3. Motores de juegos

3.1 Introducción a los motores de juegos

Un motor de juegos es un conjunto de sistemas de software extensibles y que facilita el desarrollo de un videojuego sin una gran modificación de código. Los motores de juego pueden ser:

- Motores de juego de propósito específico: Se desarrollan específicamente para un género de juegos.
- Motores de juego de propósito general: Abarcan un amplio género de juegos, su desarrollo tiene una mayor complejidad.

Desafortunadamente no existe un motor que pueda desarrollar cualquier tipo de juego, ya que se realizan optimizaciones específicas a los géneros y a las diferentes plataformas de hardware.

El término motor de juegos surge a mediados de los 90's como consecuencia de los juegos de primer persona como el increíblemente popular juego Doom por "id Software". Doom tenía una arquitectura que marca la separación entre componentes de software, recursos gráficos y reglas de juego. Otros desarrolladores podían comprar una licencia de los componentes de software y crear sus propios personajes, arte, interfaces, niveles, etcétera.

El valor de esta separación fue evidente cuando se desean crear juegos similares mediante la modificación del contenido gráfico y realizando un cambio mínimo a los componentes de software. De aquí también surge el término *modding*, donde grupos de jugadores, o desarrolladores independientes, agregan contenido a un juego.

La creación de un motor de juegos es un proceso complicado y largo, en la actualidad existen motores comerciales que venden licencias para permitir crear juegos con componentes de software ya existentes.

3.2 Desarrollo de un motor de juegos

El desarrollo de un motor de juegos es equivalente al desarrollo de un sistema informático. Un motor de juegos bien diseñado tiene las siguientes características (J. Flymnt, 2005; M. Robert, 2006; Sommerville, 2005):

- Mantenibilidad: El motor debe escribirse de tal manera que pueda evolucionar para cumplir con las necesidades de los creadores de juegos. Es inevitable un cambio, debido al uso de nuevas tecnologías y las crecientes expectativas por parte de los jugadores.

- Portabilidad: Involucra la facilidad de usar el motor en diferentes computadoras.

- Confiabilidad: La confiabilidad tiene un gran número de características, incluyendo la fiabilidad, protección y seguridad. El software seguro no debe causar daños en caso de una falla del sistema.

- Eficiencia: El motor no debe hacer que se malgasten los recursos del sistema, como la memoria y ciclos de procesamiento. Por lo tanto, la eficiencia incluye tiempos de respuesta y de procesamiento, utilización de la memoria, etcétera.

- Usabilidad: El motor de juegos debe ser fácil de utilizar para el desarrollo de juegos. Esto significa que debe tener una interfaz de usuario apropiada y una documentación adecuada.

- *Testability*: El motor de juegos debe buscar mantener sus componentes aislados, de esta manera se facilitan las pruebas de software.

Algunos síntomas de un diseño mal elaborado (M. Robert, 2006) son:

- Rigidez: Es la tendencia del software a hacer difícil el proceso de cambio. Un cambió mínimo produce una necesaria cascada de cambios en módulos dependientes. Esto genera una resistencia al cambio y el software se vuelve rígido.

- Fragilidad: Este síntoma está relacionado con la rigidez. Es la tendencia a romper el sistema cada vez que se realiza un cambio. Esto genera que el software se vuelve imposible de mantener; cada solución hace peor el problema, introduciendo nuevos errores.

- Inmovilidad: Es la incapacidad de reutilizar módulos para operaciones similares. El módulo necesario está escrito con demasiadas dependencias y resulta más fácil reescribirlo que utilizarlo.

- Viscosidad: En un proyecto viscoso el diseño del sistema es difícil de mantener. La viscosidad de software surge cuando existen diferentes maneras de realizar un cambio al software. Cuando es más difícil implementar un cambio manteniendo el diseño, que realizarlo mediante uno que no mantiene el diseño, un “hack”, es sencillo tomar la decisión inadecuada.

- Complejidad Innecesaria: La complejidad innecesaria surge cuando se agregan facilidades al código que anticipan cambios en los requerimientos. Esto puede parecer algo positivo,

ya que permite al código ser flexible; sin embargo, el efecto es el opuesto, el código se vuelve más complejo y difícil de entender.

- Repetición Innecesaria: El trabajo de cambiar el sistema se vuelve arduo cuando existe código repetido, además, es un indicador de una falta de abstracción.

- Opacidad: Es la tendencia de un módulo de volverse difícil de entender.

Estos síntomas pueden ser causados por las siguientes razones:

- En un proyecto no ágil, el diseño degrada si los requerimientos cambian.
- Cambios al código que no mantienen la filosofía del diseño inicial.
- Un mal diseño de arquitectura y de los principios de diseño orientados a objetos.
- Un análisis inadecuado de los requerimientos.

Aunque existen muchos modelos para el desarrollo de software como son: cascada, iterativo, ágiles, entre otros, algunas actividades son comunes para todos:

1. Especificación del Software: Se define la funcionalidad y las restricciones de su operación.
2. Diseño del sistema: Establece una arquitectura completa del sistema, identifica y describe las abstracciones fundamentales.
3. Implementación y validación: Se produce el software y se asegura que hace lo que el cliente desea.
4. Evolución del software: El software evoluciona para cubrir las necesidades cambiantes del cliente.

3.2.1 Requerimientos para el motor de juegos

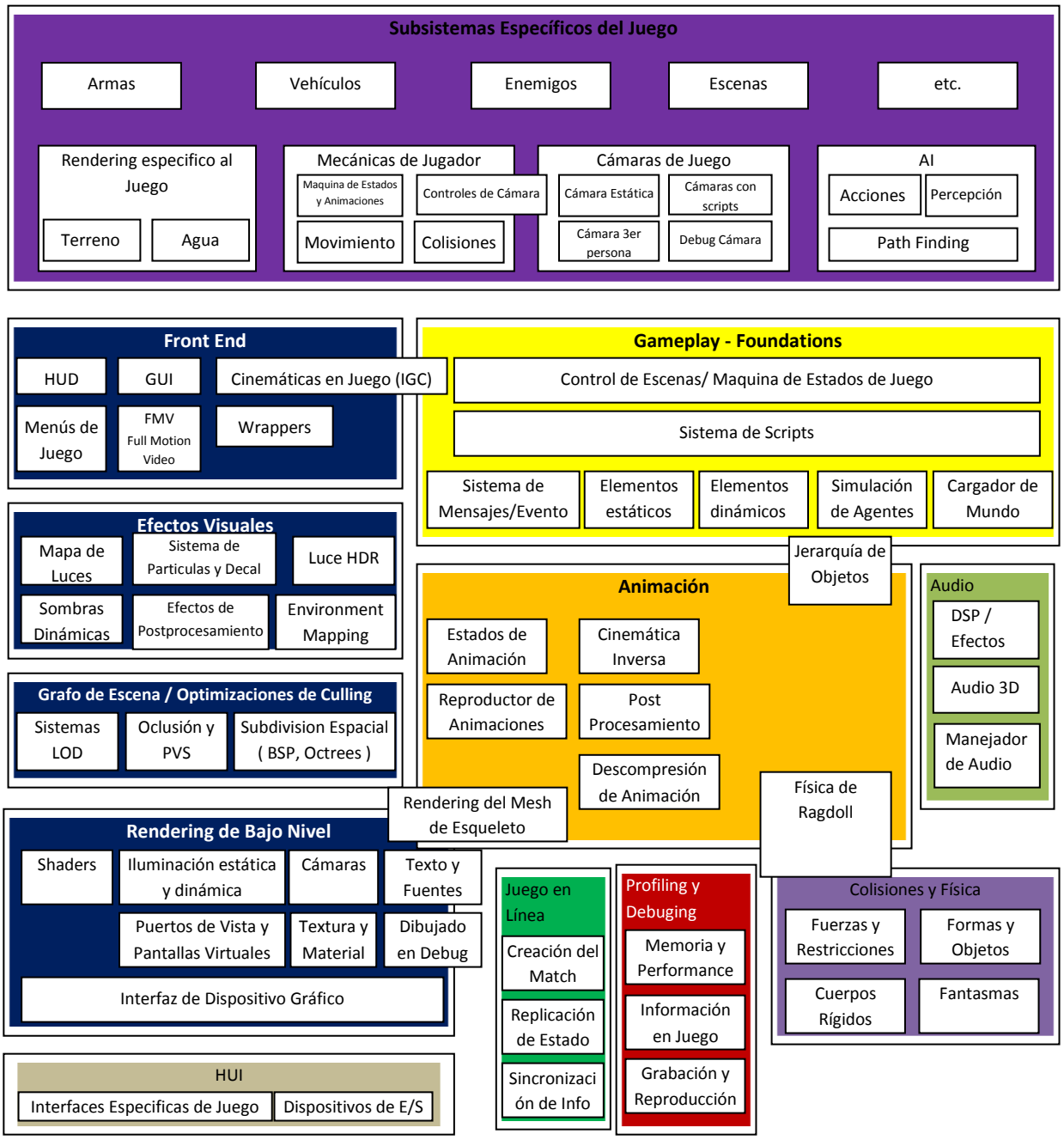
Algunas características que se deben definir antes de comenzar con el diseño del motor de juegos son:

1. Definir los géneros de juegos para los cuales se desarrollará.
2. Características funcionales y no funcionales de los juegos donde se utilizará.
3. Definir las áreas de funcionalidad del motor de juegos, como son: gráficos, audio, red, animaciones, etcétera.

3.2.2 Diseño de Arquitectura

Los grandes sistemas siempre se descomponen en subsistemas que proporciona algún conjunto de servicios relacionados (Sommerville, 2005). Algunos subsistemas del motor de juegos son llamados motores debido a su complejidad y extensión.

La presenta un diseño arquitectónico en capas de un motor propuesto por J. Gregory (J.Gregory, 2009):



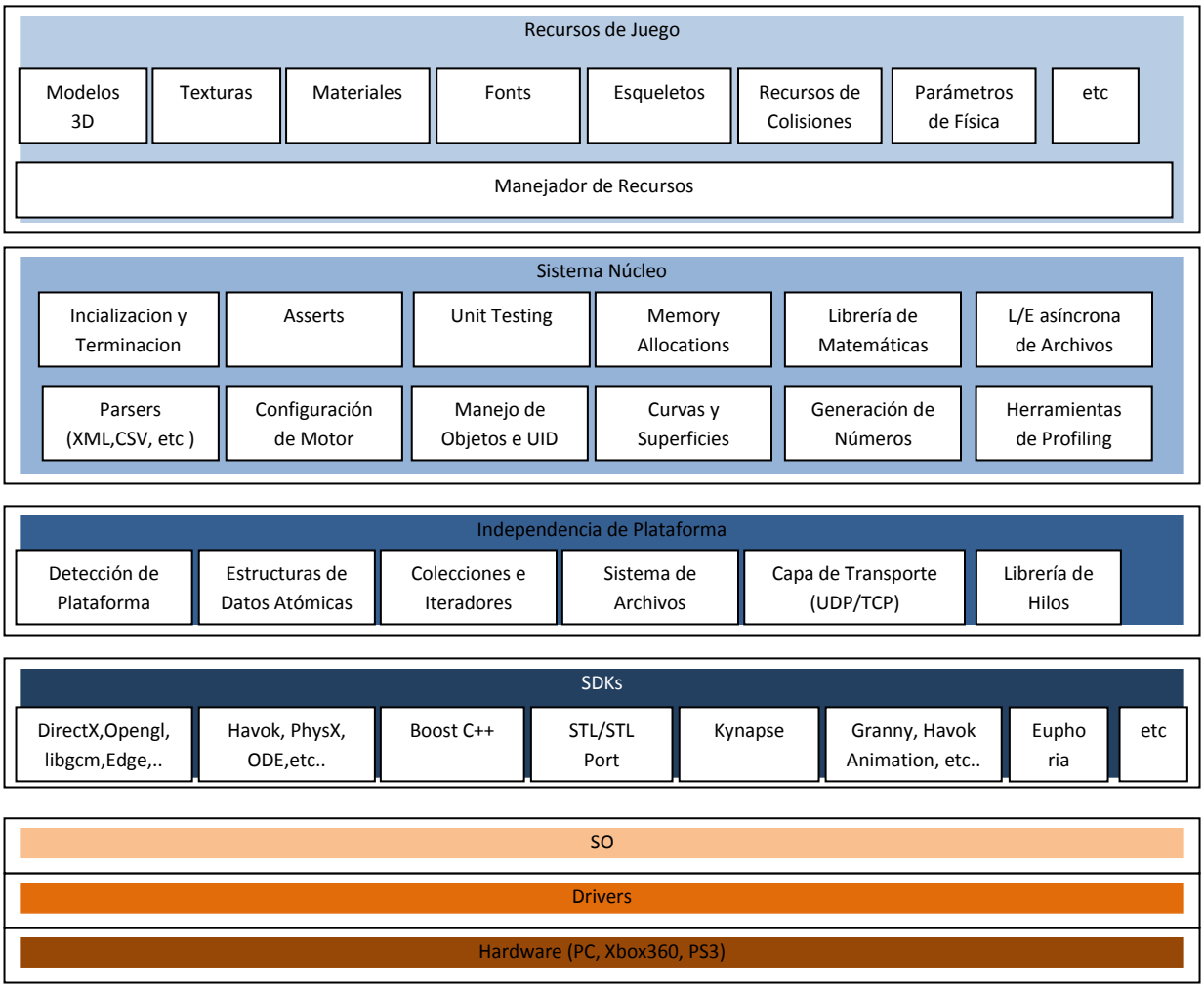


Figura 3.1. Arquitectura en capas de un motor de juegos

3.2.3.1 Hardware objetivo

Representa la computadora o consola destino del juego. Es importante tener en cuenta el dispositivo destino ya que se debe considerar la memoria, procesador, resolución, etcétera.

3.2.3.2 Drivers

Software de bajo nivel proporcionado por el sistema operativo o por el vendedor del hardware. El driver proporciona una abstracción del hardware y proporciona una interfaz para controlarlo.

3.2.3.3 Sistema operativo

En una computadora, el sistema operativo siempre se encuentra presente. Controla la ejecución de múltiples programas (entre ellos los juegos) y la administración de memoria.

En una consola, el sistema operativo es una librería pequeña que se compila dentro del ejecutable del juego. En las consolas, el juego prácticamente es dueño del sistema. Sin embargo, no es el caso en el Playstation 3 ni en el Xbox 360. El sistema operativo puede detener la ejecución del juego o utilizar ciertos recursos.

3.2.3.4 SDKs

La mayoría de los motores de juegos utilizan SDK de terceros y middleware. Un SDK (Software Development Kit) es una colección de librerías, API's y herramientas que le permite a un programador crear aplicaciones para un sistema concreto.

La Figura 3.2 muestra algunos ejemplos:

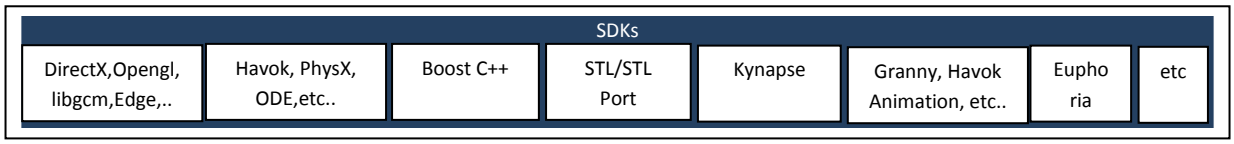


Figura 3.2. SDK's de terceros

Estructuras de Datos y Algoritmos

Como cualquier otro sistema de software, dependemos ampliamente de las estructuras de datos y algoritmos para manipularlos. Algunos ejemplos de librerías son:

- STL. La biblioteca de plantillas estándar implementa muchas estructuras de datos comunes y algoritmos para procesarlas, strings y streams de E/S.
- STLport. Es una versión portable y optimizada de STL.
- Boost. Es una librería con componentes poderosos y algoritmos, diseñada al estilo de STL.

Gráficos

La mayoría de los motores de *rendering* (dibujado a pantalla) se construyen sobre algunas de las siguientes librerías de interfaz de hardware:

- OpenGl fue inicialmente desarrollada por SGI. Es un estándar abierto utilizado en una amplia gama de dispositivos de hardware.
- DirectX es el SDK 3D de Microsoft, principal rival de OpenGL.
- Libgcm es una alternativa OpenGl que fue proporcionada por Sony. Es una interfaz de bajo nivel al hardware gráfico RSX del Playstation 3.

Colisiones y Física

Es un SDK para la detección de colisiones y física de cuerpos rígidos. Algunos de los SDK más conocidos son:

- Havok es muy popular en la industria y es considerado un motor de física y colisiones.
- PhysX es desarrollado por NVIDIA y su uso es gratuito.
- Open Dynamics Engine (ODE) es un SDK de código abierto.

Animación

- Granny incluye exportadores de modelos y animaciones de los paquetes de modelado 3D y animación más comerciales: Maya, 3D Studio Max, etc.
- Havok Animation la compañía desarrolladora del motor de física Havok que facilita la interacción entre animaciones y física.
- Endorphin y Euphoria son paquetes de animación que producen animaciones avanzadas usando modelos biomecánicas del movimiento humano.

Inteligencia Artificial

- Kynapse. La inteligencia artificial era desarrollada específicamente para cada juego, pero, la compañía Kynogon desarrollo Kynapse que contiene bloques de inteligencia artificial como son: búsqueda de caminos, obstrucciones (object avoidance), identificación de vulnerabilidades.

3.2.3.5 Independencia de plataforma

La mayoría de los motores de juegos deben correr en más de una plataforma de hardware, es por esto que la mayoría de los motores de juegos contienen una capa de Independencia de Plataforma.

Aseguramos una independencia de plataforma al remplazar o englobar las llamadas más comunes a funciones de C, llamadas de sistema operativo y otras API's.

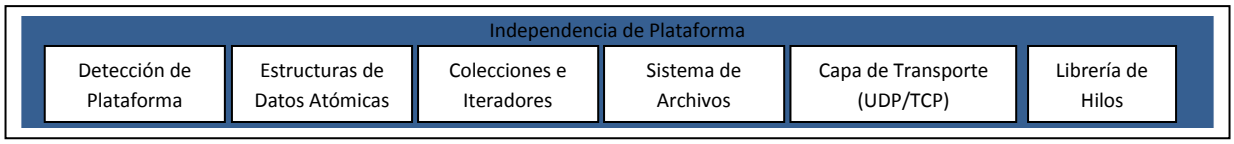


Figura 3.3. Capa de independencia de plataforma

3.2.3.6 Sistema Núcleo

Contiene librerías y utilidades de software sobre las cuales se extiende la funcionalidad de nuestro motor, algunas son:

- Los asserts son líneas de código que nos permiten verificar errores lógicos y/o suposiciones que no se cumplen.
- Manejo de Memoria. Permite un uso rápido y eficiente de la liberación y asignación de memoria.
- Librería de Matemáticas. Los juegos usan una gran cantidad de funciones matemáticas. Estas librerías nos proporcionan herramientas como operaciones con vectores, matrices, quaterniones, rotaciones, trigonometría, etc. Además, las librerías pueden estar optimizadas.

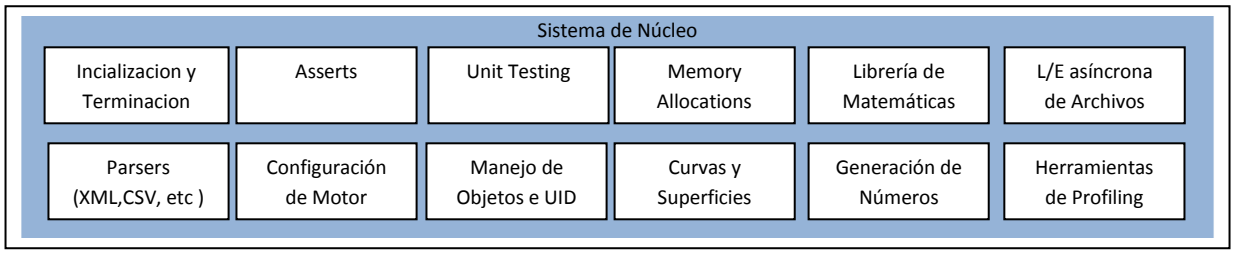


Figura 3.4. Capa de sistema de núcleo

3.2.3.7 Recursos de Juego

El motor de juegos debe ser capaz de administrar y cargar diferentes tipos de recursos como: texturas, modelos, fuentes, audio, video, etcétera.

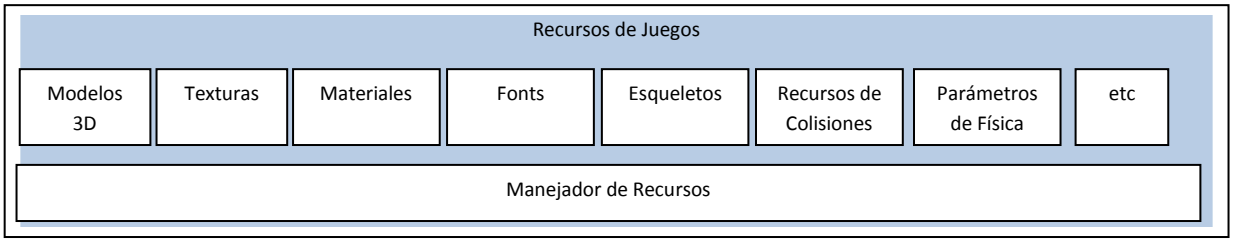


Figura 3.5. Capa de manejo de recursos

El manejador de recursos es el encargado de:

- Debido a que la memoria es limitada, debe asegurarse que los recursos de juego no se repitan y se encuentren presentes en el momento de su uso.
- Proporcionar las interfaces para al acceso a los recursos del juego.
- Envolver las llamadas de acceso al sistema de archivos y proporcionar funciones de alto nivel.
- Manejo de diferentes medios de almacenamiento como discos duros, usb, dvd, archivos en red, etc.

En algunos motores el manejador de recursos es un sistema unificado y centralizado que maneja todo tipos de recursos (Unreal, Ogres Resource Manager). En otros motores el manejador de recursos no existe como un sistema centralizado, al contrario, se conforma de diferentes subsistemas que manejan recursos específicos.

3.2.3.8 Motor de Render

El motor de *render* es uno de los más grandes y complejos componentes en cualquier motor de juegos. El diseño del motor de rendering depende del hardware de gráficos y la librería que se utiliza para interactuar con él.

3.2.3.8.1 Rendering de Bajo Nivel

El objetivo de este componente es dibujar una colección de primitivas los más rápido y visualmente atractivo posible, sin importar que partes de la escena son visibles.

Figura 3.6. Subsitema de rendering de bajo nivel

- Interfaz de Dispositivo Gráfico. Nos facilita la inicialización y selección del dispositivo gráfico, además nos permite la creación de superficies de rendering (depth buffer, color buffer, esténcil buffer, etcétera), entre otras cosas.

Los otros componentes de la capa de rendering realizan un trabajo en cooperación para obtener los datos finales que serán enviados al hardware gráfico, esto incluye triángulos, materiales, textura, matrices de cámara, proyección, etc. Además, se busca mandar esta información a la tarjeta de gráficos de manera eficiente.

3.2.3.8.2 Grafo de Escena / Optimizaciones de Culling

Es un componente del motor de rendering que optimiza la cantidad de primitivas que se mandan al Rendering de Bajo Nivel. Esto se logra al descartar los objetos no visibles (están obstruidos o fuera del área de vista).

Existen diferentes maneras de lograrlo: se puede utilizar un frustrum culling, subdivisión espacial (árbol de BSP, quadtree, octree, kd-tree, etcétera), niveles de detalle LOD.

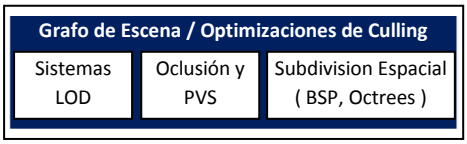


Figura 3.7. Un típico grafo de escena y subdivisión espacial para optimizaciones de culling

3.2.3.8.3 Effectos Visuales

Los actuales motores de juegos soportan una gran variedad de efectos visuales, algunos son:

- Sistemas de partículas para efectos de disparos, humo, agua, etcétera.
- Sistemas de decal para efectos de huellas, hoyos de balas, etcétera.
- Mapas de iluminación y de ambiente para simular el efecto de la iluminación.
- Sombras dinámicas que se pueden logran con shadow mapping, shadow volumes, etcétera.
- Efectos de postprocesamiento de la escena, como blur, bloom, etcétera.

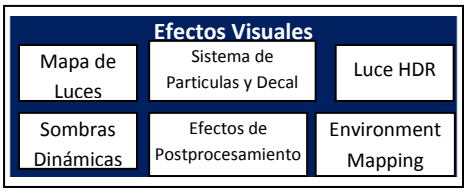


Figura 3.8. Subsistema de efectos visuales

Algunos efectos se pueden implementar dentro del motor de rendering o como un componente adicional que afecta los buffers de salida.

3.2.3.8.4 Front End

La mayoría de los juegos despliega imágenes 2D sobre una escena 3D para mostrar información al jugador. Algunos elementos que se despliegan son:

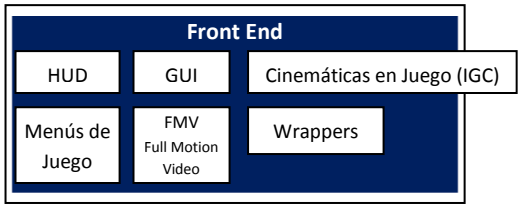


Figura 3.9. Subsistema de Front End

- El HUD (Heads Up Display) muestra información de vidas, armas, puntaje, etcétera.
- Menús en el juego
- Puede tener un GUI (Graphical Users Interface) que permite al jugador modificar inventario, configurar unidades, etcétera.
- IGC's, Cinemáticas en Juego, permiten mostrar animaciones dentro del juego.

3.2.3.9 Herramientas de profiling y debugging

Los juegos son sistemas de tiempo real, es importante contar con herramientas que nos permitan evaluar la eficiencia del sistema para realizar las optimizaciones pertinentes. Es importante evaluar el uso de memoria (heap y stack), tiempo de ejecución del juego (CPU y GPU).

Figura 3.10. Subsistema de profiling y debugging

Algunas herramientas de profiling son:

- CLR Profiler permite ver el uso de memoria heap y stack para sistemas desarrollados con el .NET Framework.
- PIX es un debuger y analizador de performance de CPU y GPU, para DirectX.
- JetBrains es una empresa que desarrolla software para el análisis de memoria y performance para diferentes lenguajes de programación.
- Intel's VTune

La mayoría de los motores incorpora su propio sistema de profiling y debuging. Algunos elementos útiles que vale la pena tener son:

- Mecanismo para evaluar tiempos de ejecución de secciones de código.
- Posibilidad de mostrar en la pantalla las estadísticas de profiling mientras el programa se encuentra en ejecución.
- Guardado de resultados de profiling a archivos de texto o Excel.
- Mecanismos para determinar el uso de memoria por el motor y subsistemas.
- Mecanismos para guardar eventos del juego y la posibilidad de reproducirlos.

3.2.3.10 Colisiones y física

La detección de colisiones es importante para todos los juegos, ya que es la forma de interactuar con el mundo virtual de manera realista. El sistema de física también es llamado motor de física debido a su complejidad y extensión.

Los motores de física por lo común usan dinámica de cuerpos rígidos, ya que únicamente nos encontramos interesados en la cinemática de cuerpos rígidos y las fuerzas que causan el movimiento.

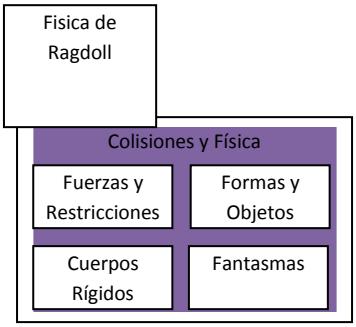


Figura 3.11. Subsistema de colisión y física

En la actualidad pocas empresas desarrollan su propio motor de física, lo común es integrar un SDK al motor, algunos son: Havok, PhysicsX de NVidia.

3.2.3.11 Animaciones

La mayoría de los juegos presentan personajes animados o inanimados que interactúan con el mundo virtual. El uso de animaciones con los personajes da una mayor atracción visual y mantiene el interés del jugador.

Algunos tipos de animaciones utilizadas en juegos son:

- Animaciones por sprites/texturas 2D.
- Animaciones por jerarquías de cuerpos rígidos.

- Animaciones por esqueleto.
- Animaciones de vértices.
- Morphing.

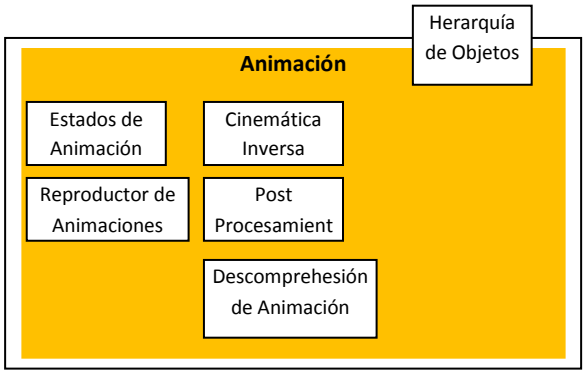


Figura 3.12. Subsistema de animación

3.2.3.12 Dispositivos de Interacción con usuario

Human Interface Device, HID, procesa la entrada por parte del jugador, algunos dispositivos incluyen: teclado, mouse, joystick, controles, otros dispositivos especializados como volantes, wiimote, etcétera.

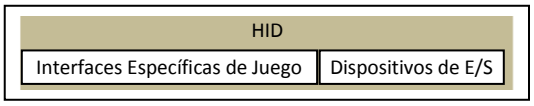


Figura 3.13. Subsistema de interacción con usuario

El subsitema de HID, por lo común, maneja detalles de bajo nivel de los dispositivos y proporciona información de alto nivel como: botones presionados, botones no presionados, interpreta la información de los acelerómetros, información de dirección de joystick, etcétera; además, puede tener lógica para manejar secuencias de botones, presión de múltiples botones, *gestures* (movimientos específicos en touchpads).

3.2.3.13 Audio

El audio es un subsistema que por lo general recibe menor atención en un motor de juegos. Para plataformas de DirectX (PC y Xbox 360) Windows proporciona una herramienta muy útil llamada XACT. Electronic Arts desarrollo un motor de audio llamado SoundR!iot.

Figura 3.14. Subsistema de audio

En caso de que se use un motor de audio ya existente, aún se necesita desarrollo de software para la integración, ajustes y detalles para producir sonido de alta calidad.

3.2.3.14 Multijugador/Red

El uso de multijugadores pude tener un profundo impacto en algunos componentes del motor de juegos. Es importante considerar esta funcionalidad durante el diseño del motor.

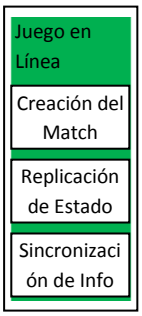


Figura 3.15. Subsistema de juego en red

3.2.3.15 Sistemas de Gameplay - Foundations

El termino gameplay significa la acción que sucede dentro del juego, las reglas del mundo virtual, la mecánica del jugador (sus habilidades) y otras entidades en el mundo, y los objetivos del jugador.

El gameplay por lo común se implementa en el leguaje con que se desarrolla el motor, un lenguaje de scripting o una combinación de ambos. Para separar el código del motor con el juego podemos introducir la capa de **Gamplay – Foundations** que proporciona los componentes sobre los cuáles la lógica específica del juego se puede implementar.

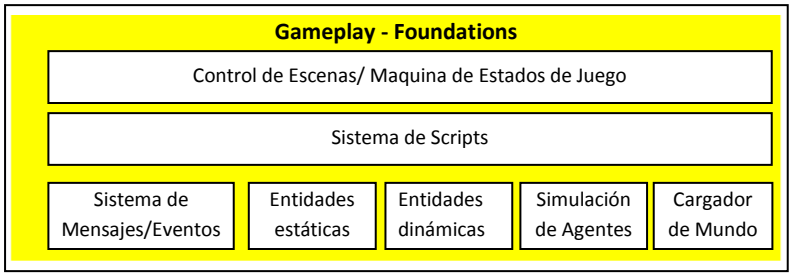


Figura 3.16. Subsistema de Gameplay-Foundations

Algunos componentes de este subsistema son:

- Entidades estáticas y dinámicas. Las entidades del mundo virtual (objetos animados e inanimados) son modelados con la programación orientada a objetos. El modelo de entidades permite tener una colección heterogénea de objetos y facilita su uso mediante polimorfismo.

  Algunas entidades típicas pueden incluir:

  - Entidades estáticas como rocas, edificios, terreno, fondo, etcétera.
  - Entidades dinámicas que utilizan cuerpos rígidos para interactuar con el usuario.
  - Personaje principal (Player Character, PC).
  - Personajes no controlados por el usuario (NPC, Non player characters).
  - Armas
  - Cámaras

- El modelo de objetos define las características de las entidades usadas, algunas son: identificadores únicos para cada objeto, tiempo de vida de los objetos, simulación de los estados de los objetos, si están diseñados con programación orientada a objetos.

- El sistema de Mensajes y Eventos establece como se realiza la comunicación entre objetos. Un objeto puede simplemente llamar una función de otro objeto para mandar el mensaje, en cambio, en un motor que usa un sistema de mensajes y eventos, un objeto crea un nuevo mensaje que posteriormente se entrega al destinatario.

- El sistema de scripts evita la necesidad de recompilar y “re-linkear” cada vez que se hace un cambio a los scripts del juego. Un script puede tener información de la lógica del juego, estados de personajes, descripción de mundos, etcétera.

- El control de escenas permite agregar nuevas escenas como: menú, instrucciones, opciones, etcétera. La máquina de estados facilita la creación de estados para las entidades del juego.

3.2.3.16 Sistemas Específicos del Juego

Los sistemas específicos del juego son diferentes para cada juego, esta es una capa que separa el juego del motor de juegos. En la práctica, esta capa nunca es perfectamente clara ya que algunos elementos se pueden encontrar en los sistemas de gameplay o ya en el juego.

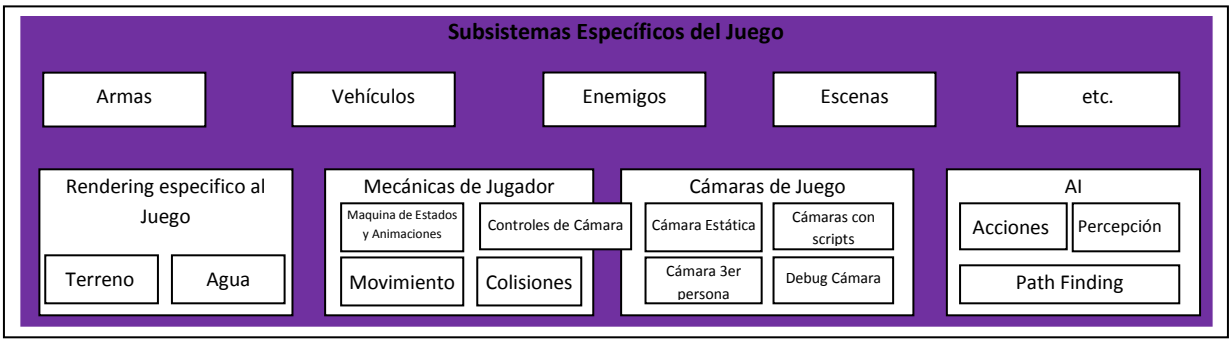


Figura 3.17. Sistema específico de juego.

3.2.3 Proceso de Diseño Orientado a Objetos

Antes de comenzar a desarrollar el motor de juegos, se deben identificar los objetos principales del sistema y desarrollar modelos de diseño basados en el diseño de arquitectura. Los modelos de diseño son el puente entre los requerimientos y la implementación del sistema, deben ser abstractos para que el detalle innecesario no oculte las relaciones entre objetos. Sin embargo, también deben incluir suficiente detalle para que los programadores tomen las decisiones de implementación.

Para desarrollar los modelos e implementarlos es importante hacer uso de los principios de diseño de clases orientadas a objetos y conocer patrones de diseño.

3.2.3.1 Principios de Diseño de Clases Orientadas a Objetos

- Single Responsibility Principle (DeMarco, 1979)

El single responsability principle, también llamado cohesión, define que una clase debe tener una sola razón de cambio. Es importante que una clase tenga una sola responsabilidad, debido a que cada responsabilidad es susceptible a un cambio por modificación de requerimientos.

Si una clase tiene más de una responsabilidad, las responsabilidades se vuelven dependientes y cualquier cambio a alguna de ellas puede afectar a la otra.

- Open Closed Principle (M. Robert, 2006)

Entidades de software (clases, módulos, funciones, etcétera) deben estar abiertas para extensión pero cerradas para modificación. Este principio puede sonar contradictorio; sin embargo, existen técnicas de programación basadas en abstracción, polimorfismo y encapsulamiento que nos permiten lograrlo.

- The Liskov Substitution Principle (Data Abstraction and Hierarchy, 1998)

Las clases derivadas deben poder sustituir a la clase base. Esto quiere decir la clase derivada debe funcionar de manera similar a la clase base y si alguna clase derivada no proporciona la misma funcionalidad que la clase base es mejor crear una nueva clase.

- The Dependency Inversion Principle (M. Robert, 2006)

Es la estrategia de depender de funciones y clases abstractas o interfaces, en vez de concretas. Además, los módulos de alto nivel no deben depender de módulos de bajo nivel, ambos deben depender de abstracciones.

Un modelo inapropiado está representado por la Figura 3.19. La capa de alto nivel *Policy* depende de la capa de bajo nivel *Mechanism*, a su vez, Mechanism depende de la capa de *Utility*. Es inapropiado porque un cambio en las capas bajas requiere modificar todas las capas de mayor nivel.

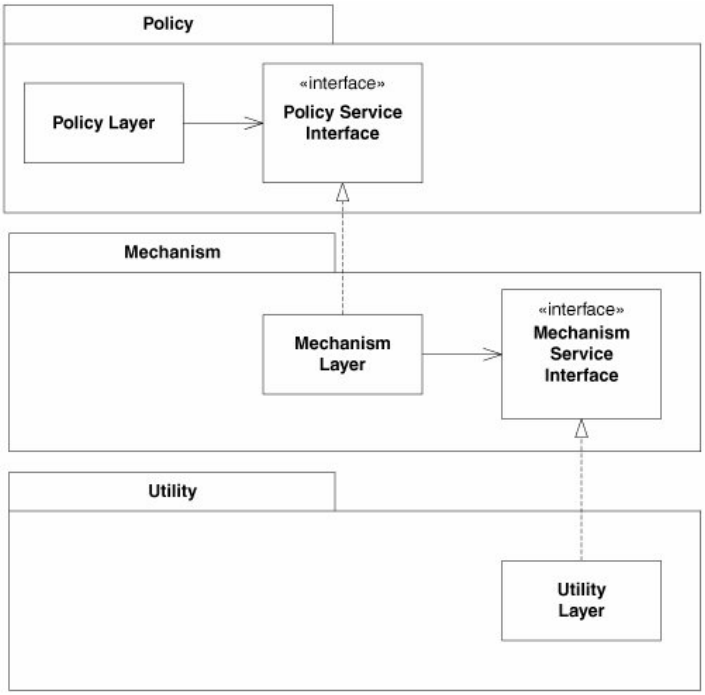


Figura 3.18 Modelo donde se usa el Dependency Inversion Principle

Un modelo que cumple con el Dependecy Inversion Principal está representado por la Figura 3.18. Cada capa de alto nivel declara una interfaz de los servicios que requiere. Las clases de bajo nivel implementan la interfaz y cualquier cambio en éstas no afecta a las capas altas.

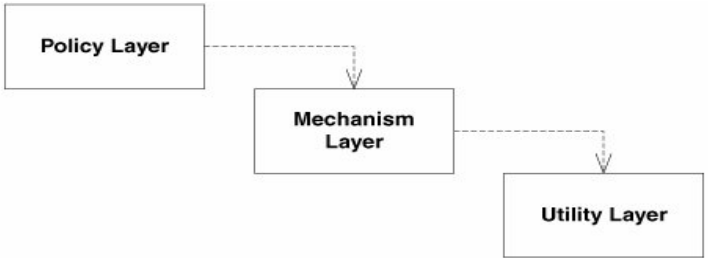


Figura 3.19. Modelo donde las clases de alto nivel depende de las clases de bajo nivel

- The Interface Segregation Principle (M. Robert, 2006)

El *interface segregation principle* asume que existen clases que no pueden cumplir con cohesión, son *fat* interfaces, y se pueden descomponer en grupos de métodos.

Este principio sugiere que los clientes no deben conocer los métodos como una sola clase, en cambio, es conveniente que los clientes conozcan las clases abstractas que tienen interfaces cohesivas, por ejemplo:

La Figura 3.20 muestra que las clases Deposit Transaction, Withdrawal Transaction y Transfer Transaction requieren de la interfaz UI para realizar la transacción adecuada. La interfaz UI es una fat interface y es precisamente lo que ISP nos recomienda evitar. Si creamos una nueva transacción, se deben agregar los métodos a UI y esto afectará a todas las clases de transacción.

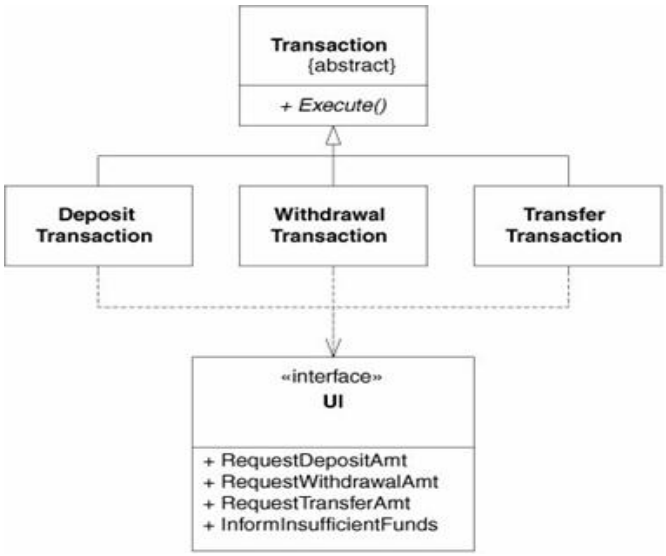


Figura 3.20. Fat interface

Un modelo que cumple con el Interface Segregation Principle se muestra en Figura 3.21. La fat interface UI se separa en interfaces individuales: DepositUI, WithdrawalUI y TransferUI. Además, cada transacción depende de su interfaz específica.

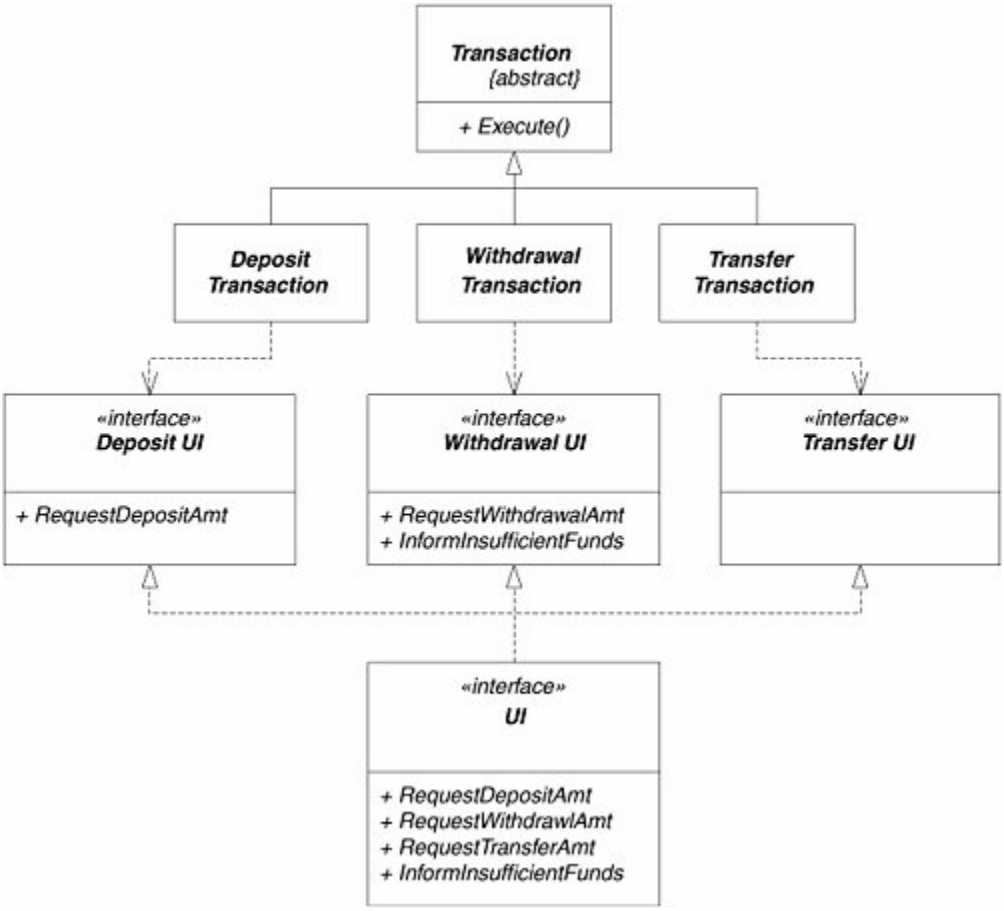


Figura 3.21. Fat interface que cumple con ISP

- Avoid Coupling (J. Flymnt, 2005)

Este principio establece que debemos evitar dependencias con otros elementos.

3.2.3.2 Patrones de Diseño

Los patrones de diseño en el contexto de la ingeniería de software describen una solución elegante a los problemas de diseño de clases orientadas a objetos.

El uso de patrones de diseño permite hacer diseños flexibles, modulares, reusables, confiables y fáciles de utilizar. En el libro de Gamma et. al. (E. Gamma, 1995) se puede encontrar más información sobre los patrones de diseño. Algunos patrones de diseño comúnmente utilizados en el desarrollo de motores de juegos y videojuegos son:

- Abstract Factory

La *abstract factory* Proporciona una interfaz para crear objetos de familias de clases relacionadas o dependientes; sin especificar la clase concreta a utilizar.

Un ejemplo se muestra en la Figura 3.22 es un sistema de sonido basado en (Llopis, 2008) que permite crear diferentes sistemas de sonido, dependiendo del hardware instalado.

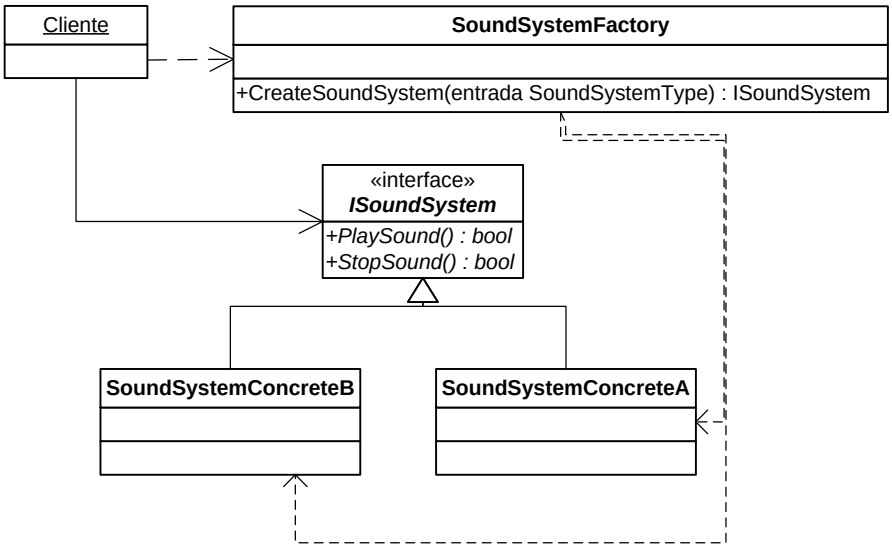


Figura 3.22. Sistema de sonido que usa abstract factory

- Singletone

En el patrón *singleton* aseguramos que una clase tenga una sola instancia, y proporcione un punto de acceso global.

El modelo de la Figura 3.23 muestra una clase que usa el patrón singleton.

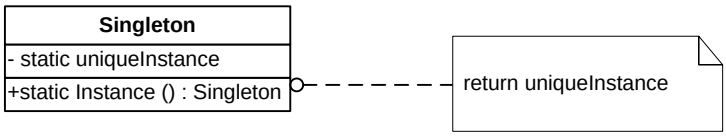


Figura 3.23. Modelo UML de un singleton

- Facade

El patrón *facade* provee una interfaz unificada para un conjunto de interfaces dentro de un subsistema. La Figura 3.24 muestra un facade que define una interfaz de alto nivel para facilitar el uso del subsistema.

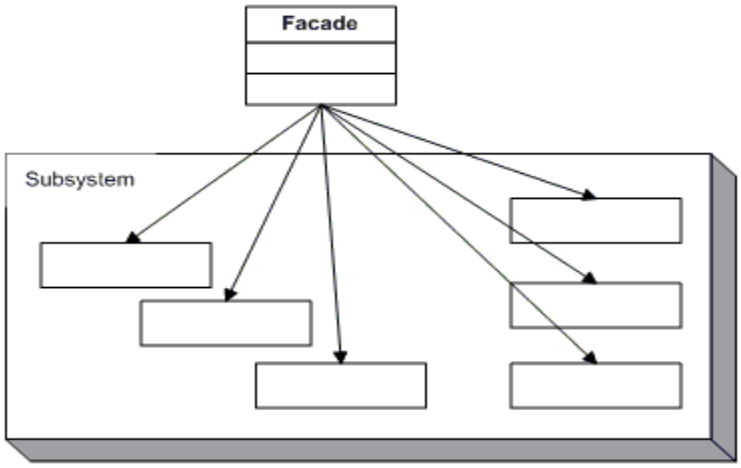


Figura 3.24. Modelo que usa facade

- Observer

En el patrón de *observer* una lista de observers se registran con un sujeto. Cuando el sujeto realiza un cambio a su estado interno, notifica a los observers que se sincronizan con el sujeto en cuestión.

La Figura 3.25 muestra un modelo del patrón observer. La clase sujeto permita agregar o remover observadores. Cuando el sujeto concreto cambia de estado llama a la función Notify() que avisa del cambio a los observadores.

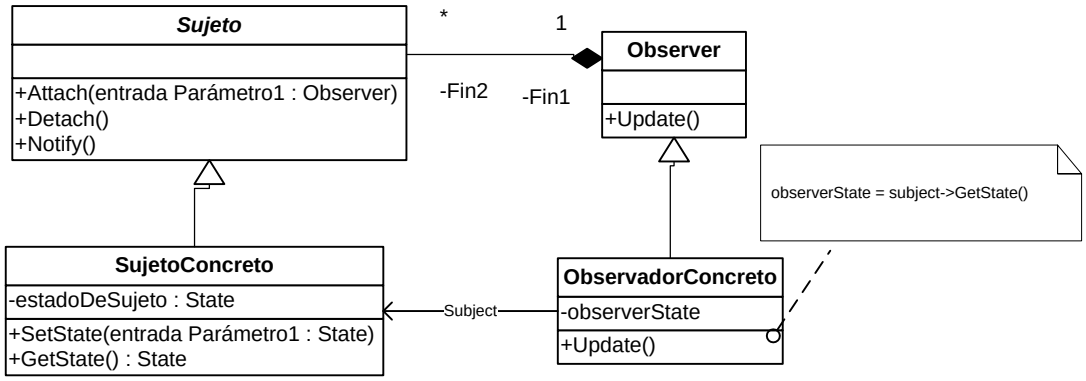


Figura 3.25. Patrón observer

- Composite

Compone a los objetos en una estructura de árbol que representa una jerarquía. Composite permite a los clientes usar objetos individuales como compuestos.

La Figura 3.26 muestra que un Componente puede ser de tipo Composite o Leaf. El tipo Composite puede estar compuesto por uno o más componentes.

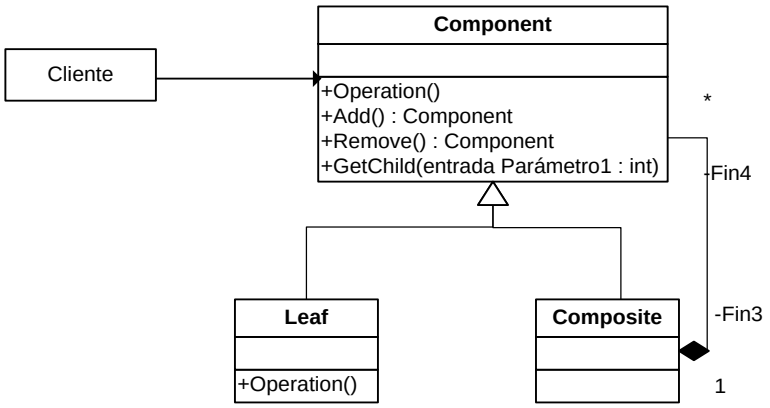


Figura 3.26. Modelo UML del patrón Composite

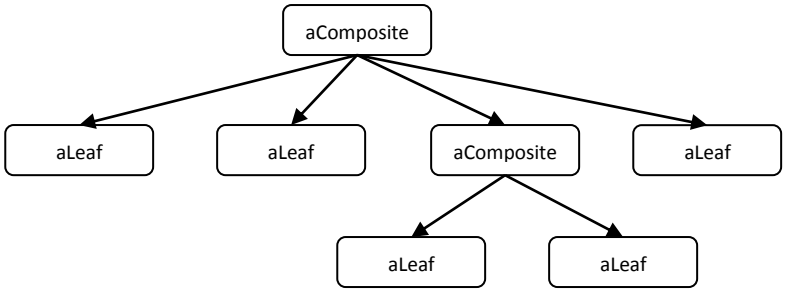


Figura 3.27. Representación de un árbol de componentes

3.3 Tipos de Motores

Anteriormente se dijo que algunos subsistemas del motor de juegos son llamados motores debido a su complejidad y extensión.

En la actualidad existe una gran cantidad motores comerciales y gratuitos, es importante conocer el estado actual de la tecnología y las posibles herramientas que podemos usar para desarrollar nuestro juego o motor de juegos.

3.3.1 Motores Generales

RAGE Engine.- Motor comercial desarrollado por Rockstar San Diego. Se comenzó a desarrollar RAGE (Rockstar Advanced Game Engine) en 2004. Permite utilizar mundos extensos y tiene una compleja IA. Utiliza Natural Motion para las animaciones y para la física usa Bullet.

Se usó en los siguientes juegos: Rockstar Presents Table Tennis, GTA IV + Episodes, Midnight Club: Los Angeles, Red Dead Redemption, L.A. Noire.

CryENGINE.- Motor comercial de desarrollo para las plataformas Xbox, PS3, DX9 y DX10. No utiliza middleware de otros desarrolladores, maneja su propia física, audio y animaciones.

Se usó en los siguientes juegos: Far Cry, Crysis, Crysis Warhead, Crysis 2, Aion: Tower of Eternity

Naughty Dog Game Engine.- Motor comercial desarrollado específicamente para el PS3. Maneja una infinidad de objetos dinámicos con física independiente, interacción ambiente-animación, efectos de iluminación y AI. Tiene transiciones entre cinemáticas y juego casi inidentificable, además de que soporta co-op y multiplayer.

Se usó en los siguientes juegos: Uncharted: Drake's Fortune, Uncharted 2: Among Thieve

Unreal Engine.- Motor comercial desarrollado por Epic Games, su núcleo fue desarrollado en C++ y funciona en las plataformas: Dreamcast, Xbox, Xbox 360, PlayStation 2 y PlayStation 3.

La última versión del Unreal Engine es la UE3 y usa: Microsoft's DirectX 9 para Windows XP, Windows Vista, Windows 7 and Xbox 360; DirectX 10 y DirectX 11 para Windows Vista y Windows 7; OpenGL para Linux, Mac OS X and PlayStation 3. Utiliza PhysX para la física y colisiones.

Se usó en los siguientes juegos: Gears of War, Mass Effect, BioShock, Unreal Tournament, Deus Ex, GRAW, Red Steel, Borderlands, Brothers in Arms, Homefront, Mirror's Edge, Singularity, Rainbow Six: Vegas, etcétera.

Panda3D.- Motor gratuito desarrollado en C++ y puede utilizar Python para agregar complementos. Incluye gráficos, audio, I/O, detección de colisiones y otras herramientas.

Delta3D.- Motor de juegos que integra otros proyectos de código abierto en una API sencillo de utilizar, como son: OpenSceneGraph, Open Dynamics Engine, Cal3D y OpenAL.

3.3.2 Motores Gráficos

id Tech 1.- Conocido como motor de Doom. Creado por John Carmack y fue un motor que revolucionó la industria en su época.

OGRE.- Motor de visualización de gráficos, es uno de los motores de gráficos más prominentes. Soporta las API de gráficos Direct3D y OpenGL, y se ejecuta en plataformas Windows, Linux y Mac. Se desarrolló en C++ y existen muchos complementos que permiten integrar motores de sonido, física, colisiones, red, inteligencia artificial, etcétera.

Irrlicht Engine.- Motor de gráficos 3D escrito en C++. Funciona en diferentes plataformas como son Mac OS X, Linux y Windows además Xbox, PSP, SymbianOS y el iPhone. Soporta OpenGL, DirectX 8 y 9, OpenGL ES. La comunidad desarrollo interfaces para el SDL, iPhone y SymbianOS.

OpenSceneGraph.- Es una herramienta de gráficos 3D. Se desarrolló con OpenGL y C++. Funciona en una variedad de sistemas operativos Windos, Mac OS X, Linux, IRIX, Solaris and FreeBSD.

Aleph One.- Motor de juegos para shooters 3D. Desarrollado por Bungie antes de ser comprador por Microsoft. Se desarrolló con C y su principal plataforma es Mac, Windows y Linux.

Axiom Engine .- Motor de visualización de gráficos en 3D. Desarrollada con C# para ser utilizada con .NET y Mono. Provee una abstracción completa del API 3D, Soporta DirectX y OpenGL, contiene un modelo scene graph y soporta shaders complejos.

3.3.3 Motores de Colisiones y Física

Havok Physics.- Es un motor comercial muy popular que se desarrolló en C/C++. Dependiendo del producto que se desea desarrollar, se puede conseguir la versión gratuita o la versión comercial. La versión actual 7.1 funciona en Xbox y Xbox 360; Wii; Sony's PlayStation 2, PlayStation 3 y PlayStation Portable; Linux; y en Mac OS X. Se usó en los siguientes juegos: Too Human, Alone in the Dark, Assassin’s Creed, Bio Shock, Halo, Starcraft 2 y muchos más.

PhysX.- Conocido anteriormente como NovodeX. Actualmente es propiedad de Nvidia y se distribuye de manera gratuita o comercial dependiendo del producto que se desea desarrollar. Funciona en Windows 7, Windows Vista, Windows XP, Mac OS X, Linux, Wii, PlayStation 3, Xbox 360. Permite el acelerar el procesamiento de la física pasando algunos cálculos al GPU, permitiendo al CPU realizar otros cálculos. Se usó en los siguientes juegos: Batman: Arkham Asylum, Mirror's Edge, Tom Clancy's Ghost Recon Advanced Warfighter 2, Unreal Tournament 3, Mafia II y muchos más.

ODE.- Open Dynamics Engine, es un motor de física de código abierto cuya principal funcionalidad es la simulación de cuerpos rígidos y colisiones. Utiliza un API C/C++ y fue desarrollado por Russell Smith, el motor tuvo mucha popularidad en 2005-2006, pero actualmente no se encuentra en desarrollo.

Bullet.- Motor de física de código abierto para objetos 3D, tiene una licencia gratuita zlib. Desarrollado por Erwin Coumans, un ex trabajador de Havok. Se ha utilizado en juegos como:

Grand Theft Auto IV, Madagascar Kartz, Regnum Online, etcétera. Además es utilizado en películas.

Box2D.- Motor de física gratuito para objetos en 2D hecho con C++. Se puede utilizar con los siguiente lenguajes Ada, C++, C#, D, Lisp, Lua, Mercury, Pascal, Perl, Python, Scheme. Se ha utilizado en los siguientes juegos *Crayon Physics Deluxe*, *Rolando*, *Fantastic Contraption*, Incredibots y muchos juegos flash.

Newton Game Dynamics.- Permite realizar la simulación de cuerpos rígidos. Es mucho más específico para detectar la colisión, sin embargo sacrifica velocidad.

3.3.4 Motores de Animación

Cal3D.- Cal3D es una librería de animaciones que permite cargar, reproducir y mezclar animaciones personajes 3D con esqueleto. Se desarrolló con C++ y es independiente de un API de gráficos.

Euphoria.- Euphoria es un motor de animación creado por NaturalMotion basado en *la Dynamic Motion Synthesis* (Síntesis de Movimiento Dinámico), la tecnología genera animaciones sobre la marcha usando una completa simulación del cuerpo, músculos y sistema nervioso. En lugar de utilizar animaciones predefinidas, los personajes, las acciones y reacciones se generan en tiempo real.

Endorphin.- Endorphin es un programa de síntesis de movimiento dinámico desarrollado por Natural Motion. Endorphin puede ser usado para simular física con objetos simples y para crear una animación 3D utilizando "comportamientos", los cuales son una serie de movimientos predeterminados que pueden ser usados en un personaje. El programa hace interactuar las animaciones con el resto del escenario, logrando que no se produzcan penetraciones entre objetos 3D. A diferencia de Euphoria, Endorphin no es un motor, sino que es un software para crear animaciones.

Havok Animation.- Es un motor de animación desarrollado por Havok. Dependiendo del producto que se desea desarrollar, se puede conseguir la versión gratuita o la versión comercial. Facilita la integración de animaciones con Havok Physics.

Granny.- Incluye exportadores de modelos y animaciones de los paquetes de modelado 3D y animación comerciales, como son: Maya, 3D Studio Max, etc.

3.3.5 Motores de Inteligencia Artificial

Kynapse.- La compañía Kynogon desarrollo Kynapse que contiene bloques de inteligencia artificial como son: búsqueda de caminos, detección de obstrucciones (*object avoidance*), identificación de vulnerabilidades, algoritmos de movimiento grupal y trabajo en equipo. Actualmente se encuentra en desarrollo por Autodesk.

Havok AI.- Es un motor comercial de inteligencia artificial desarrollado por Havok que contiene herramientas para: búsqueda de caminos, detección de ambiente, movimiento grupal y facilita la interacción con otras herramientas de Havok.

Navpower.- Motor de inteligencia artificial comercial para las consolas de Xbox 360 y Playstation3. Permite crear y modificar superficies donde se puede caminar, algoritmos de búsqueda de camino, y movimientos grupales.

EKIOne.- Es un motor comercial de inteligencia artificial que permite la creación de NPC´s inteligentes y con emociones, algortimos de planeación y decisión, búsqueda de camino y diferentes modos de percepción del entorno.

3.3.6 Motores de Audio

FMOD.- Es un API para la creación y reproducción de audio interactivo. Soporta una gran cantidad de sistemas operativos y plataformas.

Wwise.- Es un motor de audio que permite la creación, integración y administración de audio. Algunas de sus funcionalidades son: creación de bancos de sonido para juegos, combinar niveles de audio, herramientas de profiling y definición de audio ambiental.

Q3D Interactive.- Este motor se utiliza para sonido 3D, permite posición el sonido en videojuegos y otras aplicaciones de realidad virtual.