuest.h PartQuest.cpp: Parte de una misión. Contiene los detalles como los id de destino y origen, tipo de acción a realizar y otros.

MissionStatus.h: Estado de misión.

B.3.14 Módulo GameObjects

GameObjects.h GameObjects.cpp: Clase abstracta. Representa la entidad base de un objeto de juego como pueden ser los personajes o los objetos. Es parte del patrón factory.

Entity* GameObjects::prepareEntity (PropertyManager& parameters) pure virtual

Devuelve una entidad que define un objeto en el juego

Parámetros

parameters conjunto de parámetros

Devuelve

Entidad formada

FactoryGameObjects.h FactoryGameObjects.cpp: Dado un tipo de objeto, devuelve la clase que crea dicho objeto. Sigue el patrón factory.

StaticBlock.h StaticBlock.cpp: Crea una entidad que representa un bloque estático no visible y en cuya colisión no permite pasar. Importante la función que define su comportamiento ante colisiones.

void StatickBlock::makeOnCollision (IdEntity idObject,
OnCollision* onCollision)

Define una función que se ejecutará ante las colisiones contra esta entidad

Parámetros

idObject id de la entidad recién creada

onCollision la propiedad de colisión de la entidad

Hole.h Hole.cpp: Crea una entidad que representa un "agujero" estático no visible y en cuya colisión acaba con la vida de la entidad que choca contra él. Importante la función que define su comportamiento ante colisiones.

void Hole::makeOnCollision (IdEntity idObject,
OnCollision* onCollision)

Define una función que se ejecutará ante las colisiones contra esta entidad

Parámetros

idObject id de la entidad recién creada
onCollision la propiedad de colisión de la entidad

Villager.h Villager.cpp: Crea una entidad representando un aldeano cualquiera, que puede tener charlas, vida y acciones entre otras cosas dependiendo de los parámetros pasados. Importante las funciones que define su comportamiento ante colisiones, acciones de otras entidades y procesar las acciones de forma que comprueba si cumple misiones.

void Villager::makeOnCollision (IdEntity idObject,
OnCollision* onCollision)

Define una función que se ejecutará ante las colisiones contra esta entidad

Parámetros

idObject id de la entidad recién creada
onCollision la propiedad de colisión de la entidad

void Villager::makeBehaviour (Behaviour* behaviour,
Entity* entity)

Define el comportamiento de esta entidad ante las acciones de otra entidad

Parámetros

behaviour estructura donde dejar la función que se ejecutará
entity la entidad recién creada

void Villager::processQuest (Questeable* quest,
Entity* entity,
Actions action)

Define el comportamiento de esta entidad ante las acciones de otra entidad

Parámetros

quest estructura donde dejar los datos
entity la entidad recién creada
action acción realizada

Character.h Character.cpp: Crea una entidad representando al personaje del jugador, con su vida, acciones y demás. Importante la función siguiente, donde reacciona a diversos eventos de comportamiento.

void Character::makeBehaviour (Behaviour* behaviour,
Entity* entity)

Define el comportamiento de esta entidad ante las acciones de otra entidad

Parámetros

behaviour estructura donde dejar la función que se ejecutará
entity la entidad recién creada

Weapon.h Weapon.cpp: Entidad representando un arma.

ChangeLevel.h ChangeLevel.cpp: Entidad que representa una zona en cuyo contacto provoca el cambio de nivel. Importante la función que define su comportamiento ante colisiones, cambiando el nivel.

void ChangeLevel::makeOnCollision (IdEntity idObject,
OnCollision* onCollision)

Define una función que se ejecutará ante las colisiones contra esta entidad

Parámetros

idObject id de la entidad recién creada
onCollision la propiedad de colisión de la entidad

B.3.15 Módulo Debug

FrameClock.h: Una clase de reloj para poder realizar medidas más precisas.

ClockHUD.h: Una clase para dibujar datos estadísticos en la pantalla y conocer detalles sobre los grames y otros.

B.3.16 Módulo Command

Actions.h: Enumeración con posibles acciones que puede hacer el jugador.

Command.h Command.cpp: Representa un comando que guarda una acción/función a realizar.

CommandQueue.h CommandQueue.cpp: Pila para guardar comandos.

B.3.17 Módulo Animations

StateMachine.h StateMachine.cpp: Contiene las transiciones que definen una máquina de estados finita.

void StateMachine::processEntry (int entry)

Procesa la entrada y devuelve el nuevo estado

Parámetros

entry estructura donde dejar los datos

Animation.h Animation.cpp: Contiene una secuencia de imágenes que forman una animación y guarda si es una animación que se repite y/o por la que esperar a que termine antes de reaccionar a otras acciones o animaciones.

AnimatedSprite.h AnimatedSprite.cpp: Contiene una animación y se encarga de pasar las imágenes para animar la secuencia que contiene la animación además de controlar la pausa o reinicio de la animación entre otras cosas.

StateMachineAnimation.h StateMachineAnimation.cpp: Controla todo el movimiento y cambio de animaciones de una entidad con ayuda de máquinas de estado finitas (la clase StateMachine).

void StateMachineAnimation::update (int action)

Actualiza la máquina de estados finitos con la entrada

Parámetros

entry entrada para la máquina

B.3.18 Módulo Collision

Collision.h Collision.cpp: Define un objeto colisionable por medio de una serie de vértices.

QuadTree.h QuadTree.cpp: Estructura de datos que mantiene organizado las entidades con la propiedad Collision, permite métodos de consulta. La estructura es manejada por el sistema de colisiones. La importancia de los métodos siguientes radica en que definen la división y ordenación de los objetos en el árbol, y como consecuencia, el rendimiento y efectividad de las consultas

bool QuadTree::inside (sf::FloatRect rect)

Comprueba si la zona encaja totalmente dentro del nodo (es decir, todos sus vértices están contenidos)

Parámetros

rect zona a testear

Devuelve

True si todos sus vértices están dentro del nodo

int QuadTree::getIndex (sf::FloatRect rect)

Devuelve el nodo hijo que contiene la zona dada

Parámetros

rect zona a testear

Devuelve

Índice del nodo hijo que contiene la región, o -1 si ningún hijo puede contenerlo

int QuadTree::getIndex (Collision* collision)

Devuelve el nodo hijo que contiene la zona dada

Parámetros

collision figura colisionable

Devuelve

Índice del nodo hijo que contiene la región, o -1 si ningún hijo puede contenerlo

Por último, los métodos principales y más usados, son los siguientes ya que determinan el conjunto de entidades que colisionan o pueden colisionar.

```
std::vector<Entity*>*  QuadTree::retrieve ( std::vector<Entity*>* list ,
                                            Entity* object)
```

Devuelve una lista de todas las entidades que colisionan con la entidad dada.

Parámetros

list lista donde guardar entidades colisionadas
object entidad al que testear colisiones

Devuelve

Lista con las entidades que colisionan

```
std::vector<Entity*>*  QuadTree::retrieve ( std::vector<Entity*>* list ,
                                            sf::FloatRect* rect)
```

Devuelve una lista de todas las entidades que están en la zona dada

Parámetros

list lista donde guardar entidades de la zona
rect zona a testear

Devuelve

Lista con las entidades que tocan la zona

TypeCollision.h: Enumeración con los tipos de colisiones que pueden suceder.

DelaunayTriangulation.h DelaunayTriangulation.cpp: Ficheros con métodos para realizar la triangulación de delaunay en los objetos de colisión para testear si hay colisión. El principal método es el siguiente

```
int DelaunayTriangulation::CreateDelaunayTriangulation (sf::VertexArray* verts,
                                                        int n_verts,
                                                        std::vector<sf::Vector2f>& extraPoints,
                                                        int pointCount,
                                                       std::vector<Triangle>& triangles,
                                                       sf::Transform& transform)
```

Triangula una figura. Devuelve el número de triángulos de la figura, cualquier vector inválido se ignora.

Parámetros

verts vértices de la figura
n_verts número de vértices de la figura
extraPoints vértices extra de la figura
pointCount conteo de los vértices extra
triangles vector de triángulos resultantes, se rellena en el método
transform transformada de los vértices

Devuelve

Número de triángulos

Triangle.h: Estructura que define un triángulo y algunos métodos para testear colisiones.

Vector.h Vector.cpp: Clase representando un vector de dos puntos y en el que se han sobreescrito operadores a conveniencia para su uso en colisiones.

MTV.h MTV.cpp: Minimum Translation Vector, el vector de traslado mínimo, usado para en colisiones saber cuánto ha de moverse una entidad para evitar la colisión realizada.

MessageCollision.h MessageCollision.cpp: Estructura para guardar el mensaje que pasamos cuando hay colisión, dicha colisión sea tratada por las entidades o sistemas que correspondan, dando los datos necesarios para ello.

B.3.19 Módulo Data

B.3.19.1 Submódulo DataStructure

DataUnion.h: Guarda en cada momento alguno de los tipos de datos que se pueden leer de los xml externos.

StructMap.h StructMap.cpp: Guarda datos leidos sobre la estructura de un mapa, como es el fondo entero y tiles sueltos que van por encima para ser tratados y convertirlos en sus clases finales para ser usados. Es una clase de buffer en la lectura de xmls.

StructCollision.h StructCollision.cpp: Guarda de forma temporal información sobre las colisiones que va leyendo en el xml.

StructAnimation.h StructAnimation.cpp: Guarda de forma temporal información sobre las animaciones que va leyendo en el xml.

QuestData.h QuestData.cpp: Guarda de forma temporal información sobre las misiones que va leyendo en el xml.

StructPeople.h StructPeople.cpp: Estructura para guardar la relación de personajes de un mapa con sus respectivos archivos de datos.

B.3.19.2 Submódulo XMLParser

IXMLParser.h IXMLParser.cpp: Interfaz para los parseadores de datos xml. El método más importante es el que hay que implementar para realizar el parseo.

void IXMLParser::parse (DataUnion& data)

Parsea los datos

Parámetros

data estructura donde dejar la información parseada

TypeParser.h: Enumeración con los distintos parseadores que hay.

XMLDocument.h XMLDocument.cpp: Clase que representa un documento XML haciendo uso de la librería tinyxml2 y el que se lee para parsear.

XMLParserAnimation.h XMLParserAnimation.cpp: Hereda de IXMLParser. Parseador xml para las animaciones donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserCollisions.h XMLParserCollisions.cpp: Hereda de IXMLParser. Parseador xml para las colisiones donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserMap.h XMLParserMap.cpp: Hereda de IXMLParser. Parseador xml para el mapa en su vertiente gráfica (los tiles) donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserCollisionsMap.h XMLParserCollisionsMap.cpp: Hereda de IXMLParser. Parseador xml para las colisiones del mapa, donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserQuests.h XMLParserQuests.cpp: Hereda de IXMLParser. Parseador xml para las misiones, donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserStateMachines.h XMLParserStateMachines.cpp: Hereda de IXMLParser. Parseador xml para las máquinas de estados finitos, donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserCharacter.h XMLParserCharacter.cpp: Hereda de IXMLParser. Parseador xml para personajes, donde deja la información leida en estructuras temporales que serán tratadas, si fuera necesario, en sus correspondientes sitios.

XMLParserPeople.h XMLParserPeople.cpp: Hereda de IXMLParser. Parsea en el mapa del nivel los personajes que se encuentra en dicho nivel.