j3dEngine

Alvarez, Pablo Nicolás

Rodriguez, Ignacio Gabriel

2007. Universidad Austral

Director: Gonzales Clua, Esteban Walter

Resumen

La industria de la computación gráfica 3D ha crecido exponencialmente en los últimos años, tanto que los números que se manejan se asemejan a los de la industria cinematográfica. El mercado target principal de este desarrollo es el mismo que en el cine, el mercado del entretenimiento.

Este crecimiento va acompañado de una competencia feroz entre empresas de hardware y de software. En el campo del hardware, sólo unas pocas empresas dominan el mercado de las placas gráficas aceleradoras 3D. En el campo del software, las más grandes empresas de juegos han consumido a las más pequeñas y compiten en productos de consumo final, los juegos, y productos que otras empresas de desarrollo compran como herramienta, los engines 3D.

Este documento expone los fundamentos de la programación gráfica 3D y explica el diseño y validación de una arquitectura orientada a objetos para un engine 3D, j3dEngine.

Universidad Austral		Proyecto de Trabajo de Grado	
Proyecto	j3dEngine	**Alumnos**	Alvarez, Pablo N.
			Rodriguez, Ignacio G.
Fecha	29/03/2007	**Director de Trabajo de Grado**	Gonzales Clua, Esteban Walter

Índice

Capítulo 1 – Introducción

1.1 Desarrollo y Estado Actual de la Industria de Video Games

Historia

La industria de los videojuegos involucra un sector de desarrollo y un sector de comercialización. Esta industria tiene sus origines en el año 1958 de la mano de William Higinbotham, un científico estadounidense, inventor del primer video juego conocido: "Tennis for two". Pero para ser exactos, la verdadera industria dio comienzo en 1971 con "Pong" un video juego inspirado en la creación de Higinbotham, pero con fines ya comerciales.

Figura 1.1.1 Pong

Mercado

El mercado de los videojuegos involucra una gran diversidad de disciplinas, desde programadores y matemáticos hasta artistas gráficos y actores de cine y músicos. Las ganancias de esta industria son comparables a la de la industria del cine. En el año 2005 se reportó una ganancia de U$S7000 millones solo en Estados Unidos según la ESA[1]. Otro mercado altamente importante es el mercado japonés, en el cual predomina el uso de consolas de juegos sobre el de las computadoras.

[1] Entertainment Software Association (http://www.theesa.com)

Figura 1.1.2 Ventas de videojuegos en EEUU en dólares

Algunos datos interesantes de la industria en EEUU:

1. El Mercado de los videojuegos en EEUU creció un 6% en el 2006, unos US$7400 millones. Casi triplicando las ventas desde 1996

2. 69% de los estadounidenses jefes de hogar juegan videojuegos.

3. La edad promedio del jugador es de 33 años y ha estado jugando videojuegos por 12 años.

4. La edad promedio del comprador más frecuente de videojuegos es de 40 años. En el 2006 un 93% de los compradores de juegos de computadora y un 83% de los compradores de juegos de consola eran mayores de 18 años.

5. 85% de los todos los juegos vendidos en el 2005 estaban clasificados "E" para todos, "T" para adolescentes, o "E10+" para mayores de 10 años.[2]

6. 87% de los compradores menores a 18 años reportan que tenían permiso de sus padres para alquilar o comprar los juegos. 89% dice que sus padres están presentes al momento de la compra

7. 35% de los padres en EEUU dice que juega videojuegos. 80% de ellos dice que juega con sus hijos. 66% siente que jugar videojuegos ha unido más a sus familias.

8. 38% de los jugadores son mujeres. La proporción de jugadoras mayores a 18 años (30%) es mayor a la de los hombres de 17años o menores (23%).

[2] Para mayor información consultar: ESRB: www.esrb.org/

9. En el 2005, un 25% de los americanos mayores a 50 años jugó videojuegos. Un notable incremento del 9% que era en 1999.

10. 44% de los jugadores dice que juegan juegos online 1 o más horas por semana. 32% de los jefes de hogar juega en dispositivos inalámbricos, tales como celulares o PDAs, en contraste a un 20% en el 2002.

Distribución de edad y género de los jugadores

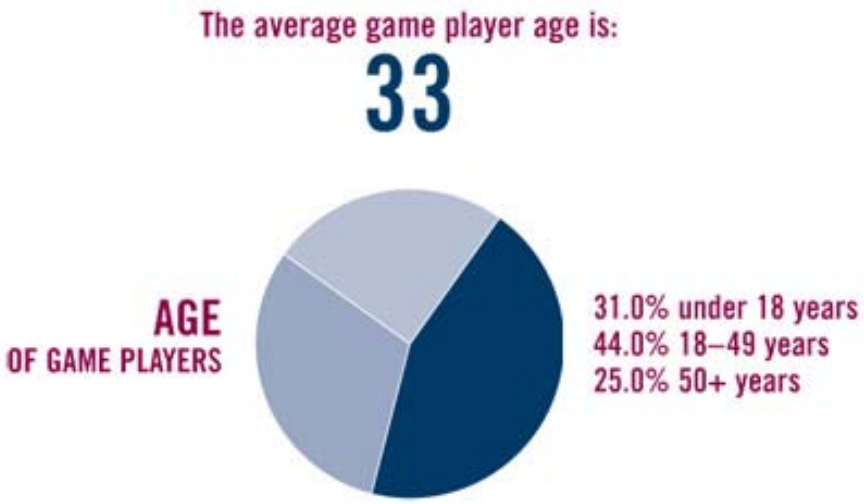

Figura 1.1.3 Edad promedio del jugador

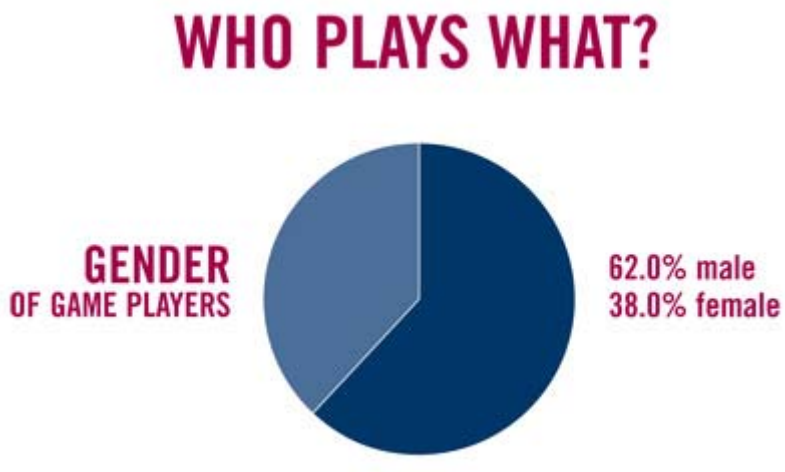

Figura 1.1.4 Género del los jugadores

Ver apéndice A para más detalles.

Economía

En los comienzos los costos de desarrollo eran mínimos y el margen de ganancia de todos los videojuegos era alto. Los juegos podían ser desarrollados por un solo programador o un pequeño equipo de desarrolladores y artistas. Incluso los publicadores eran más generosos con los beneficios que otorgaban, como por ejemplo un canon por cada copia vendida. Muchas de las empresas grandes de hoy en día, empezaron en este clima. Por lo general hoy en día los desarrolladores reciben un 20% por cada venta y el resto queda en manos del publicador.

Pero a medida que la capacidad de procesamientos y gráfica aumentaba, también lo hacían los equipos de desarrollo. Ahora los presupuestos podían alcanzar millones de dólares, aun cuando se utilizara software pre hecho, como motores gráficos, etc. Los tiempos de desarrollo también pasaron de pocos meses a varios o incluso años.

Muchas empresas optaron por la distribución online de juegos, para abaratar costos. Y muchas optaron por el desarrollo de juegos online masivos (MMOG), tales como juegos de rol online (MMORPG). Este modelo de juegos requiere por lo general la compra inicial del juego, ya sea en mano o por algún sistema online y luego el pago mensual del servicio.

Esta industria, al igual que la del cine y música, es blanco constante de la piratería, más en algunos sectores que otros. Este problema impide que empresas pequeñas que ya de por sí tienen un pequeño margen de ganancias, puedan sobrevivir en el medio.

Nuevos mercados

En los últimos años han empezado a aparecer nuevas áreas para las cuales desarrollar videojuegos. La más notoria de ellas es la de los juegos para teléfonos celulares. Esta área viene creciendo a ritmos agigantados durante los últimos años, dado que la tecnología que incorporan los celulares es cada vez más poderosa, permitiendo juegos de muy alto nivel. Lo más llamativo de esta industria es que los costos de desarrollo son muy bajos, convirtiéndola en un excelente punto de entrada para empresas nuevas que no pueden competir en el mercado principal.

Otro mercado que apenas está empezando es el originado por la integración de las computadoras y consolas con la televisión.

Salarios

Los valores corresponden a salarios promedio anuales entre todos los niveles de experiencia en el área.

Información completa y gráficos en el Apéndice A

- Programación: U$S 82,107
- Arte y animación: U$S 65,986
- Diseño: U$S 63,986
- Aseguramiento de la calidad: U$S 37,210

1.2 Introducción a los Engines 3D

Un engine 3D es el principal componente de software de un videojuego o aplicación con gráficos en tiempo real. Provee las tecnologías de bajo nivel, simplifica el desarrollo y a veces permite a la aplicación correr en diversas plataformas, como consolas y diferentes sistemas operativos de computadoras de escritorio.

Los engines proveen un set de herramientas de desarrollo para permitir un desarrollo más simple y rápido de aplicaciones 3D. Dada esta función que cumplen se los suele considerar "middleware"[3], ya que provee una cierta funcionalidad "out of the box" para simplificar el desarrollo, reducir los tiempos y el costo.

Como la mayoría de las soluciones "middleware", los engines generalmente son independientes de plataforma, de manera de poder corren tanto en consolas como diversos sistemas operativos. Los engines diseñados con orientación a objetos generalmente permiten intercambiar la implementación de diversos de sus componentes de middleware, tales como el motor de física, renderer, etc.

Los usos de los engines 3D van desde video juegos en tiempo real, aplicaciones para simulacros de accidentes y aplicaciones que no son en tiempo real, pero sí más exactas y reales, tales como simulaciones de modelos y experimentos.

[3] Software intermedio entre las aplicaciones de bajo y alto nivel. Cuya función es reducir costos y tiempos. Y a su vez independizar a la implementación de alto nivel del hardware o implementación de bajo nivel

Los componentes principales de un engine 3D son:

- Renderer
- Engine de física o detección de colisiones
- Sonido
- Scripting
- Animación
- Inteligencia artificial
- Redes
- Manejo de memoria
- Scenegraph

Algunos de estos componentes serán tratados en capítulos posteriores.

Algunos ejemplos de engines 3D son: Ogre, Torque, Unreal Engine, CryEngine, RealmForge, Source Engine. Los engines gráficos más modernos incluyen un un scenegraph, que es una representación en objetos del mundo y sus objetos.

Abstracción del hardware

Por lo general los engines o mejor dicho su sistema de rendering son construidos utilizando APIs[4] gráficas como Direct3D u OpenGL que proveen una abstracción de software de la GPU[5]. Utilizan también otras librerías de software que proveen abstracción de otros dispositivos de hardware como sonido, dispositivos de entrada, red, etc. Entre estas podemos encontrar a DirectX (Direct3D, DirectSound, DirectInput, etc), OpenAL (audio), etc.

[4] API: Application Programmer Interface

[5] GPU: Graphical Processor Unit: Es el procesador de la placa de video. Similar al CPU pero orientado puramente al rendering.

Engines especializados

Existen también engines que fueron diseñados especialmente para ciertos tipos de juegos. Hay engines especiales para First Person Shooters, engines para MMORPGs[6] o incluso engines orientados a juegos online masivos en general.

Costos y utilización

En el mercado existen tanto engines comerciales como gratuitos y open-source. Existen también muchos engines que solo son usados para los juegos que la empresa creadora realiza y no está a la venta a otros desarrolladores.

La licencia individual de un engine comercial puede salir desde US$10.000 a U$S 3.750.000 (Warcraft 3 Engine)

Top 10 Engines Comerciales	Top 10 Engines Open-Source
1. Torque Game Engine	1. OGRE
2. TV3D SDK 6	2. Crystal Space
3. 3DGameStudio	3. Irrlicht
4. C4 Engine	4. jME
5. Unity	5. Panda3D
6. Cipher	6. Reality Factory
7. 3Impact	7. The Nebula Device 2
8. Beyond Virtual	8. RealmForge GDK
9. Deep Creator	9. OpenSceneGraph
10. DarkBASIC Pro	10. Axiom

Tabla 1.1.1 Top 10 de engines más usados

[6] MMORPG: Massive Multiplayer Online Role Playing Game

Ejemplos

Figura 1.2.1 CryEngine 2: Izquierda, Foto real. Derecha: Imagen renderizada

Figura 1.2.2 Final Fantasy XIII (PlayStation 3)

Capítulo 2 – Fundamentos de 3D Game Programming

2.1 Introducción

En el mundo real, percibimos tres dimensiones de los objetos que nos rodean. En la pantalla de una computadora, sólo pueden proyectarse elementos en dos dimensiones. Es por ello que la visualización 3D consiste en una ilusión lograda a partir de complejos cálculos matemáticos que proyectan objetos en tres dimensiones sobre una pantalla, en la cual sólo pueden percibirse dos dimensiones, ancho y altura.

La profundidad no existe en la pantalla, es la dimensión que debe simularse al graficar objetos 3D. Para simular esta tercera dimensión, se agregan efectos visuales y se proyectan los objetos 3D sobre 2D a través de un proceso conocido como *rendering*.

Un *modelo 3D* es una abstracción matemática utilizada para representar la información de un objeto virtual. En el proceso de rendering, el modelo matemático se procesa hasta obtener una imagen 2D que se muestra en la pantalla.

Hay dos tipos bastante diferente de rendering 3D. El rendering lento (pre-rendering) utilizado para películas e imágenes de alta calidad, y el *real-time rendering* utilizado para simulaciones en tiempo real y juegos. El real-time rendering debe ser un proceso veloz para dar la ilusión de realismo, y por lo tanto se realizan muchas aproximaciones de los modelos para simplificar los cálculos matemáticos.

Sistemas de Coordenadas

Las medidas dimensionales de un objeto se especifican en coordenadas. De esta forma, cada vértice de un objeto está representado por un conjunto de coordenadas.

Para poder representar los atributos dimensionales de un vértice, es necesario definir primero un *sistema de coordenadas*. Un sistema de coordenadas es un sistema para asignar una n-upla de valores escalares a cada punto de un espacio n-dimensional. En este contexto sólo trabajamos con objetos de tres dimensiones y por lo tanto el sistema de coordenadas debe permitir un mapeo de triplas a vértices del espacio tridimensional.

Para especificar el sistema de coordenadas debe determinarse qué dimensión representa cada variable y cuál es el punto de referencia o punto cero para el sistema.

En el contexto de objetos 3D, podemos definir un sistema de coordenadas XYZ en el cual la variable X representa el ancho, Y la altura y Z la profundidad. Dado un objeto aislado, el centro del mismo (centro geométrico) será el punto cero del sistema de coordenadas. Esto puede visualizarse en la **Figura 2.1.1**. Se ha definido un sistema para representar las coordenadas del objeto, lo cual define un *espacio de objeto* (en el cual está aislado).

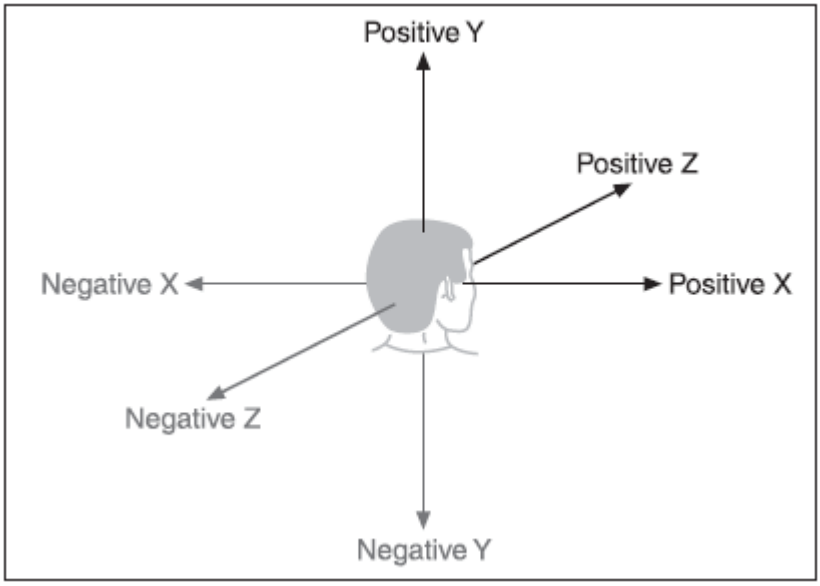

Figura 2.1.1 – Sistema de Coordenadas XYZ [3DGP]

Un sistema de coordenadas 3D puede ser "de mano derecha" o "de mano izquierda". Esto se refiere a la relación de sentidos entre los ejes. Además debe determinarse cuál eje será el vertical.

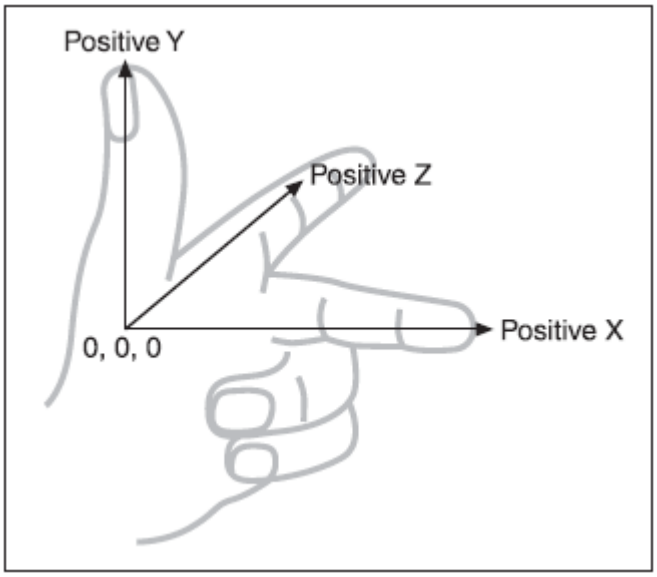

Figura 2.1.2 – Sistema de Coordenadas de mano izquierda [3DGP]

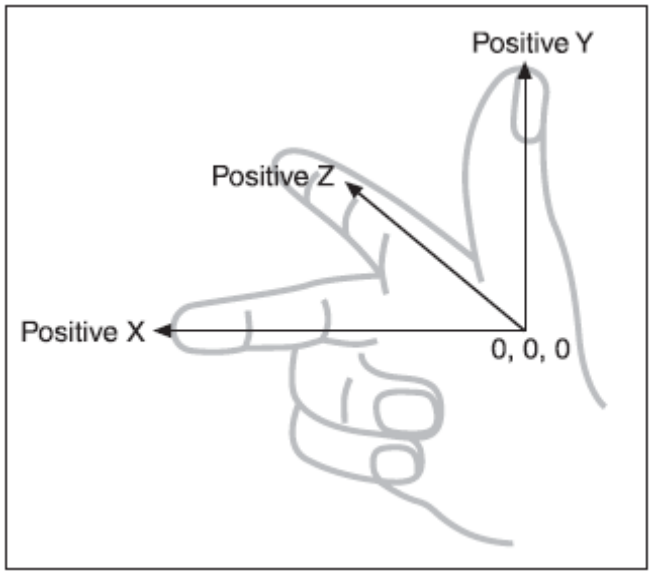

Figura 2.1.3 - Sistema de Coordenadas de mano derecha [3DGP]

Cuando disponemos de varios objetos que conviven en un mismo espacio, tenemos un *mundo* virtual. Al definir un sistema de coordenadas y un centro dentro de este mundo, hemos definido un *espacio de mundo*.

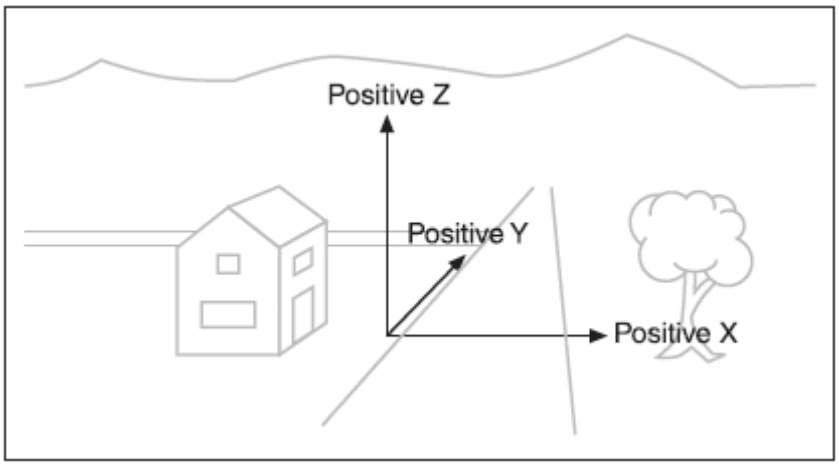

Figura 2.1.4 – Espacio de mundo con sistema de coordenadas de mano derecha y Z vertical. [3DGP]

Figuras 3D

Los objetos 3D pueden modelarse mediante *vértices*. Un conjunto de vértices unidos a través de *segmentos* en una figura cerrada constituyen un *polígono*. Un objeto 3D se modela entonces a través de polígonos.

Los complejos cálculos matemáticos realizados durante la manipulación de modelos 3D se efectúan sobre los polígonos que los representan. Como el polígono más simple es el triángulo, una simplificación muy utilizada de los modelos consiste en representar cualquier polígono mediante una descomposición en triángulos. Esto es conocido como *malla (mesh)* de triángulos.

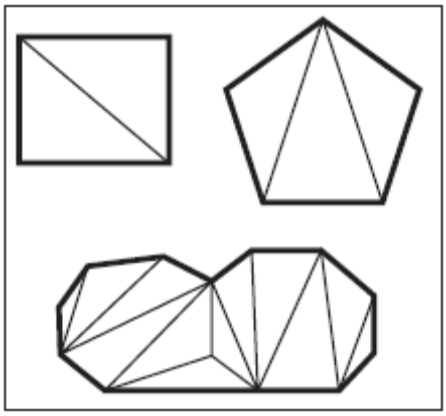

Figura 2.1.5 – Polígonos descompuestos en triángulos [3DGP]

Decimos que un objeto 3D representado por polígonos tiene una *superficie*. A su vez, una superficie tiene *caras*. En función del punto de vista del observador, habrá una cara visible y otra invisible (*backface*) y por lo tanto polígonos visibles y otros invisibles.

Vistas

Una *cámara* es una posición en el espacio del mundo virtual. Se trata de un punto de observación. Una *ventana de vista* o *vista* es una ventana en el espacio 3D del mismo tamaño que la pantalla.

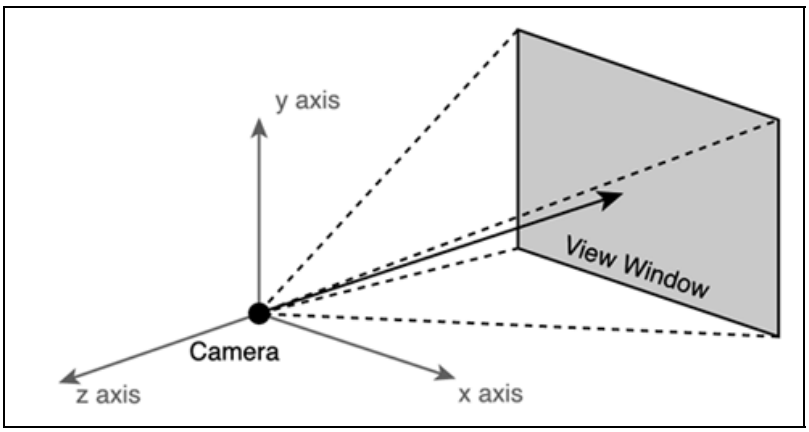

Figura 2.1.6 – Cámara en el centro de coordenadas y vista. [DGJ]

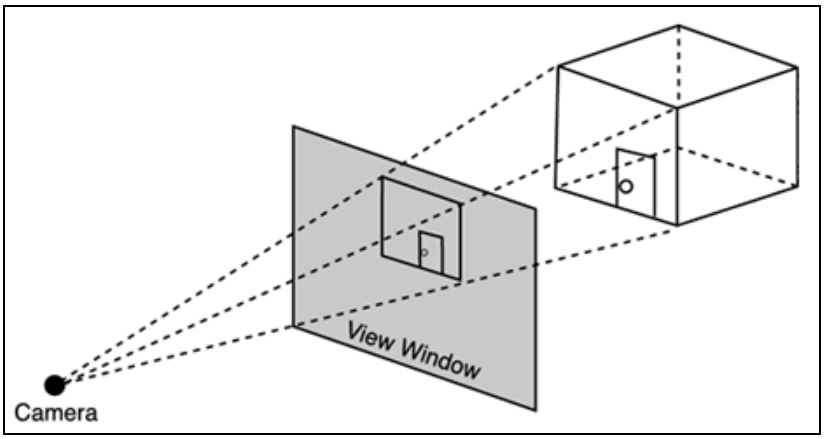

Figura 2.1.7 – Objetos 3D proyectados sobre la vista [DGJ]

Los objetos 3D del mundo virtual deben proyectarse hacia 2D sobre la vista, la cual constituye la imagen que se verá en la pantalla. Esto es parte del proceso de rendering.

Dadas una cámara (o posición en el espacio 3D) y una vista, hay un conjunto de objetos 3D del mundo que se visualizarán en la vista y otros que no. Se define entonces el concepto de *view frustum*. Se trata de una pirámide definida por la vista y la cámara, todo lo que esté encerrado por el view frustum será visible para la cámara.

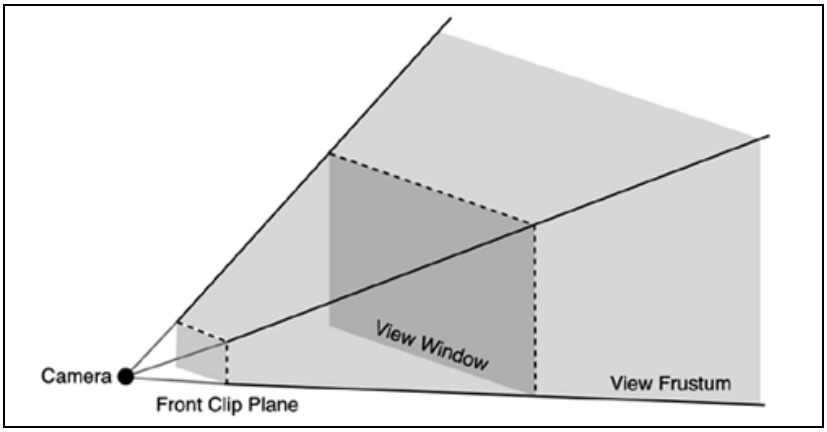

Figura 2.1.8 – View Frustum [DGJ]

El view frustum es la pirámide entre el *plano de corte frontal* y el plano de máxima distancia a la cual podemos visualizar objetos 3D. El plano de corte frontal se establece para no mostrar polígonos que estén demasiado cerca de la cámara.

2.2 Display de un Modelo 3D

Un mundo virtual es un modelo constituido por un conjunto de objetos 3D relacionados espacialmente. Cada objeto 3D tiene un sistema de coordenadas *local* a partir del cual está definida su geometría. A la vez, para ubicar a los objetos 3D en un mismo espacio, necesitan compartir un mismo sistema de coordenadas. Este es el sistema de coordenadas *global* o de mundo.

Para poder proyectar en la pantalla a un objeto 3D, éste debe pasar por tres etapas de un proceso de conversión:

1. Convertir las coordenadas locales a coordenadas en el espacio global
2. Convertir las coordenadas a coordenadas relativas a la vista
3. Convertir las coordenadas a coordenadas 2D de la pantalla

Transformaciones

La primera etapa ubica a nuestro objeto 3D en el espacio del mundo. Esta operación se realiza a través de operaciones matemáticas de *transformación*.

Hay tres tipos de transformaciones básicas para convertir las coordenadas locales a globales.

La *escala* (o *scaling*) es una operación para adaptar el tamaño del objeto 3D al adecuado para el mundo en que se encuentra.

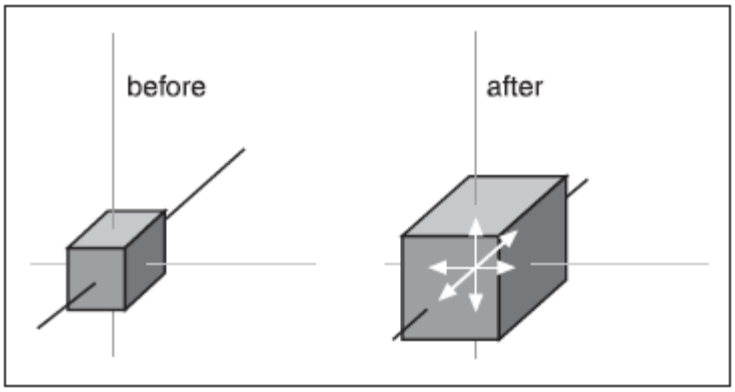

Figura 2.2.1 – Scaling. [3DGP]

Otra operación es la *rotación*, la cual permite orientar al objeto 3D en el mundo.

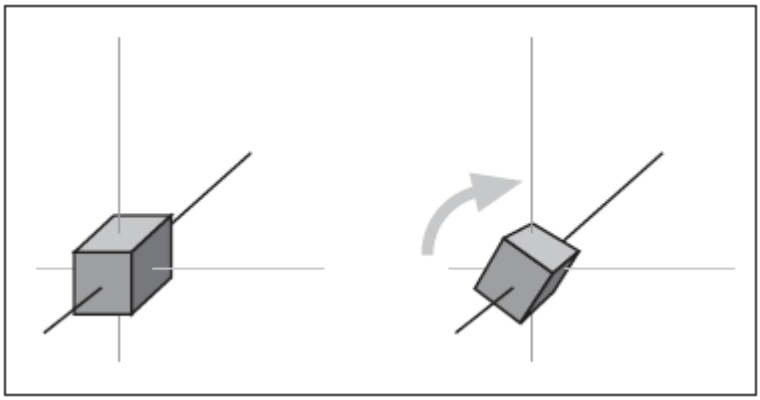

Figura 2.2.2 – Rotación. [3DGP]

Finalmente, la otra operación posible es la *traslación*. A través de esta se ubica al objeto 3D relativo al sistema de coordenadas del mundo.

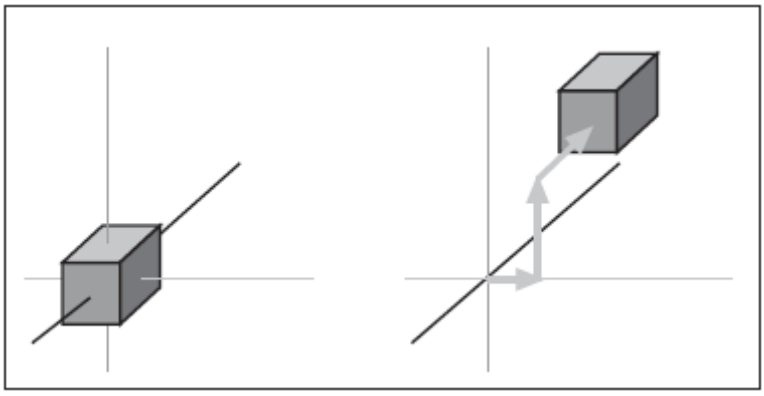

Figura 2.2.3 – Traslación. [3DGP]

Estas operaciones se realizan en conjunto para conformar una sola transformación compuesta, que termina convirtiendo al objeto 3D al sistema de coordenadas global.

Pipeline Gráfico

Para proyectar objetos 3D en la pantalla se realiza un proceso iterativo que incluye el procesamiento lógico, propio de la aplicación, y el proceso de rendering que es más específico de la proyección de gráficos en la pantalla. Es conocido como *Pipeline Gráfico*.

El pipeline, entonces, tiene dos grandes fases:

1. Lógica de Aplicación
2. Rendering

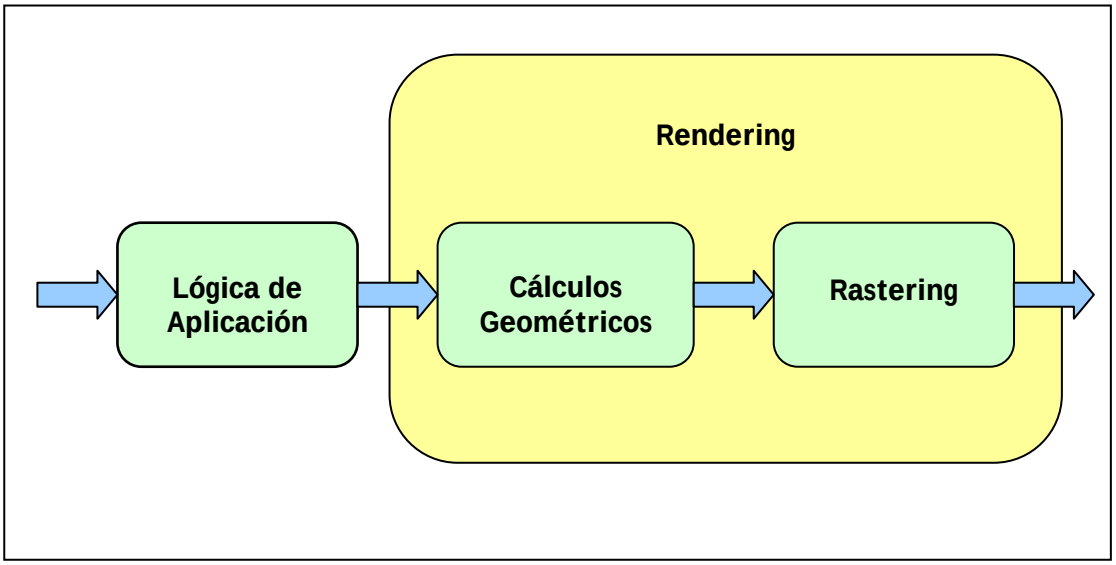

Figura 2.2.4 – Pipeline Gráfico

En la primera fase se procesa la lógica propia de la aplicación. Por ejemplo: actualizar el estado del teclado y mouse, calcular posiciones de los objetos, calcular las colisiones entre ellos o aplicar algoritmos de inteligencia artificial.

La segunda fase es la transformación del modelo matemático 3D a los gráficos 2D en pantalla.

Rendering Pipeline

Puede considerarse a la fase de rendering como un pipeline dentro del pipeline gráfico. Hoy en día el rendering ya no se procesa en el CPU, este procesamiento es realizado por el *GPU*, que es un tipo de microprocesador desarrollado específicamente para realizar estos cálculos con gran velocidad. Es una parte esencial de las placas gráficas aceleradoras actuales.

Este pipeline consiste en dos etapas.

1. Cálculos geométricos (o etapa de Geometría): transformación y proyección de los objetos 3D modelados por vértices y polígonos, a píxeles en 2D.

2. *Rastering*: procesamiento de los píxeles para obtener la imagen final que será mostrada en pantalla.

El procesamiento de la geometría incluye las siguientes operaciones:

1. Transformaciones geométricas: Como se explicó anteriormente, convierten al objeto 3D hacia el sistema de coordenadas del mundo virtual.

2. Selección de objetos visibles: dada una posición de la cámara se define un view frustum. Esto permite seleccionar qué objetos serán renderizados y cuáles no. Es un paso necesario para la renderización en tiempo real.

3. *Iluminación* de los vértices (*vertex shading*): en el rendering uno de los cálculos más importantes para lograr realismo es el de iluminación. Hay diferentes técnicas para lograrlo. En la etapa de geometría se efectúan los cálculos necesarios para obtener la intensidad y color de la luz en cada vértice de los polígonos que se mostrarán en pantalla.

4. *Proyección*: este cálculo es el que realiza una proyección matemática de los polígonos 3D sobre el sistema de coordenadas 2D de la ventana de vista[7].

5. *Backface culling*: los polígonos cuya cara apuntan hacia el lado contrario a la cámara, puede ser eliminados del proceso de renderización. Se trata de una optimización opcional, que logra importantes mejoras de performance.

6. Recorte o *clipping*: si una vez proyectados, los vértices quedan fuera de la vista, entonces esos vértices no deben seguir su camino en el pipeline hacia el rastering, pues introducen costo de procesamiento innecesario ya que no se llegarían a ver en pantalla. El clipping es la operación que consiste en eliminar esos vértices del cálculo.

[7] También conocida como *Viewport*.

Una vez que se obtienen los polígonos a renderizar en 2D, se pasa a la etapa de rastering, en la cual se realizan cálculos sobre los vértices proyectados y se entrega la imagen final a la salida de la placa gráfica, para que sea proyectada por el monitor.

En contexto de la pantalla (2D), debemos dejar de hablar de vértices para pasar a hablar de píxeles. Luego de la primera etapa, cada elemento del mundo virtual 3D que será visible en la pantalla ha sido mapeado a uno más *píxeles*. La cantidad de píxeles que constituyen la pantalla 2D dependerá de la resolución que se esté utilizando para formar la imagen final.

El rastering a su vez es una serie de operaciones, las más importantes son:

1. Ocultamiento por buffer de profundidad *(depth buffer occlussion)*: cada píxel contiene, entre otros datos, información sobre la profundidad en relación al polígono desde el cual fue obtenido. En el buffer se guardan los píxeles que potencialmente formarán la imagen. Cuando un nuevo píxel tiene las mismas coordenadas 2D en pantalla que uno previamente calculado, se descarta el de mayor profundidad ya que los objetos más cercanos son los que deben ser renderizados. El depth buffer se conoce también como *z-buffer*.

2. *Texturización* (*texturing*): lo normal es que cada mesh tenga una textura mapeada. Eso significa que se mapean las coordenadas de una imagen a las coordenadas de los polígonos. En esta operación los colores de una imagen guardada en memoria se mapean a los colores de los píxeles que formarán la imagen 2D final.

3. Iluminación por píxel *(pixel shading)*: En la iluminación por píxel se asigna el color correspondiente a cada píxel que formará parte de la imagen.

4. *Frame buffer*: es un buffer que se guarda en una parte de la memoria RAM de la placa gráfica. En este buffer se coloca el color para cada píxel que será mostrado en pantalla. El monitor genera una imagen en la pantalla a partir de los datos que lee de este buffer. Una técnica muy utilizada para lograr que los *frames* (cuadros) se rendericen completamente antes de ser mostrados, es *double buffering*. Esta técnica consiste en almacenar las imágenes en dos frame buffers e ir alternando la salida de la placa gráfica entre ambos.

Al final del rendering pipeline, la placa gráfica entrega al monitor la información almacenada en el frame buffer, logrando mostrar en el mismo una imagen renderizada.

Frames per second (FPS) y Game Time

Cada imagen mostrada en pantalla se conoce como *frame (cuadro)*. Cada pasada del rendering gráfico termina en un frame.

El pipeline gráfico puede tener cuellos de botella (bottlenecks) que reduzcan el throughput del mismo. En el contexto de real time rendering, al throughput del pipeline gráfico se lo conoce como frames per second (cuadros por segundo).

Los FPS son una medida muy utilizada para determinar la performance de una aplicación 3D en tiempo de ejecución. Los FPS pueden verse afectados por cualquier elemento del pipeline. Puede ser que la lógica del juego sea ineficiente, o que el CPU no pueda procesarla. Puede ser que la placa gráfica no pueda realizar a tiempo todos los cálculos matemáticos para renderizar. Si es así, los FPS se reducirán y la apreciación del juego o aplicación 3D se verá muy afectada.

Un concepto importante es el de *Game Time (tiempo de juego)*. Es el valor del tiempo que transcurre en el mundo virtual. Este valor no puede verse afectado por el rendimiento del pipeline gráfico, los FPS y el game time deben ser independientes.

Esto puede verse fácilmente si pensamos en un juego multiplayer. Varios jugadores se conectan a través de una red, y comparten un mismo mundo virtual durante un juego. Si uno de ellos tiene una computadora más lenta que los demás, el pipeline gráfico tendrá un throughput menor al de las demás computadoras. Si esto afectase al juego, los demás jugadores observarían un mundo virtual que pierde realismo, por ejemplo, podrían empezar a ver las cosas en cámara lenta.

Por esta independencia necesaria, cuando los FPS disminuyen, el usuario pierde escenas o cuadros de lo que sucede en el mundo virtual. Si un automóvil del mundo virtual se está moviendo a una velocidad determinada y los FPS reducen sensiblemente, el automóvil no reducirá su velocidad, sino que el usuario no podrá ver el movimiento completo, o sea, se saltará unos cuadros de la animación.

En determinadas condiciones pueden alcanzarse valores altos de FPS, como 100 – 120 o más. Pero un muy buen valor ronda los 30 FPS. Puede compararse esto a una película del cine o a la televisión, en los cuales para alcanzar la ilusión de movimiento se deben logran 24 cuadros por segundo. Si el valor fuese menor, veríamos un parpadeo de la imagen. De hecho esto sucedía en las primeras películas de cine y se corrigió proyectando cada imagen de la película dos veces seguidas.

2.3 Agregando realismo – Iluminación y Texturas

Iluminación

Uno de los componentes esenciales para lograr la ilusión de visualizar un objeto 3D es la iluminación. La luz reflejando en un objeto y las sombras son lo que logran la sensación de profundidad.

Las superficies de los objetos 3D tienen distintos *materiales*. En función de su material y la luz que reciben, las superficies adquieren un determinado color.

En la realidad, la luz emitida por una fuente puntual se refleja en todos los objetos con los que colisiona y se va reflejando y refractando hasta que se atenúa completamente. Este es un cálculo muy complejo de realizar, y por lo tanto se han desarrollado modelos de iluminación que dan la ilusión de una iluminación real. Hay distintos tipos de iluminación y los más importantes se explican a continuación. Cada tipo se calcula por

separado, cada una es una componente de la iluminación que recibe cada punto de la superficie de los objetos del mundo virtual.

Es importante considerar que las luces pueden tener color, intensidad y atenuación.

Luz Ambiental

En una habitación cerrada, la luz de una lámpara se refleja en todas las paredes, en el piso y en el techo. Esto se simula agregando una componente de luz que afecta a todos los objetos del mundo virtual por igual. Matemáticamente, se trata de un escalar.

Los materiales de los objetos 3D poseen a la vez una componente de luz de emisión (*emissive lighting*). Esta componente simula la iluminación ambiental en cada objeto en particular.

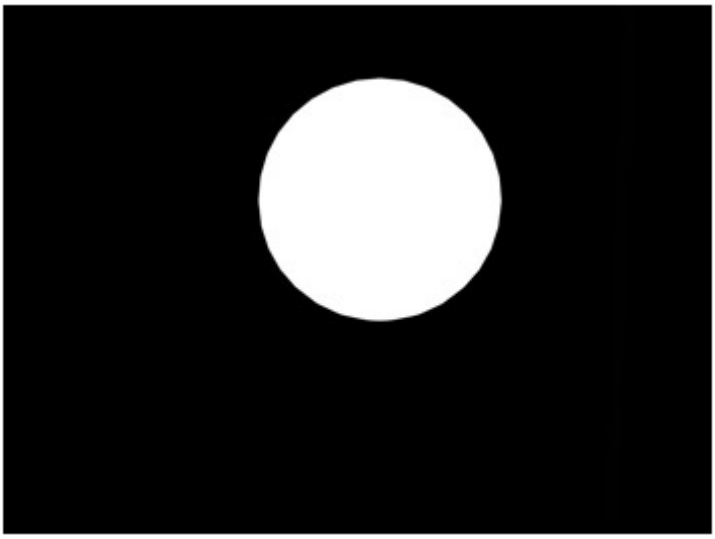

Figura 2.3.1 – Esfera con iluminación ambiental blanca [RTR]

En la **figura 2.3.1** puede visualizarse una esfera que sufre una iluminación ambiental blanca o emite luz blanca, sin verse afectada por otra fuente de luz.

Como la luz ambiental no considera la dirección de la luz, en general no agrega realismo a la imagen.

Iluminación Difusa (Diffuse lighting)

La iluminación difusa es la componente de luz que recibe un objeto a partir de una fuente puntual de luz proveniente de una determinada dirección. Según la ley de Lambert, la intensidad de la luz en un punto depende del ángulo formado por la dirección del rayo de luz y la normal a la superficie del objeto iluminado.

Según esta definición, cada punto de una superficie iluminada recibe una intensidad diferente de luz y los puntos no iluminados están en oscuridad.

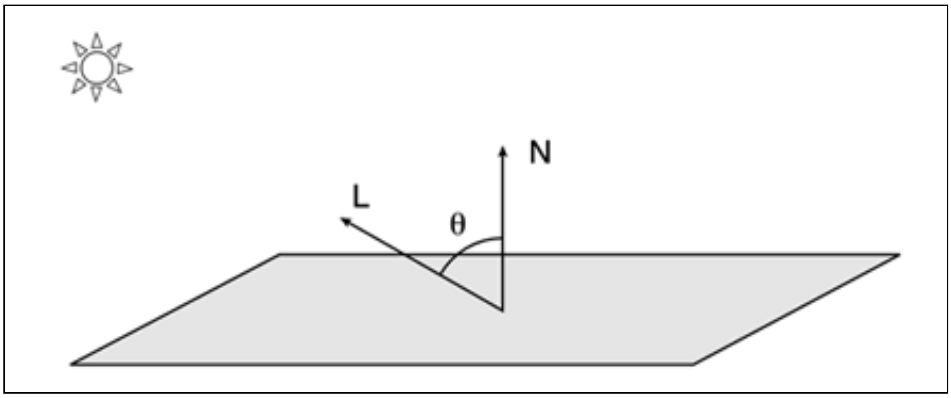

Figura 2.3.2 – Iluminación difusa [DGJ]

El cálculo de esta iluminación es mucho más complejo que la iluminación ambiental, pero logra mucho mayor realismo en las imágenes.

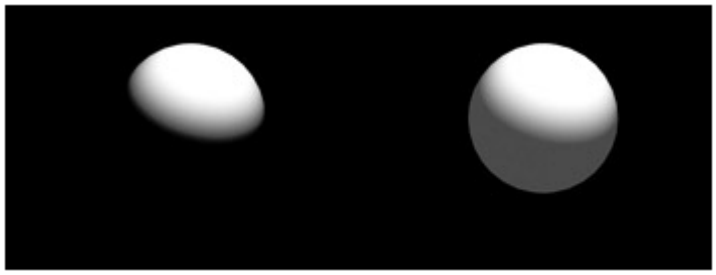

Figura 2.3.3 – a) Esfera con iluminación difusa. b) Iluminación difusa y ambiental. [RTR]

En la **figura 2.3.3** se nota el efecto. En **a)** podemos ver que una fuente de luz puntual ilumina desde arriba a la esfera. En **b)** se suma el efecto de la luz ambiental.

Este modelo ignora la posición del observador, es decir, desde cualquier punto que observemos una parte de un objeto, visualizaremos el mismo color. Es claro que esto no es así en la realidad, ya que la cantidad de luz que nos llega reflejada desde el objeto hasta nosotros depende de nuestra posición.

Iluminación Especular (Specular lighting)

La iluminación especular permite simular que una superficie brillosa refleja la luz en determinadas direcciones, y no la difunde en todas direcciones por igual (como en la iluminación difusa).

La iluminación en este caso depende de cuán brillosa sea una superficie y de la posición del observador. Por ello, este cálculo es incluso mucho más complejo que para la iluminación difusa y se utiliza mucho menos.

Fuentes de Luz

En un mundo virtual, la iluminación proviene de las fuentes de luz que se dispongan. Las fuentes de luz más comunes que se utilizan son:

- Luz ambiental: ilumina uniformemente todos los objetos. Produce la el tipo de luz ambiental explicado anteriormente.

- Luz puntual: fuente puntual de luz, la cual se caracteriza por una posición y una atenuación. Es una fuente omnidireccional.

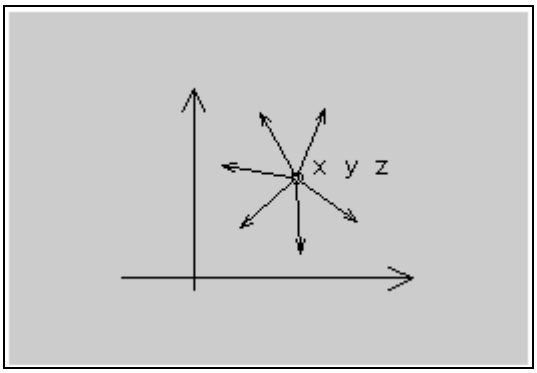

Figura 2.3.4 – Fuente de luz puntual omnidireccional [X3DN]

- Luz direccional: fuente de luz con una determinada dirección pero con origen en el infinito. Sirve por ejemplo para simular la iluminación del sol.

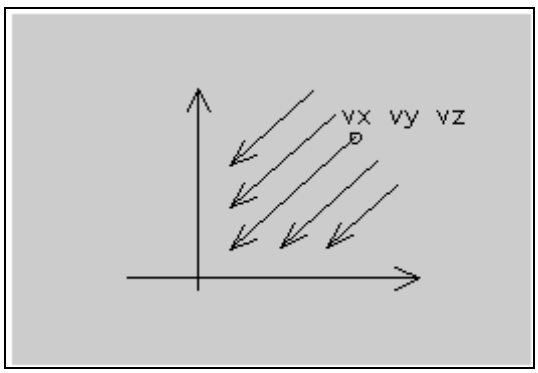

Figura 2.3.5 – Fuente de luz direccional [X3DN]

- Spot Light: fuente puntual pero con una determinada dirección y un ángulo de dispersión.

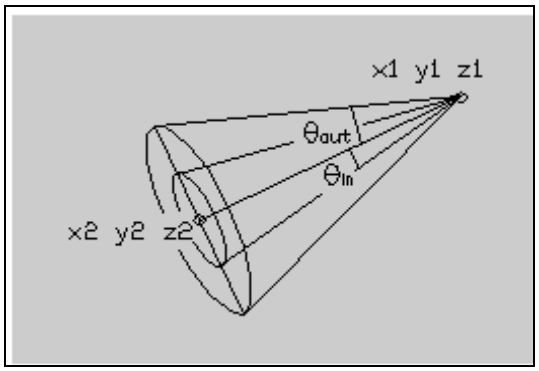

Figura 2.3.6 – Spotlight [X3DN]

Todas las fuentes de luz poseen además un atributo en común, el color.

Texturas y Texture Mapping

Las *texturas* son otro componente esencial para la visualización 3D.

Una textura es una imagen 2D en formato binario que se utiliza como material para un objeto 3D. La aplicación de texturas a las figuras 3D las asemeja más a la realidad.

Figura 2.3.7 – Estructura con aplicación de texturas. [3DGP]

Para poder aplicar una imagen 2D a un objeto 3D, debe pasarse por un proceso conocido como *texture mapping.*

Lo que se desea obtener en este proceso es un color apropiado para cada punto de un polígono, en función de su textura y de la orientación con que esta se aplica al mismo.

La orientación de una textura sobre un polígono se define a través de un punto de origen y dos vectores directores. Una vez definido esto, el proceso de texture mapping debe asignar cada elemento de la textura (*texel*) a cada punto del polígono.

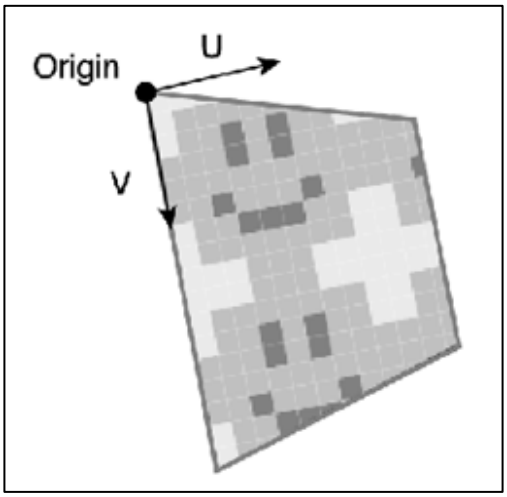

Figura 2.3.8 – Orientación de una textura sobre un polígono [DGJ]

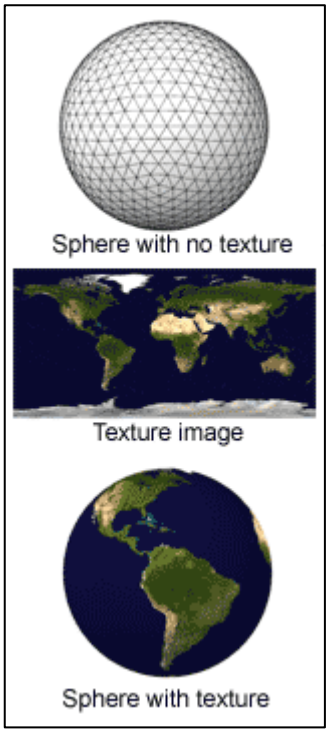

Figura 2.3.9bis – Esfera después de realizado el mapeo de textura.

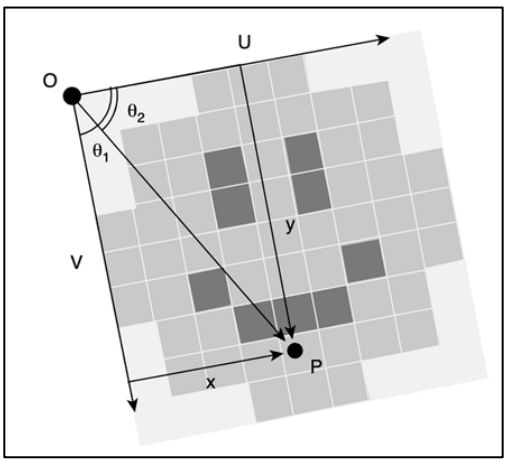

Figura 2.3.9 – Mapeo de texels a puntos de un polígono [DGJ]

Al finalizar la renderización, el color de cada punto será función del material del polígono al que pertenece y de la intensidad y color de la luz que impacta contra éste.

Skyboxes

Una técnica muy utilizada en la renderización de escenarios abiertos es el *Skybox*.

Esta técnica consiste en encerrar al escenario en un cubo y asignar una textura, cuidadosamente seleccionada, a cada cara del cubo. Si esto se hace adecuadamente, puede simularse con un bajo costo de procesamiento el cielo, el sol o un paisaje a lo lejos, creando la ilusión de horizonte.

Figura 2.3.10 – Horizonte de nubes en un escenario logrado con un Skybox. [3DGP]

2.4 Librerías Gráficas

Hoy en día gran parte de la programación en computación gráfica 3D está dedicada a utilizar los recursos que brindan las placas aceleradoras 3D.

El GPU es un procesador dedicado a cálculos matemáticos orientados a figuras 3D. Está diseñado para procesar una parte del pipeline gráfico: el rendering.

Dado que los cálculos se hacen al nivel de un hardware optimizado para realizarlos, ejecutar las mismas operaciones con una CPU, la cual es de aplicación general, requeriría de muchas más instrucciones y tomaría mucho más tiempo.

Para acceder a estos recursos de hardware, disponemos de librerías gráficas que nos permiten acceder fácilmente a las funciones que éstos proveen. El acceso a estas librerías se da en forma de API (Application Programming Interface) para permitir al desarrollador invocar llamados a la placa gráfica.

Las dos librerías gráficas más importantes son *OpenGL* y *DirectX*.

OpenGL (Open Graphics Library)

OpenGL es una especificación estándar cross-platform. Existe una implementación para los sistemas operativos más utilizados. En particular hay implementaciones para Mac OS, Windows, Linux, algunas plataformas Unix y PlayStation 3.

Fue creada por Silicon Graphics Inc. en 1992 y desde entonces ha sido ampliamente utilizado para realidad virtual, CAD, simulaciones y juegos.

Existe mucha documentación sobre el uso de OpenGL, por lo cual se ha hecho muy popular en la comunidad de desarrollo 3D.

Hay muchas librerías de utilidades que extienden la funcionalidad básica provista por OpenGL y simplifican la el desarrollo. Algunas son: GLU, GLUT, SDL, GLUI, FLTK.

Las implementaciones de OpenGL brindan una interfaz para el lenguaje C.

Hoy en día C++ es el lenguaje más utilizado para el desarrollo de juegos, y es con el que se logran mayor performance. Igualmente en la actualidad Java está comenzando a ser utilizado para el desarrollo de aplicaciones 3D y para utilizar OpenGL se hace uso de un framework de Java llamado JNI (Java Native Interface) el cual permite realizar invocaciones nativas (llamadas específicas de la plataforma).

Utilizando JNI, se han creado *ports* de OpenGL a Java para brindar un API similar al de OpenGL al desarrollador Java. Los port existentes son JOGL (Java OpenGL) y JWJGL (Lightweight Java Game Library).

DirectX

DirectX es un API creada por Microsoft para acceder a los recursos de hardware asociados con sonido y video. Una parte de DirectX, Direct3D, es específica para el uso de recursos de hardware orientados a rederivación 3D.

Inicialmente los creadores de juegos distribuían su implementación de Direct3D, pero hoy en día DirectX es parte de Microsoft Windows.

DirectX provee mayor funcionalidad que OpenGL, como el manejo de periféricos, sonido y música. Además se actualiza con mayor periodicidad. Sin embargo la desventaja principal de DirectX es que sólo funciona bajo la plataforma Windows.

Esto último influye a la hora de decidirse por una de las librerías antes de desarrollar un producto 3D, ya que al utilizar DirectX la aplicación sólo podría funcionar en Windows.

Capítulo 3 – 3D Game Engines

3.1 Funcionalidad

Introducción

Tal como fue dicho en el capítulo 1. Un engine gráfico es una aplicación middleware que provee una capa de abstracción entre las implementaciones de bajo y alto nivel. Permitiendo al juego la utilización de diferentes tecnologías de forma fácilmente intercambiable. Otras de sus funciones es brindar la compatibilidad multiplataforma. Pero sin duda una de las principales ventajas de un engine es simplificar el desarrollo de los juegos, tanto desde el punto de vista de no tener que implementar gran parte de la funcionalidad sino también del lado de brindar al diseñador y programador de juegos un conjunto de herramientas, gráficas o no, que le permitan desarrollar más rápida y fácilmente.

Componentes

Los componentes que se suelen encontrar en los engines son:

- Motor de ejecución:

  Es el componente principal y presente en todo engine. Provee toda la implementación de procesamiento haciendo uso de todas las tecnologías y módulos que engine utilice. Es el corazón de todo engine gráfico. Provee también al desarrollador las librerías e interfaces necesarias para poder extender su funcionalidad utilizando módulos o tecnologías alternativas.

  Entre los componentes del motor de ejecución se encuentran:

  - Módulo de matemática
  - Módulo de física
  - Módulo de redes
  - Módulo de audio / video
  - Módulo de inteligencia artificial
  - Módulo de animación
  - Módulo de scripting
  - Módulo de dispositivos de entrada
  - Renderer
    - Scenegraph
  - Importadores y exportadores de recursos

- Editor de niveles:

El editor de niveles es una de las herramientas gráficas más importantes de un engine. Esta brinda al desarrollador y diseñador una forma sencilla de visualizar el mundo 3D que están creando. Permitiéndoles cargar modelos, ubicarlos y definir como interactuarán, definiendo el comportamiento de los objetos dinámicos, tales como objetos que se muevan, personajes controlados por la computadora, etc.

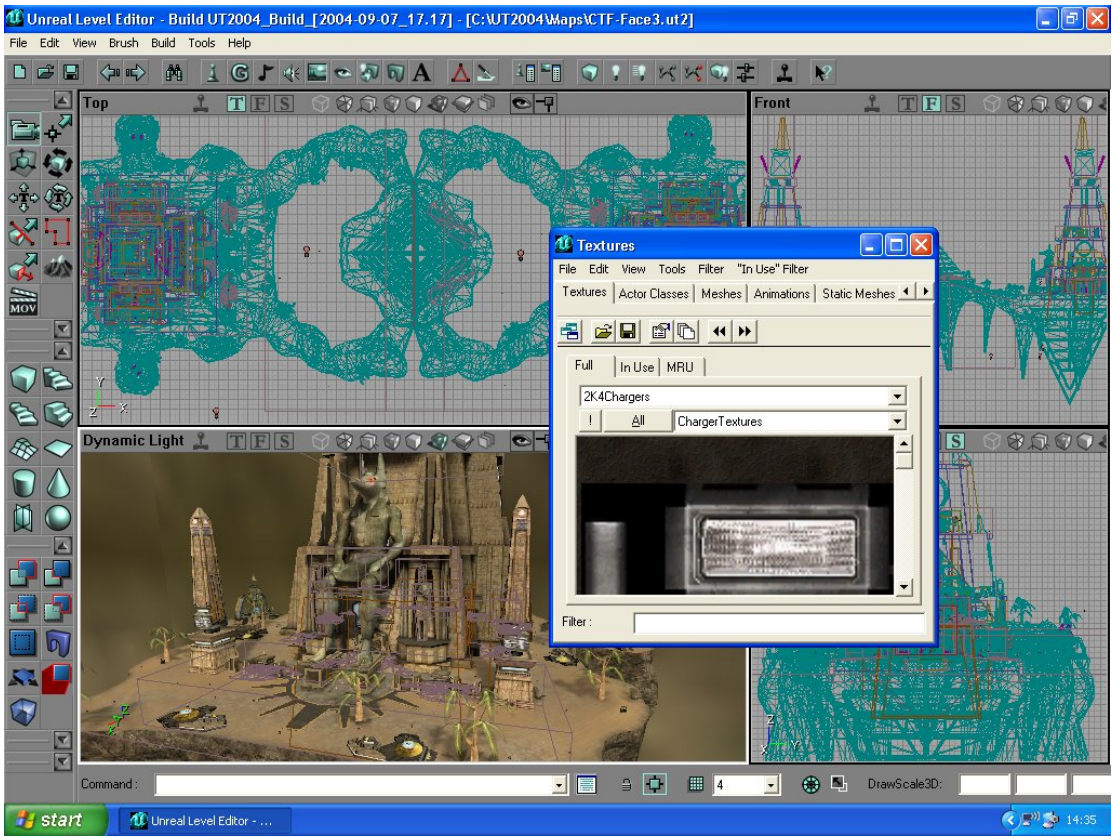

Figura 3.1.1 Editor de niveles: Unreal Level Editor.

- Editor de modelos:

Algunos engines proveen también un editor de modelos. Normalmente los modelos se crean utilizando herramientas especificas de modelado 3D tales como 3D Studio, Maya, Blender, etc. El editor de modelos de los engines permite visualizar estos modelos y a veces modificarles alguno de sus parámetros, como texturas, colores, etc. Es común que el editor de modelos este integrado dentro del editor de niveles.

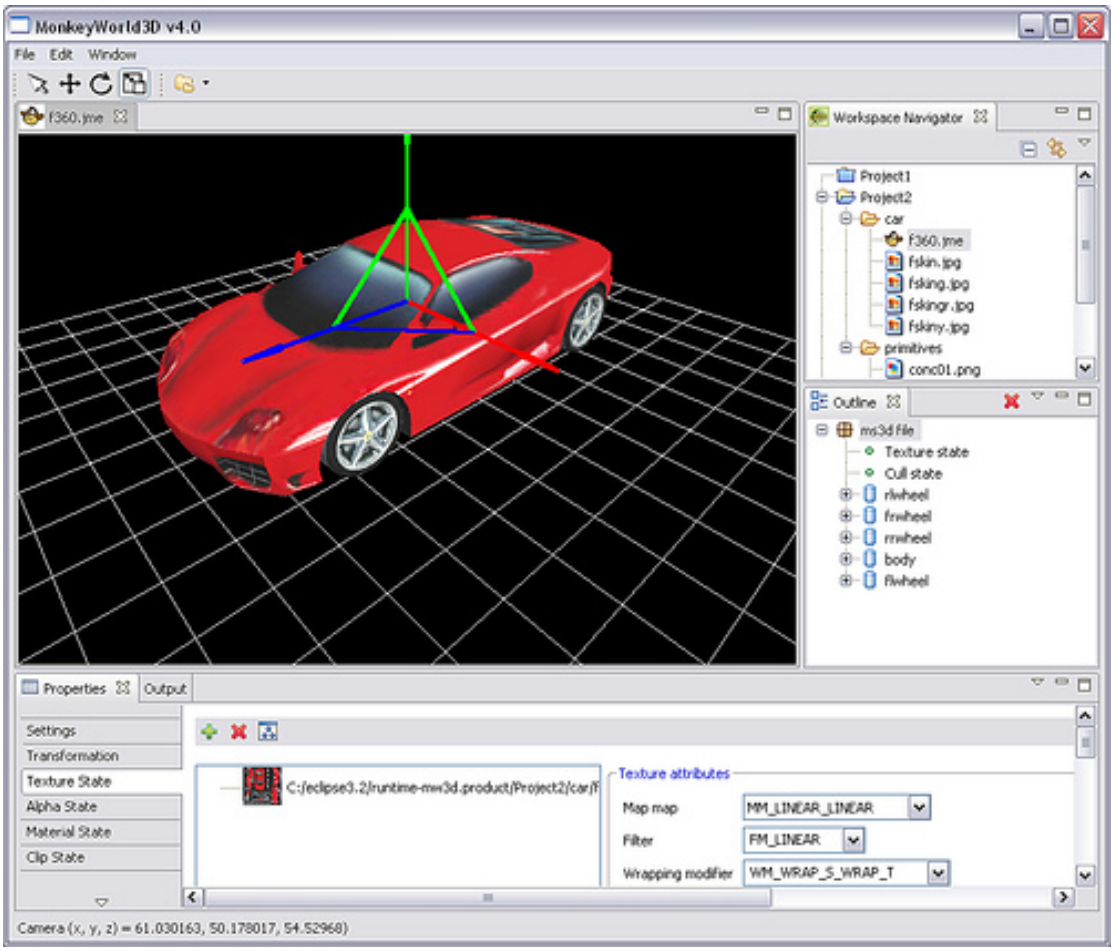

Figura 3.1.2 Editor de modelos del JMonkeyEngine. MonkeyWorld3D

- Editor de scripts:

El editor de scripts es una herramienta que permite al desarrollador y diseñador escribir de forma sencilla y asistida los scripts[8] que definirán el comportamiento que se le puede asignar a los diversos objetos dinámicos del mundo. El script se escribe utilizando un lenguaje de scripting que el engine utiliza, este puede ser propietario del engine o estándar, algunos engines permiten escribir los scripts en más de un lenguaje posible. Estos scripts pueden acceder a diversos componentes del motor de ejecución de forma simplificada, para definir condiciones o acciones. Como por ejemplo, acceder al teclado o mouse, para disparar un evento al presionarse una tecla especifica. La idea principal de los scripts es que sean simples y entendibles de manera de que un diseñador sin el conocimiento cómo funciona el engine en el fondo pueda diseñar su juego. Normalmente este editor viene integrado en el editor de niveles.

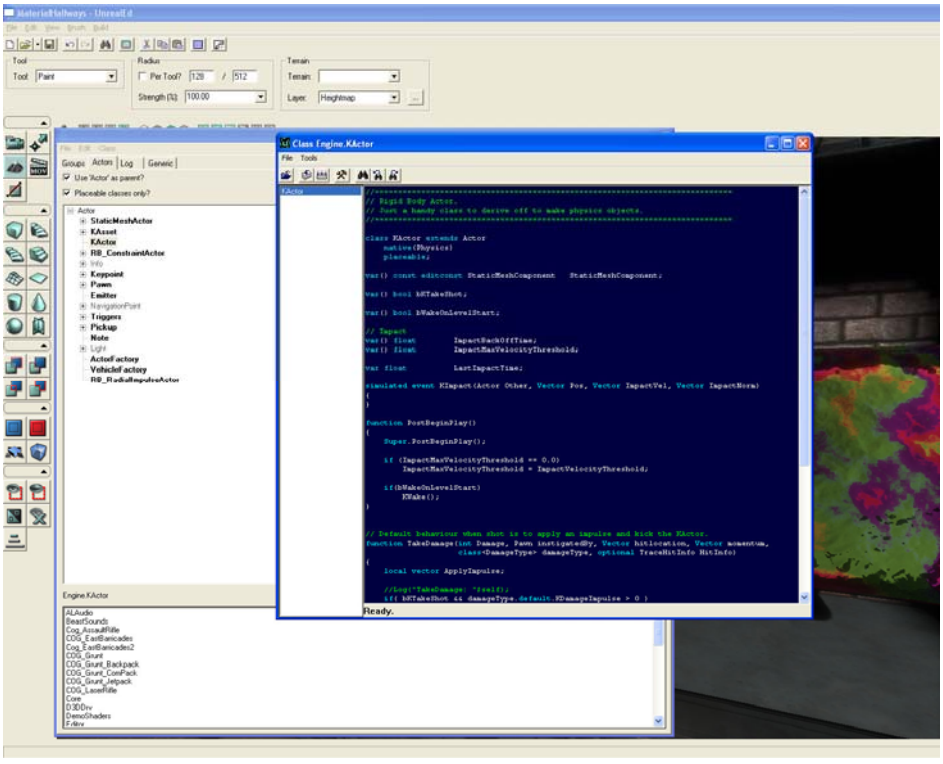

Figura 3.1.3 Editor de scripts del UnrealEngine 3. UnrealScript

[8] Script: Es un programa de computadora escrito en un cierto lenguaje de scripting. La característica principal de un script, es que su código no se compila en un binario, sino que se interpreta desde su fuente.

- Otras herramientas:

  Algunos engines proveen otras herramientas tales como:

  - Importadores/exportadores de modelos a un formato de modelo propietario del engine.
  - Editores de animaciones que permiten al diseñador animar sus modelos ya se creando animaciones utilizando este editor o utilizando animaciones pre hechas con otro software.
  - Herramientas para monitoreo de servidores, estas herramientas se dan en el caso de los engines orientados a juegos puramente multiplayer en los que debe existir un server de red. (Ej.: MMORPGs).

3.2 Arquitectura básica de un game engine

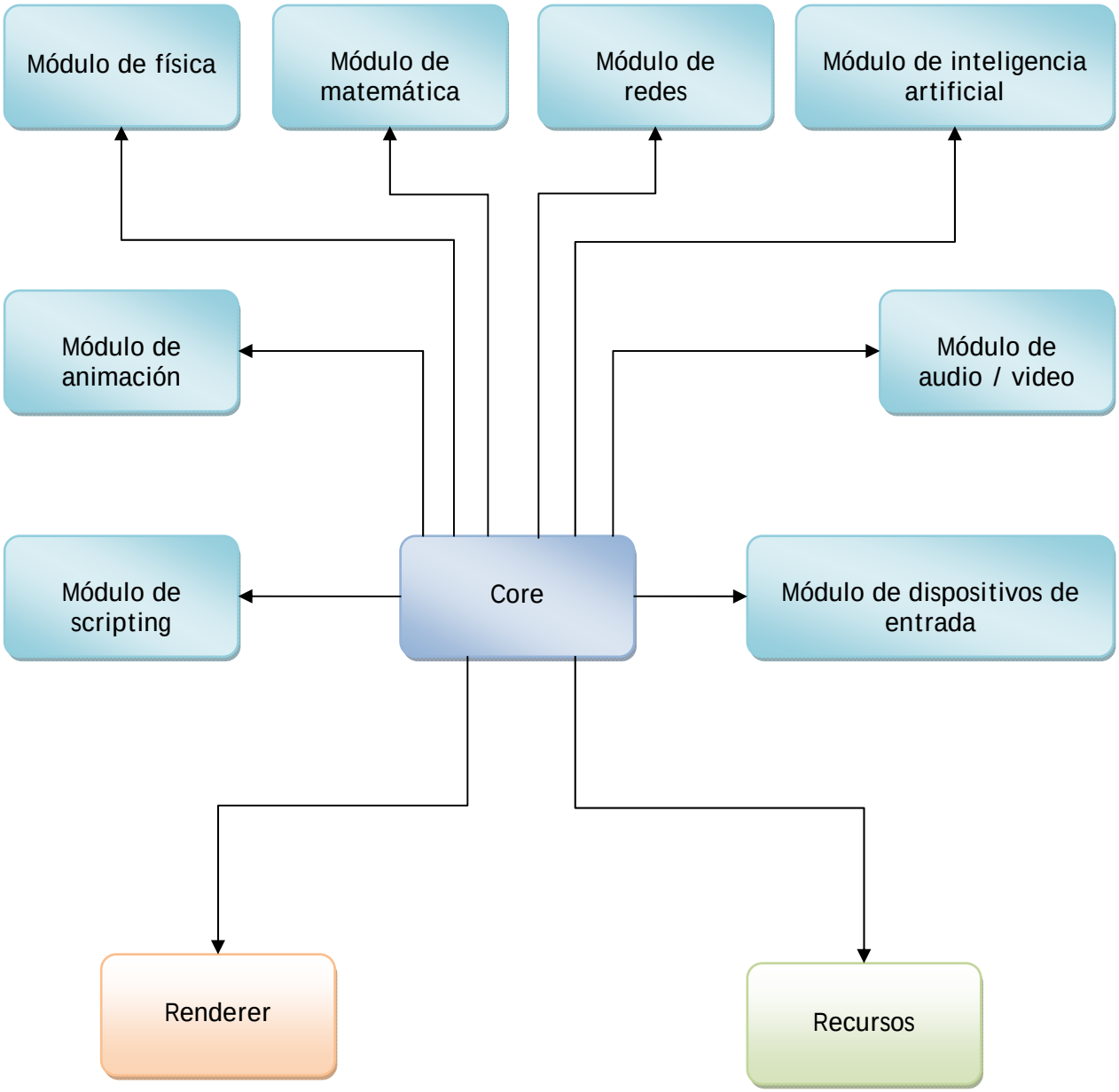

Figura 3.2.1 Arquitectura de módulos básica de un game engine

Los game engines pueden incluir en su implementación todos o ninguno de los módulos nombrados. En caso de no incluir su propia implementación. Se provee una interfaz de manera de poder utilizar cualquier módulo externo que se desee, haciéndolos intercambiables y también reduciendo la complejidad de desarrollo del engine y su costo. Dependiendo de la implementación de los módulos, estos pueden invocarse entre sí directamente o pueden ser las interfaces del módulo core las que lo realicen. Por ejemplo, el módulo de física puede necesitar utilizar el módulo de matemática para sus cálculos si es que no incluye estos en su implementación.

Módulo de física

Este módulo provee al engine de lo que se conoce como motor de física. La función de este motor es la de simular modelos físicos Newtonianos utilizando variables como masa, velocidad, rozamiento y resistencia al viento. Simula condiciones que se asemejarían a lo que ocurriría en el mundo real. Una de las principales funciones de este motor dentro de un game engine es la del cálculo de colisiones, predicción de trayectorias de los objetos y sistemas de partículas

Existen dos tipos de engines de física, los de tiempo real y los de alta precisión. Estos últimos se utilizan para cálculos de simulaciones extremadamente reales y exactas, cuya finalidad es generalmente científica o cinematográfica. Dada esta precisión, requieren una enorme capacidad de procesamiento y por ende no pueden utilizarse en tiempo real para juegos. En el caso de los juegos se utilizan los motores de física en tiempo real, que realizan un cálculo mas simplificado y menos real, pero aun así lo suficientemente real para que el juego lo parezca.

Antiguamente los motores de física trataban todos los objetos como cuerpos rígidos, de manera de simplificar el modelo y su cálculo. Los motores de hoy en día permiten tratar a los cuerpos como cuerpos blandos, permitiendo realizar efectos de partículas, tela y movimientos de fluidos.

Con el descubrimiento de nuevas tecnologías y técnicas de producción, los procesadores son cada vez más poderosos, lo que permite hacer al motor de física cada vez más realista. Existe también un tipo especial de procesador que se puede incluir en el hardware de una computadora o consola, este procesador de física se lo conoce como PPU o Physics Processing Unit. Este procesador no es nada más que un procesador enteramente diseñado y dedicado para el cálculo físico. Fue diseñado por una empresa llamada Ageia, y el único modelo existente es el Ageia PhysX.

Otra variante de hardware que compite directamente con los PPU son los GPGPU (General Purpose processing on Graphics Processing Unit) o procesador gráfico y de propósito general. Este es un nuevo tipo de procesador gráfico (GPU) que poseen las placas gráficas más recientes, y su principal característica es que además de realizar las funciones normales de un GPU permite el procesamiento acelerado por hardware de cálculos físicos Newtonianos. Este tipo de tecnología ya se puede encontrar en las placas de video Nvidia Geforce de la familia 8 o en las Ati/AMD X1900 XT

Algunos ejemplos de motores de física son: PhysX (ex Novodex), Havok y ODE.

Módulo de matemática

Este módulo provee funciones para la realización de operaciones matemáticas avanzadas que serán necesarias para el game engine. Tales como operaciones con vectores, matrices, quaternions[9] y operaciones de intersección. Algunos ejemplos de librerías de matemática son: OpenMali y Vecmath.

Módulo de redes

Este módulo provee funciones de conectividad de red al engine. Permitiendo crear juegos multiplayer, tanto del lado cliente como del lado servidor. Provee una abstracción del hardware de comunicación, permitiendo portabilidad de plataformas. En el caso de los juegos online en tiempo real, este módulo cobra vital importancia. Debe estar correctamente optimizado y contar con los algoritmos necesarios para corrección de pérdidas, reducción de latencia y seguridad. Algunos ejemplos de librerías de conectividad son: Quazal, GameSpy y Bigfoot Networks.

[9] Quaternion: Son extensiones no conmutativas de números complejos. Fueron descubiertas por el matemático irlandés Sir William Rowan Hamilton en 1843. Su principal uso es en mecánica tridimensional. En el campo de la computación gráfica se utilizan para representar rotaciones y orientación de objetos en el espacio tridimensional.

Módulo de inteligencia artificial

Este módulo provee los algoritmos y operaciones para simular comportamiento en un personaje controlado por la computadora. Los algoritmos utilizados por el módulo de IA puede ser desde simple lógica discreta, pasando por árboles de decisión hasta llegar a modelos más complejos como redes neurales. La representación más simple utilizada para IA es de la máquina de estados, como se construye esta máquina depende de la complejidad del motor de IA, pudiendo ser estática o adaptativa.

El objetivo de este módulo es dar a los personajes controlados por la computadora (NPC[10]) una cierta ilusión de inteligencia que los asemeje a un jugador humano. La funcionalidad provista por este módulo permite al desarrollador o diseñador, definir el comportamiento de los NPCs de acuerdo a diferentes factores, eventos y condiciones que vayan ocurriendo. Entre las funciones más importantes se destacan la de toma de decisiones y búsqueda de caminos (pathfinding). Algunos ejemplos de librerías de inteligencia artificial son: Havok – Behavior, SpirOps, TruSoft y Kynogon.

Al igual que con los motores de física, a medida que la capacidad de procesamiento aumenta o la carga del CPU se libera gracias a otros procesadores dedicados, es posible realizar procesamientos de IA más complejos que hacen a los juegos más reales y desafiantes.

[10] NPC: Non player character. Personaje que no es controlado por el jugador sino por la computadora.

Módulo de animación

Este módulo provee las operaciones y rutinas para agregar animaciones a un modelo. Uno de sus principales usos es en la animación de personajes. Estas librerías proveen facilidades para la animación de rostros, movimiento de cuerpos, sincronización de labios con diálogos, etc. Juegan un papel crucial en la realización de personajes que parezcan y actúen como personas reales, ajustando por ejemplo los músculos faciales para representar una emoción pre definida en el personaje. Algunos ejemplos son: Emotion Fx, Natural Motion, LifeStudio, Annosoft, Havok – Animation.

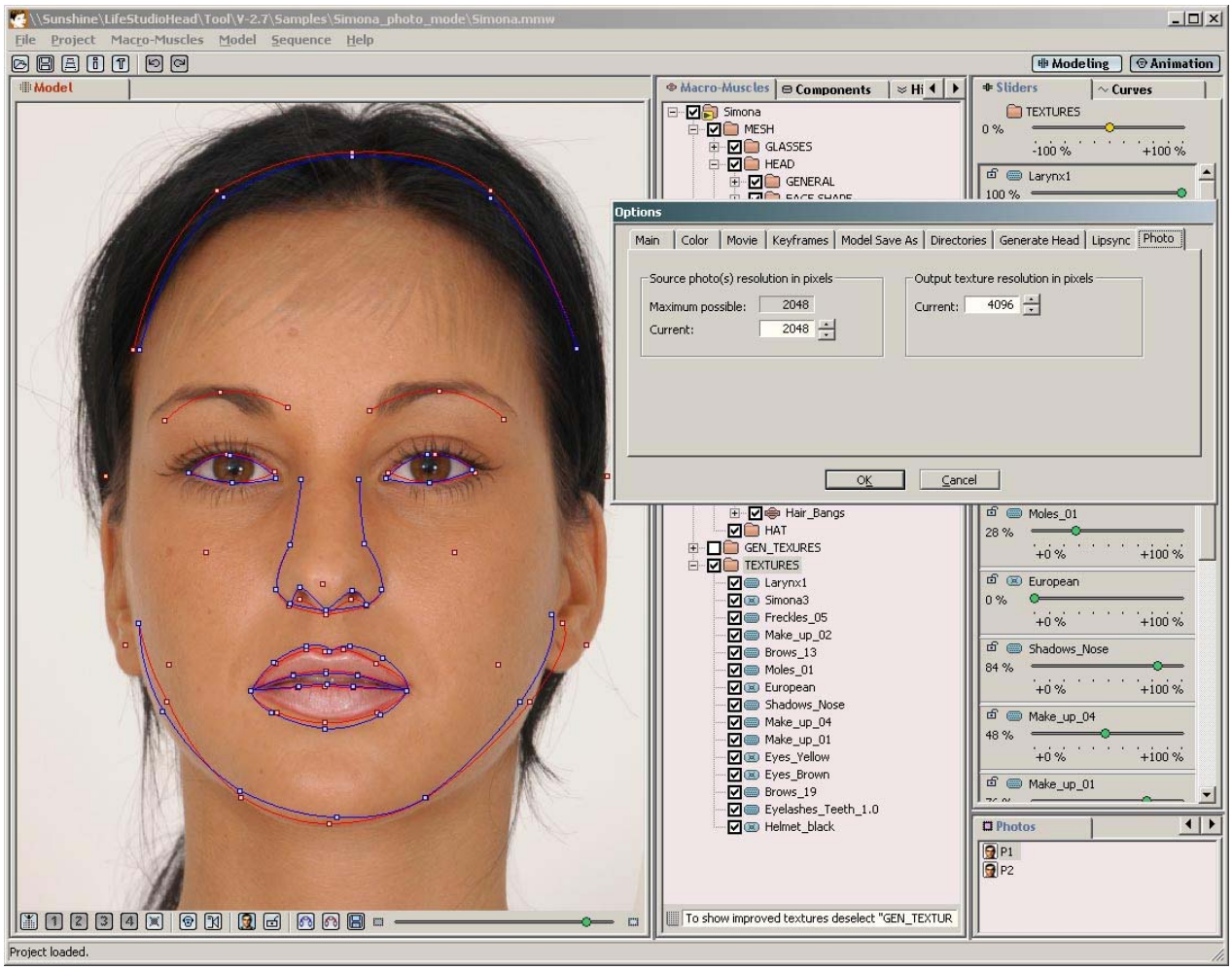

Figura 3.2.2 LifeStudio:Head. Software para animación de rostros

Módulo de scripting

Este módulo provee la implementación del o los lenguajes de scripting utilizados por el game engine para definir las acciones o comportamientos de los objetos del mundo. El componente principal de este módulo es el intérprete del lenguaje que permitirá transcribir operaciones de los scripts en operaciones concretas del game engine accediendo a los objetos de este por medio de un contexto configurable. Algunos ejemplos de lenguajes de scripting utilizados son: UnrealScript, C-Script, GameMakerLanguage y Lua.

Módulo de audio / video

Este módulo cumple la función de controlar las funciones multimedia del engine. Que pueden ir desde reproducir archivos de audio o video. Hasta agregarles efectos. Es también el encargado de saber decodificar estos archivos dependiendo del códec de compresión que estos utilicen. Por lo general para poder realizar estas operaciones el módulo provee una capa de abstracción del hardware de audio y video por medio de librerías estándar tales como DirectX, u OpenAL. Esto le permite independizarse de las funciones de hardware básicas y a su vez proveer implementaciones para hardwares más avanzados tales como las placas de sonido con soporte para sonido envolvente tales como el EAX™ de Creative Labs. Algunos ejemplos de librerías multimedia son: OpenAL, Sensaura, Firelight Technologies y RAD Game Tools.

Módulo de dispositivos de entrada

Este módulo es el encargado de leer y procesar los comandos que el usuario ingresa por cualquiera de los dispositivos de entrada posible, tales como teclado, mouse, joystick, etc. Este módulo está constantemente consultado el estado de estos dispositivos para detectar cambios y proveer a el core y los demás módulos información de que una tecla fue presionada o que el mouse se desplazó una cierta distancia en alguna dirección. Suelen contener algoritmos para filtrar la entrada de datos espurios o para evitar la pérdida de algún dato. Al igual que otros módulos este utiliza librerías y frameworks que permiten abstracción del nivel de hardware. Algunas implementaciones proveen también soporte para hardwares especiales como por ejemplo el teclado G15 de Logitech que provee una pantalla para mostrar información del estado del juego. Algunos ejemplos son: DirectX (DirectInput) y HIAL.

Recursos

Este módulo es un administrador de recursos que los diferentes módulos del engine utilizan. Tales como archivos de modelos, animaciones, audio, video, etc. Provee operaciones simples para acceder a estos a modo de un repositorio común. Administra la carga y descarga de memoria de estos archivos cuando sea requerido.

Render

Este es el modulo más importante del game engine. Es el encargado de la representación gráfica de los objetos del mundo 3D. Entre sus principales funciones además de saber representar en la placa de video los objetos del modelo es la de optimizar los datos que se envían a la placa de video, por ejemplo, eliminando partes de los modelos que no se verían desde donde está viendo el jugador en ese momento, utilizando técnicas tales como culling, árboles BSPs, portales, etc. Se encarga también del manejo de luces, cámaras y todos los algoritmos que aplican a éstas. El rendering es una de las ramas de la computación gráfica más importante y desarrollada.

Este módulo suele soportar también la utilización de programas "shader" que es un tipo especial de código que se ejecuta en la misma placa de video con el fin de generar efectos de rendering. Estos programas se escriben utilizando un set de instrucciones propio del API gráfica utilizada.

El elemento principal de este módulo es la implementación del pipeline gráfico[11], la cual se realiza normalmente utilizando alguna de las dos implementaciones más usadas y conocidas[12]: OpenGL y DirectX. Ambas proveen una implementación del pipeline gráfico que provee una abstracción del hardware. A su vez para el caso del los programas shader proveen implementaciones de alto nivel de estas instrucciones, OpenGL Shading Language, o GLSL, para el caso de OpenGL y High Level Shader Language en el caso de DirectX.

Usualmente, especialmente en los engine gráficos no muy antiguos existe el concepto de scenegraph, como forma de representar el mundo. Este concepto es una representación orientada a objetos, en forma de árbol de los objetos de que componen al mundo, como se componen entre sí y que transformaciones se les aplica. Ampliaremos más este concepto en el capítulo 4.

[11] Ver Capítulo 2.2

[12] Ver Capítulo 2.4

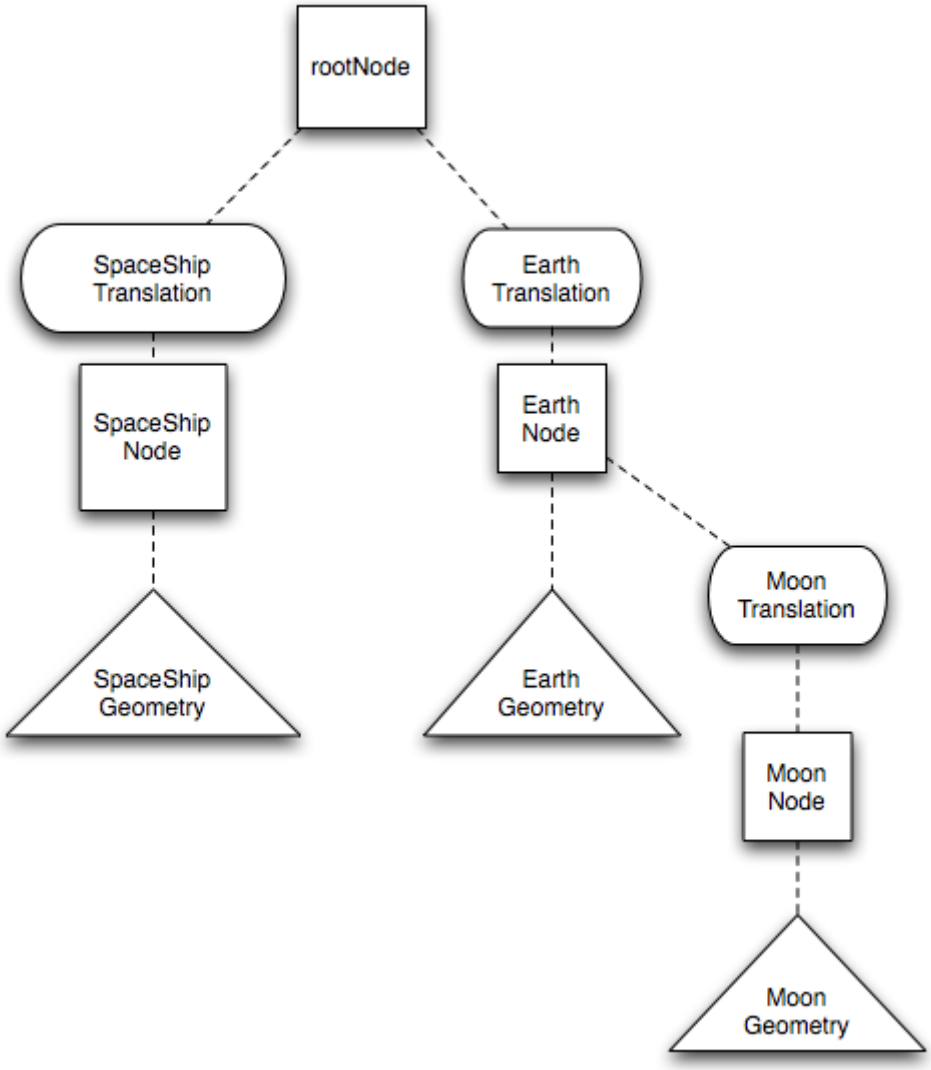

Figura 3.2.3 Ejemplo de scenegraph

En este ejemplo se puede observar el concepto de jerarquía, se puede observar que en el universo existe una nave (spaceship) y el planeta Tierra (Earth) que no están relacionados entre sí, por ende cuelgan del nodo raíz (root). A su vez existe también la Luna (Moon), pero esta gira alrededor de la Tierra y siempre lo hará si importar las condiciones, por ende depende del nodo de la Tierra. Cada objeto a su vez tiene asociado una geometría, que sería su representación gráfica o modelo y una transformación o traslación que puede ser por ejemplo, un desplazamiento en algún eje, o una rotación. En el caso de aplicar una transformación a la Tierra, esta se aplicaría automáticamente a la Luna porque depende de la Tierra. En sí ese es el concepto detrás de un scenegraph.

Core

Es el núcleo del engine, contiene toda la implementación propia del engine que no está delegada los módulos. Se encarga de la utilización de los módulos por medio de las interfaces que este core define para su acceso. Realiza el procesamiento en serio o paralelizado de cada uno de los módulos en los momentos que deben ser invocados (Ej.: el render loop). Está compuesto por varias clases, incluyendo las interfaces para cada módulo. Se puede entonces identificar las siguientes funciones principales:

- Objetos que representan al modelo, escenas, luces, cámaras, etc.
- Control de la ejecución y procesamiento.
- Utilización de los módulos.
- Manejo de configuraciones del engine.
- Operaciones de alto nivel para ser utilizadas en los scripts e implementaciones de los módulos.
- Utilidades para las herramientas gráficas.

Capítulo 4 – Arquitectura de j3dEngine

4.1 Alcance

El trabajo consiste en el diseño y desarrollo prototipo de un engine para el desarrollo y ejecución de escenarios interactivos 3D.

La mayor parte de los 3D engines existentes han sido desarrollados en lenguajes C y C++, principalmente por la alta adopción de estos lenguajes y por las necesidades de performance de este tipo de desarrollos.

Esta performance se logra gracias a que estos lenguajes proveen una gran flexibilidad para el desarrollador ya que el nivel de abstracción de los mismos es cercano al del código máquina. Sin embargo, hay lenguajes más avanzados, en el nivel de abstracción, como java que provee soluciones a problemas complejos típicos (ej: manejo de la memoria dinámica), tiene un nivel semántico superior y lo más importante: hay una enorme comunidad que lo soporta y aporta miles de librerías para el desarrollo de aplicaciones de software. Estas características permiten un desarrollo más veloz y una calidad de código mayor, en comprensibilidad, reusabilidad y mantenibilidad.

Hoy en día la performance de rendering en Java no difiere demasiado de la obtenida en lenguajes de menor nivel. Gracias a una gran comunidad en expansión de desarrolladores, existen muchas librerías open source para el desarrollo de este tipo de aplicaciones. Además, Java es portable, por lo que podemos ejecutar j3dEngine en cualquier sistema operativo con una JVM[13] instalada sin recompilar el código.

[13] Java Virtual Machine

Universidad Austral		Proyecto de Trabajo de Grado	
Proyecto	j3dEngine	**Alumnos**	Alvarez, Pablo N.
			Rodriguez, Ignacio G.
Fecha	29/03/2007	**Director de Trabajo de Grado**	Gonzales Clua, Esteban Walter

Objetivos de j3dEngine

- Provee renderización de escenarios 3D en tiempo real.
- Verificar que es posible desarrollar un engine 3D en Java con funcionalidad similar a la de los engines actuales.
- Basado en una arquitectura *flexible*[14] y *extensible*[15], debe permitir pluggear componentes a través de su configuración para poder agregar nueva funcionalidad.
- Permite interacción con el usuario en tiempo de ejecución.
- Configurable por el usuario:
    - Debe poder asignar comportamiento a los objetos 3D.
    - Debe poder extenderlos con comportamiento custom.
- Desarrollado con librerías open source.
- *Portable*[16].
- Desarrollado en Java y utilizando OpenGL para la renderización 3D.
- Preparado para *multithreading*.

Elementos excluidos del Proyecto

- Características multiplayer y networking
- Sonido
- Motor de física y cálculo de colisiones
- Renderización de terrenos

[14] Flexible: Debe poder configurarse para poder cambiar el modelo de ejecución

[15] Extensible: Deben poder agregarse componentes sin tener que modificar ni recompilar el código exitente.

[16] Portable: Puede ejecutarse en otro ambiente sin necesidad de recompilar el código.

- Motor de inteligencia artificial
- Objetivos de performance[17]

Un Game Engine 3D, como vimos, brinda funcionalidad para ejecución de escenarios virtuales y una serie de herramientas para su diseño, desarrollo y configuración. Hemos seleccionado algunas de estas funciones para crear un primer diseño e implementación fundamentales para j3dEngine.

Multithreading

Hemos decidido no enfocar el trabajo en lograr una performance que compita con los engines existentes, pero sí debimos considerar un elemento importante que afecta a la definición de la arquitectura de j3dEngine: el multithreading.

Hoy en día las computadoras de escritorio ya disponen de múltiples procesadores (o procesadores que lo emulan[18]) y hace ya años que los sistemas operativos más utilizados brindan herramientas de procesamiento paralelo. Es por ello que hemos decidido que j3dEngine disponga de una arquitectura apta para aprovechar el multithreading.

Uno de los objetivos es lograr un modelo que permita configurar fácilmente los threads de procesamiento.

4.2 Herramientas para la Implementación

Una de las funcionalidades principales que j3dEngine brinda es la renderización de escenarios 3D en tiempo real. Este procesamiento es complejo y de alto costo para el CPU y la placa gráfica.

Para poder manejar escenarios complejos en memoria, es necesario utilizar estructuras de datos que permitan una renderización performante.

[17] No realizamos comparación con otros engines en los FPS obtenidos durante la ejecución.

[18] En referencia a la tecnología *Hyperthreading* de Intel

SceneGraph

Para la renderización de escenarios 3D se ha creado una estructura de datos conocida como *SceneGraph*. No tiene una definición formal estándar, pues es normal que cada vendor de engine 3D utilice su propio scenegraph.

Sin embargo hay ciertas características comunes a todos los scenegraph y es conveniente no tratar de inventar uno nuevo, sino utilizar uno existente o definirlo aprendiendo de los que ya se utilizan comercialmente.

Para j3dEngine utilizamos un scenegraph open source desarrollado en Java, se trata de *Xith3D*. Xith3D a la vez está basado en el scenegraph de Java 3D, que es el framework de desarrollo de aplicaciones 3D en Java desarrollado y distribuido por Sun.

La palabra scenegraph tiene dos acepciones. Un scenegraph es una estructura de datos pero a la vez se le llama scenegraph a las librerías de renderización de escenarios 3D.

Un scenegraph es un grafo cuyos nodos representan los elementos visibles y no visibles de un escenario 3D. Se trata de un grafo de estructura arbolada, pero no es estrictamente un árbol.

Los nodos de un scenegraph mantienen una relación *padre-hijo* para modelar la estructura de un escenario 3D (los objetos se armar a través de la composición de otros). El conjunto de nodos y arcos padre-hijo forman un árbol, que es un subgrafo dentro del scenegraph. Una característica importante del árbol de un scenegraph es que debe ser un DAG (Directed Acyclic Graph) o Grafo Dirigido Acíclico. Esto implica que los arcos que relacionan a los nodos del grado son dirigidos y no deben formar ciclos. Es por ello que un nodo no puede tener dos padres, y hay un único camino en el grafo desde la raíz hasta un nodo hoja.

Los elementos conocidos como NodeComponent se relacionan con los elementos del árbol a través una relación de *referencia*.

El scenegraph permite organizar un escenario 3D, ya que relaciona a los objetos 3D en su árbol. Almacena la información de los objetos 3D, su geometría y transformaciones, comportamiento, animaciones, sonido y lo todo lo que sea necesario para modelar los escenarios. Esta estructura facilita el desarrollo de aplicaciones 3D, ya que agrega una capa de abstracción entre las llamadas a la librería gráfica para renderizar y la lógica de aplicación.

Cada camino desde la raíz del scenegraph hasta una hoja determina completamente la información de estado geométrico (tamaño, ubicación y orientación) de un objeto del universo virtual. Los atributos visuales de un objeto dependen de su ubicación en el grafo. Esta característica es una de las razones por la que utilizar un scenegraph permite lograr una rápida renderización de un escenario 3D.

Xith3D es un scenegraph, almacena la información del escenario en una estructura de datos arbolada y a la vez provee un renderer, el cual recorre el scenegraph interpretándolo para transformarlo en polígonos y finalmente en invocaciones a la librería gráfica.

Xith3D utiliza la estructura de scenegraph de Java 3D.

Esta estructura tiene un nodo raíz. El *Virtual Universe*. De éste se desprenden los demás nodos de la estructura arbolada.

El scenegraph se divide en dos grandes ramas (o subgrafos):

- Rama de contenido: sus nodos representan el contenido del universo virtual, o sea, geometría, apariencia, comportamiento, sonido, ubicación, orientación e iluminación.

- Rama de vista: representa la ubicación de las vistas y su dirección.

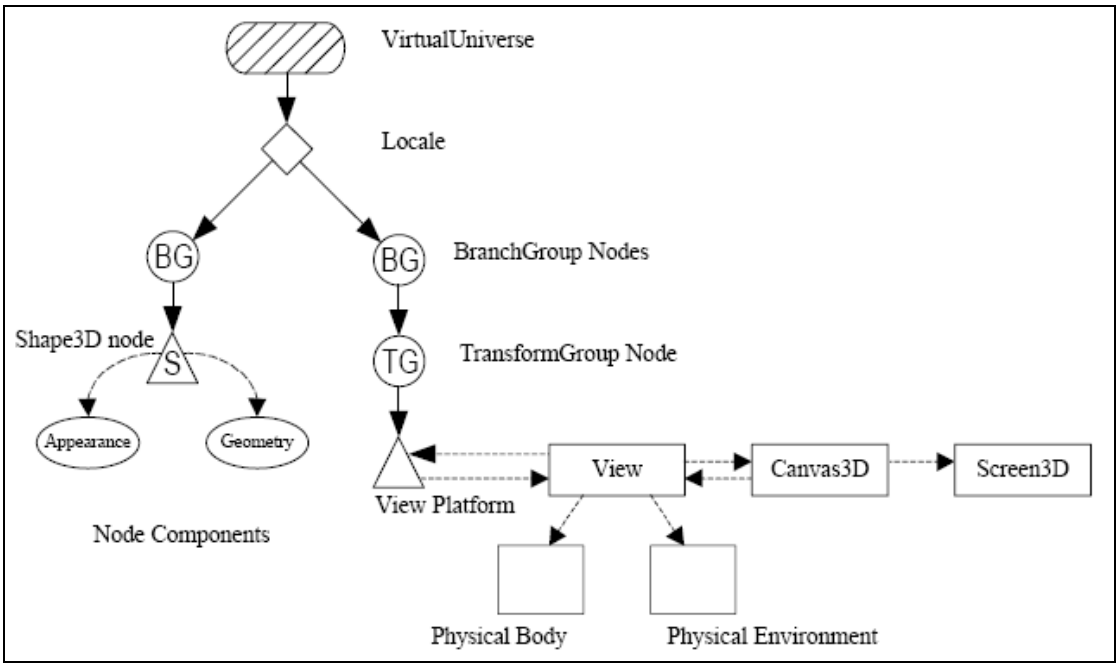

Figura 4.2.1 – Ejemplo de un SceneGraph de Java 3D [J3DT]

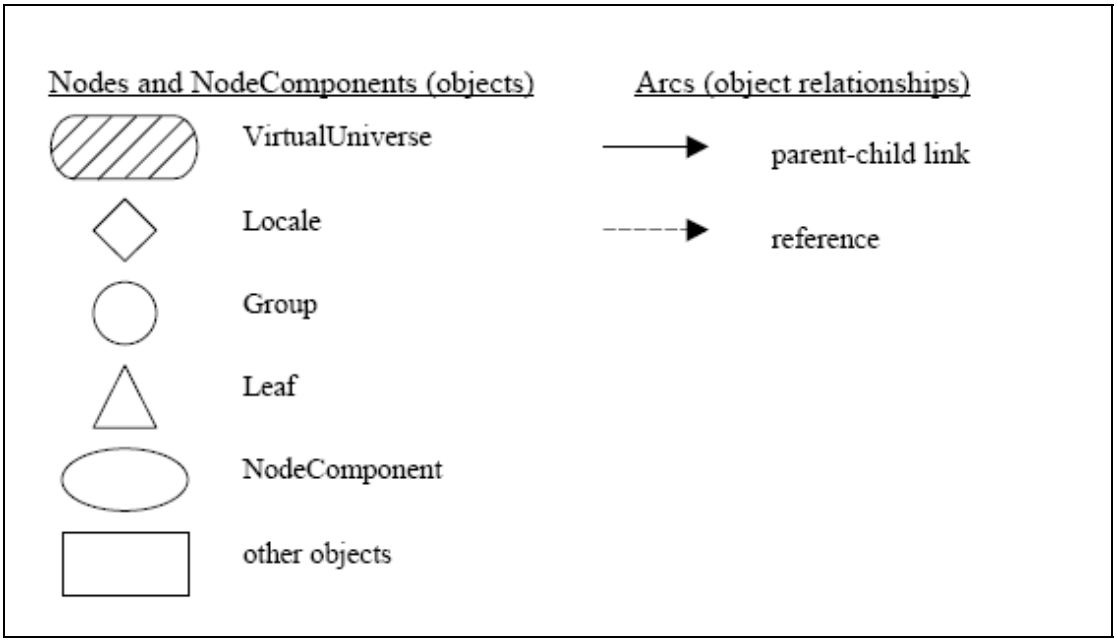

Figura 4.2.2 – Símbolos utilizados para representar un scenegraph [J3DT]

Los tipos de nodos que componen al scenegraph de Java 3D son:

- Virtual Universe: Raíz del universo virtual[19].

- Locale: Representa el centro geométrico del universo virtual, para el sistema de coordenadas global. Es el punto de referencia para determinar la ubicación de los elementos visuales dentro del universo virtual.

- Nodo (Node): El nodo representa a un elemento del árbol del scenegraph (subgrafo de contenido). Puede ser un grupo o un nodo hoja.

- Grupo (Group): Agrupa nodos, que son hijos de éste. Puede ser un BranchGroup (BG) o un TransformGroup (TG). Un BranchGroup simplemente agrupa nodos hijos, un TransformGroup define transformaciones geométricas que afectan a todos sus hijos.

- Hoja (Leaf): Es la representación concreta de un objeto del universo virtual. Hay varios subtipos de hoja: figura 3D, luz, comportamiento, sonido. No tienen hijos pero pueden referenciar NodeComponents.

- NodeComponent: Define atributos de las hojas del scenegraph. Puede ser referenciado por más de una hoja, pero no forma parte del árbol de contenido. Algunos subtipos son: geometría, apariencia, textura, material.

- Vista (View): Representa una cámara o posición de observador, y una dirección de observación. Indica el punto geométrico inicial a partir del cual iniciar el proceso de renderización.

[19] Los términos *mundo virtual*, *escenario 3D* y *universo virtual* se consideran sinónimos en el contexto de este documento.

Librerías Third-party

La implementación prototipo de j3dEngine no incluye todos los componentes de un Engine 3D comercial. Decidimos incluir sólo los módulos básicos que fuesen necesarios para validar la arquitectura.

Para la implementación de esos componentes, hemos utilizado librerías open source que se encuentran en desarrollo y son ampliamente utilizadas por la comunidad Java que se dedica al desarrollo de aplicaciones 3D.

Xith3D

http://www.xith.org/

Para la implementar el rendering[20] y la carga de modelos 3D, hemos utilizado Xith3D.

Xith3D es un scenegraph creado en 2003 para el desarrollo de juegos. Está basado en Java 3D pero tiene entre sus objetivos lograr una mayor performance que la de éste. Uno de los problemas principales de Java 3D es que su scenegraph es thread safe[21], lo cual es en muchos escenarios innecesario y agrega un overhead que afecta a la performance del rendering sensiblemente.

Hay una gran comunidad activa que aporta al desarrollo de este scenegraph ya sea programándolo o utilizándolo para el desarrollo de aplicaciones 3D.

Xith3D está compuesto por dos proyectos: Xith3D core y toolkit. El core es el scenegraph, la estructura de datos para modelar escenarios 3D y el renderer. El toolkit es un conjunto de librerías desarrolladas por terceros, es decir, no miembros del equipo de desarrollo de Xith3D. Este toolkit tiene, por ejemplo, librerías para el cálculo vectorial, manejo de dispositivos de entrada y *loaders*[22] para los formatos de modelos 3D más utilizados.

[20] Rendering y Renderización se utilizan en este documento como sinónimos.

[21] No puede ser modificado simultáneamente por más de un thread de ejecución.

[22] Herramienta que interpreta un modelo 3D en algún formato específico y lo transforma a un scengraph de Xith3D.

Openmali

https://openmali.dev.java.net/

Openmali es una librería open source para cálculo vectorial, que provee funciones típicamente utilizadas en aplicaciones 3D. Xith3D utiliza esta librería para el cálculo geométrico necesario en el rendering.

Es una implementación abierta de Vecmath, una especificación estándar de funciones matemáticas.

HIAL (Hybrid Input Abstraction Layer)

http://input.jtank.net/

Para el manejo de dispositivos de entrada (teclado y mouse) hemos utilizado HIAL.

HIAL agrega una capa de abstracción al manejo de dispositivos de entrada. Está diseñado para proveer una interfaz independiente de la implementación del manejo del hardware.

Para manipular estos dispositivos, existen implementaciones como la de Java (parte de la librería gráfica AWT) o la de lwjgl. Con HIAL podemos cambiar de implementación sin modificar el código de nuestra aplicación.

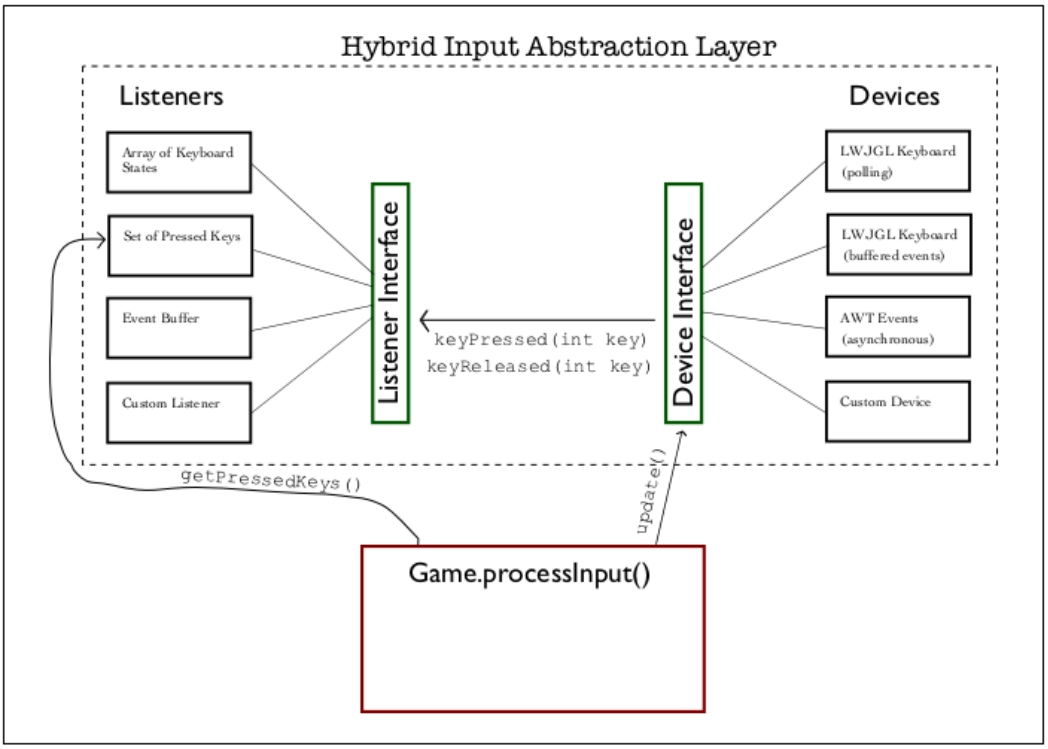

Figura 4.2.3 – Hybrid Input Abstraction Layer. (http://input.jtank.net/)

Jogl

Jogl ya fue explicado en el capítulo 2. Es una librería que permite a Java realizar llamadas nativas a la librería gráfica OpenGL para el proceso de rendering.

La arquitectura de Xith3D está preparada para cambiar de implementación de librería gráfica, es decir, puede cambiar de jogl a lwjgl simplemente modificando la configuración.

JAXB (*Java Architecture for XML Binding*)

Para guardar la configuración de los escenarios y la ejecución de j3dEngine hemos utilizado archivos en formato XML[23]. Para poder realizar esto, se debe implementar un serializador/deserializador de objetos Java a XML. Pero en vez de implementarlo decidimos utilizar una librería que forma parte del stack actual de web services de Sun.

JAXB permite mapear objetos Java a XML y viceversa con poca codificación, agregando al código de los objetos Java annotations[24] descriptivas y algunos archivos configuración.

JUnit

Es importante durante el desarrollo ir agregando tests automáticos. No sólo ayudan a mejorar el desarrollo porque obligan a repensar los diseños, además ayudan notablemente a mejorar la calidad del producto obtenido. Lo más importante de los tests automáticos no es que permiten verificar que funciona correctamente el componente desarrollado, sino que permiten verificar que su funcionalidad no fue dañada por el desarrollo de otros componentes relacionados, por ser fácilmente repetibles. Esto es muy difícil de lograr con el testing manual, ya que raramente una persona vuelve a verificar lo que ya ha verificado que funcionaba anteriormente.

Para los tests automáticos utilizamos JUnit, la librería más usada en proyectos de desarrollo Java.

[23] XML: Extensible Markup Language

[24] Annotations: Herramienta del lenguaje Java que permite agregar meta información a las clases codificadas.

Licencias

Para el desarrollo de cualquier aplicación o cualquier librería, es esencial la utilización de recursos preexistentes. Dada la complejidad de los desarrollos actuales, es prácticamente imposible concretar un proyecto sin utilizar frameworks o librerías de terceros.

Al seleccionar una librería third-party, no sólo es importante validar que provee la funcionalidad que requiere nuestro proyecto, también es esencial prestar atención al tipo de licencia que ofrece.

En función del objetivo final de nuestro proyecto, debemos elegir los tipos de licencia third-party que nos convengan, de lo contrario podemos acabar con problemas legales que pueden dar fin a la vida del proyecto.

En nuestro caso, elegimos como librería principal para el proyecto a Xith3D, la cual posee una licencia de tipo *BSD*.

BSD (*Berkeley Software Distribution*) es una licencia de código abierto que establece que el producto puede ser utilizado por cualquier persona para desarrollar cualquier producto derivado con cualquier propósito sin ningún requerimiento de retorno para el creador del producto licenciado. También indica que el dueño de la licencia no responde por el funcionamiento del producto.

Para la librería de cálculo vectorial hemos utilizado openmali (implementación de Vecmath). Existe una implementación de referencia de Vecmath pero que posee una licencia de tipo JDL (*Java Distribution License*) la cual aplica ciertas restricciones cuando el producto final tiene fines comerciales. En cambio openmali utiliza una licencia LGPL (*Lesser General Public License*) que permite a quien la utilice permite a quien utilice la librería modificarla y renombrarla si es necesario.

La licencia que hemos seleccionado para j3dEngine es de tipo LGPL, la cual es una de las más utilizadas para distribuir abiertamente una librería.

4.3 Arquitectura de j3dEngine

La arquitectura de j3dEngine fue diseñada con el objetivo de que sea extendida fácilmente. En nuestra implementación prototipo, no hemos implementado la funcionalidad de física, audio, networking ni inteligencia artificial. Sin embargo j3dEngine debe permitir que se le agregue esta funcionalidad sin sufrir cambios en el código.

No debería ser necesario que el desarrollador usuario de j3dEngine tenga que modificar el código, sólo debe agregar a su directorio de librerías la de j3dEngine y desarrollar las extensiones en un módulo independiente.

j3dEngine está preparado para ser modificado desde una configuración en formato XML, lo cual le otorga una gran flexibilidad de extensión.

Módulos de la Arquitectura

Un engine 3D está básicamente compuesto por dos arquitecturas, una para el desarrollador de código (SDK) y una de herramientas (Toolkit).

Los módulos que componen la arquitectura SDK de j3dEngine son:

- *Core Objects*
- *Processors*
- *GameAction API*

Más un tool para la configuración de j3dEngine. El *Configuration Editor*.

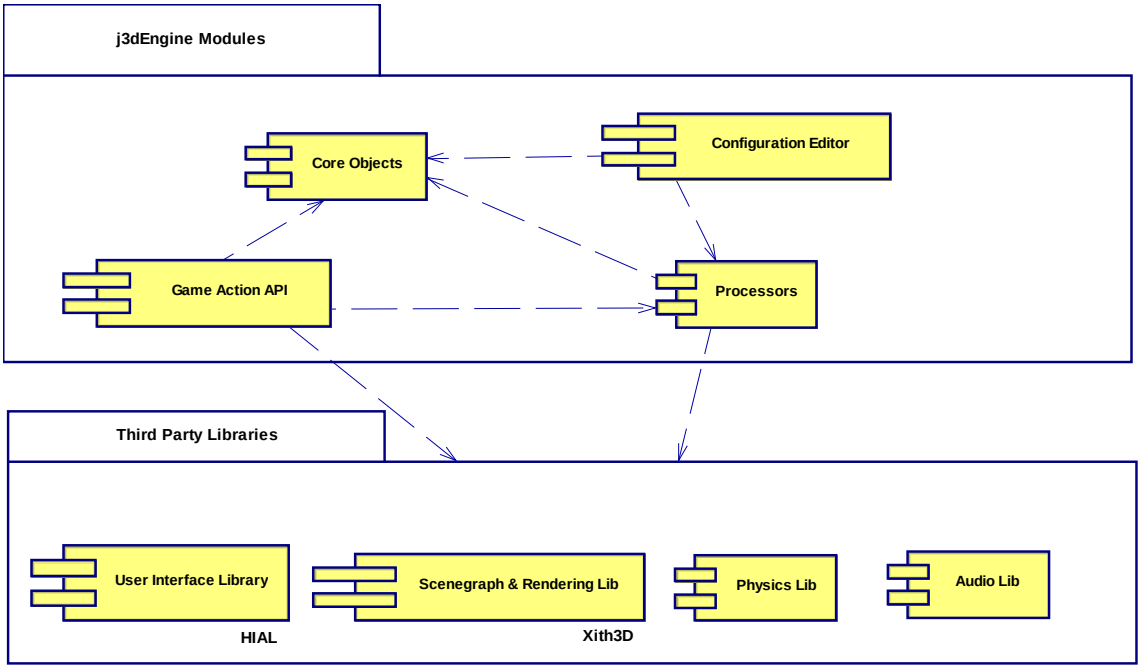

Figura 4.3.1 – Módulos y dependencias de la arquitectura de j3dEngine.

Como se ve en la **Figura 4.3.1**, los módulos del SDK Game Action API (GAPI) y Processors dependen del módulo CoreObjects, que es el módulo que tiene las clases principales. Estos módulos dependen a la vez de las librerías third-party, utilizan esas librerías para su implementación.

Funciones de cada Módulo

CoreObjects

CoreObjects es el módulo principal de la arquitectura. Tiene los objetos fundamentales para la creación de un escenario 3D. En j3dEngine, llamamos *World* al conjunto de objetos 3D que conforman un escenario.

En la **Figura 4.3.2** puede verse el diagrama de las clases principales de CoreObjects. La clase central es *GameObject*, y representa cualquier objeto de juego, ya sea visible o no, estático o dinámico. Un tipo especial de GameObject es el *World,* objeto que referencia a todos los objetos del escenario cargados en memoria en run-time y lleva el control de algunas variables de juego como el tiempo (Game Time).

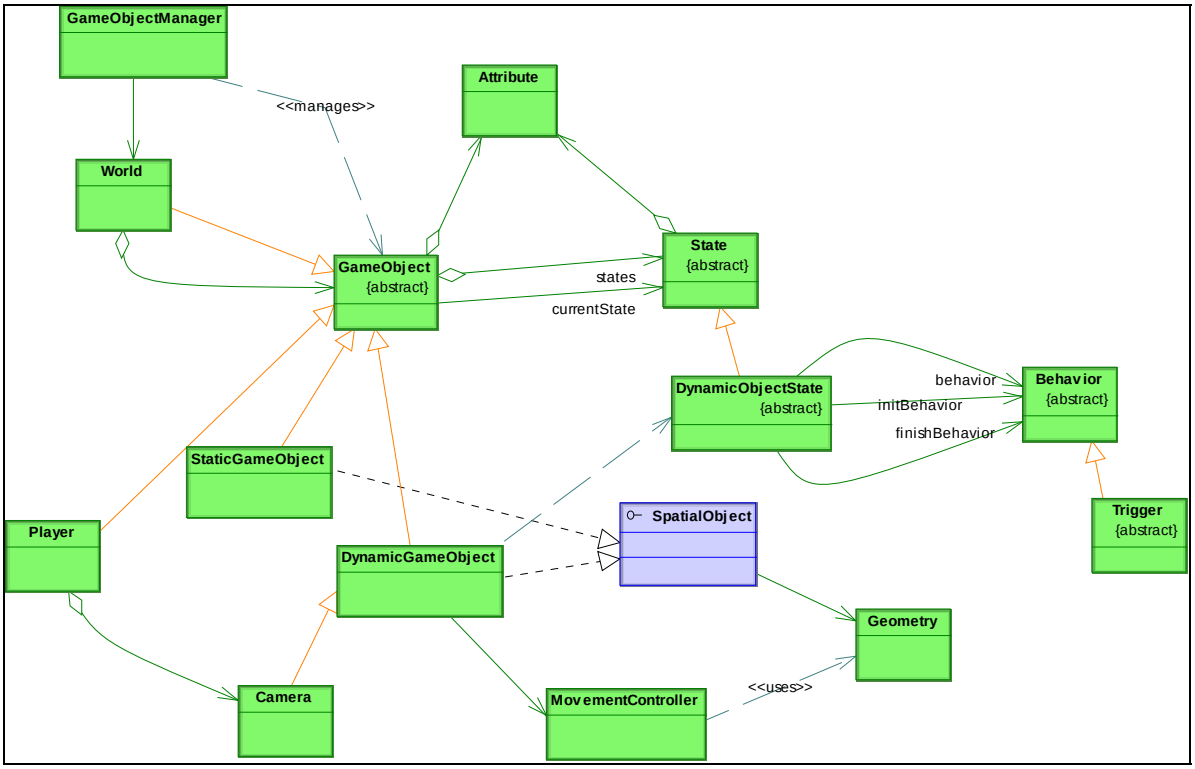

Figura 4.3.2 – Diagrama de clases del módulo CoreObjects.

El *GameObjectManager* es el encargado de cargar el World desde un archivo XML y administrar los GameObject. Provee un servicio de lookup para éstos, centralizando el acceso a la información del juego en run-time.

Un GameObject tiene atributos y estado. El estado de un GameObject también tiene atributos propios.

Otro subtipo especial de GameObject es el Player, objeto que representa a un jugador y guarda toda la información de su estado. Relacionado con este objeto también existe otro, el *Camera*, que representa un punto de referencia y una dirección de observación para generar una vista sobre el World.

Un GameObject puede o no referenciar a un *Geometry*, sólo los *SpatialObject* lo hacen. En este caso se trata de un objeto visible. En la implementación de j3dEngine, la geometría de cada objeto está representada por un nodo en el scenegraph, y para lograrlo existe un adapter [25]para el Geometry que adapta el modelo del scenegraph a la interfaz de Geometry conocida por los objetos core.

[25] Adapter: Design pattern que sirve para adaptar una interfaz desconocida en una conocida por los objetos clientes.

Los GameObject se subdividen en dos tipos genéricos: *StaticGameObject* y *DynamicGameObject*. A menos que sea necesario, no debe crearse una clase nueva para cada objeto que queremos agregar al World. Un objeto genérico es, en run-time, una instancia de un StaticGameObject o un DynamicGameObject. Los StaticGameObject son GameObject inanimados. Los DynamicGameObject son objetos de juego, dinámicos, visibles o no. Tienen un *MovementController,* el cual se encarga de controlar el movimiento de los mismos, en el caso de que tengan una geometría, transformando la posición y orientación de la misma durante el juego.

Para los DynamicGameObject existe un subtipo de *State,* el *DynamicObjectState* el cual almacena comportamiento. El comportamiento está representado por la clase *Behavior* y depende del estado del DynamicGameObject. Cada estado puede tener un comportamiento de activación y uno de desactivación, y tiene un comportamiento designado que será el comportamiento del DynamicGameObject durante el tiempo que se encuentre en ese estado. Esto se logra aplicando el pattern State[26] que permite modificar el comportamiento del objeto en run-time.

Al cambiar de estado, el comportamiento de desactivación del estado anterior se ejecuta, y luego se ejecuta el comportamiento de activación del nuevo estado. Esto sirve para permitir modificar atributos del objeto durante un cambio de estado.

Processors

El módulo Processors contiene la lógica de ejecución de j3dEngine.

El objeto principal es el *Processor*, cuyas implementaciones definen la lógica de ejecución de cada dominio de j3dEngine.

Existe un Processor para cada dominio o funcionalidad: *User Interface, Rendering, GameLogic, Physics, Audio* y cualquier Processor con el cual se quiera extender la funcionalidad de j3dEngine. Cada Processor hace uso de las librerías necesarias para llevar a cabo la ejecución correspondiente. En la **Figura 4.3.3** puede verse el diagrama con las clases más importantes del módulo.

[26] State: Design pattern que permite cambiar el comportamiento de un objeto en run-time, como si el objeto cambiase de tipo durante la ejecución.

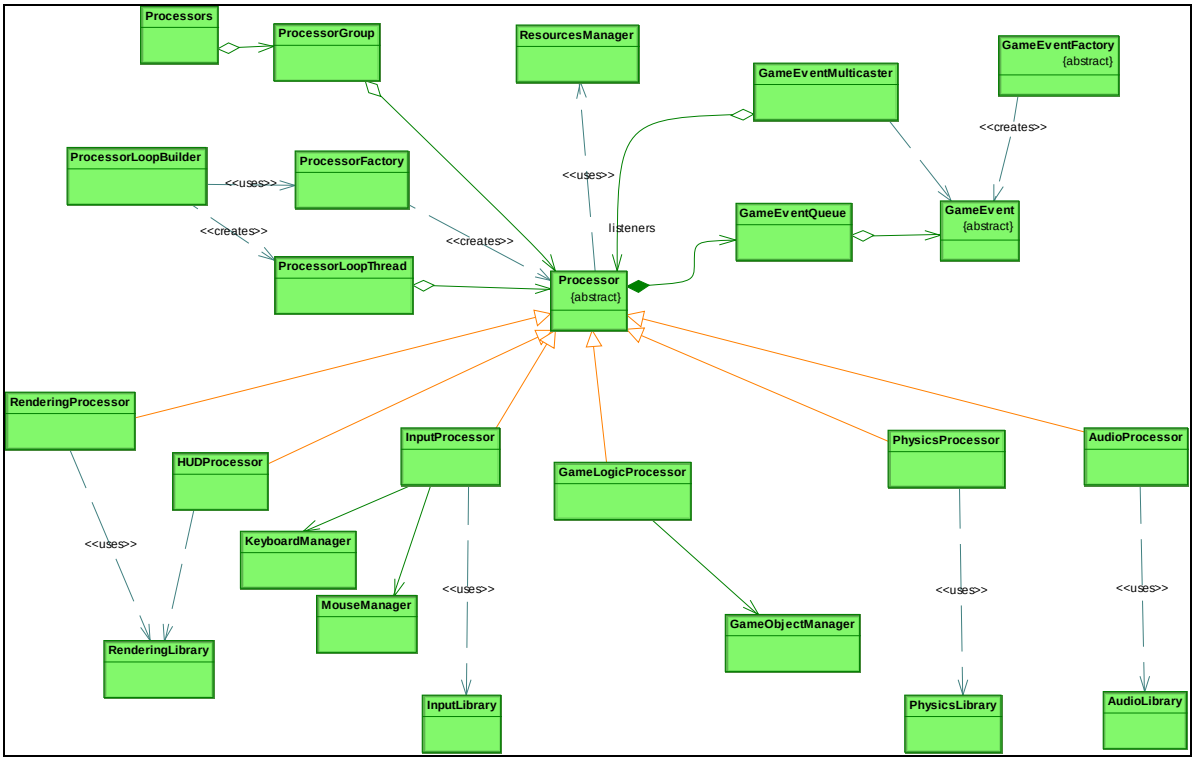

Figura 4.3.3 – Diagrama de clases del módulo Processors.

En este sentido j3dEngine es extensible. Por ejemplo, si deseo agregar una funcionalidad nueva para cálculo de colisiones utilizando una librería existente, sea PhysicsLib, debo agregar un Processor, sea PhysicsProcessor. En ese caso j3dEngine dispondría de un "Physics Engine" que estaría compuesto por el PhysicsProcessor más el PhysicsLib.

Para agregar este Processor simplemente debo cambiar el archivo XML de configuración de la ejecución de j3dEngine y asegurarme de que el classloader[27] de j3dEngine tenga acceso a las clases del nuevo Processor.

Además de procesar la lógica de su dominio, cada Processor debe consumir los eventos externos (*GameEvent*) que son colocados durante la ejecución en la cola de eventos (*GameEventQueue*). El *GameEventMulticaster* de encarga de ejecutar el dispatch de los eventos, enviándolos al Processor que corresponda.

Los Processor implementados en el prototipo son el RenderingProcessor, el InputProcessor y el GameLogicProcessor.

---

[27] Classloader: Componente de la Java Virtual Machine que busca y carga en memoria la definición de las clases Java.

La ejecución de j3dEngine es iterativa. Los Processor se configuran en un loop de ejecución que se ejecuta repetitivamente hasta que se detenga la ejecución de j3dEngine.

Aunque no nos enfocamos en la performance de j3dEngine y no realizamos pruebas sobre su rendimiento, la arquitectura de j3dEngine debe permitir que sea mejorada y optimizada. Para aprovechar las capacidades multi-threading que provee el sistema operativo y los nuevos CPU multicore, j3dEngine permite disponer a los Processor en distintos threads de ejecución.

Los Processor se agrupan en un ProcessorLoopThread, y cada instancia de éste se convierte en un thread de ejecución en run-time. Esta agrupación y el orden de ejecución de los Processor se define en el archivo XML de configuración de ejecución.

Por ejemplo:

```
<processors>

   <processorGroup priority="9" name="Group 1">
      <processor>
            edu.ua.j3dengine.processors.rendering.RenderingProcessor
      </processor>
    </processorGroup>

    <processorGroup priority="7" name="Group 2">
      <processor>
            edu.ua.j3dengine.processors.input.InputProcessor
      </processor>
      <processor>
            edu.ua.j3dengine.processors.GameLogicProcessor
      </processor>
    </processorGroup>

</processors>
```

En esta configuración se han dispuesto dos loops de ejecución. Uno principal que ejecuta el RenderingProcessor y uno secundario que ejecuta en orden el InputProcessor y el GameLogicProcessor. Esta configuración asume que el rendering y el resto de la lógica son independientes y por lo tanto, paralelizables. Vemos que se supone también, que es necesario para ejecutar la lógica del juego (Game Logic), haber ejecutado primero la actualización de los dispositivos de entrada (Input).

Cada loop tiene una prioridad que será la que interpreta el scheduler de threads para priorizar la ejecución de los mismos.

Para ejemplificar, un ProcessorLoopThread podría tener un pseudo-código como el siguiente:

```
class ProcessorLoopThread : Thread{

	run(){
		while(active){
			for each processor in processors{
				processor.execute();
			}
			//...

		}
		//...
	}
}
```

Luego el GameLogicProcessor podría ser algo como:

```
class GameLogicProcessor : Processor{

	execute(){
		for each event in eventQueue{
			processEvent(event);
		}

		objects = GameObjectManager.getAllDynamicObjects();
		for each object in objects{
			object.update();
		}
		//...
	}
}
```

Se ve en el código que en cada iteración el ProcessorLoopThread se ejecutan en orden todos los Processor que se configuraron dentro de éste.

En la arquitectura de Processors, también se encuentra el *ResourceManager*. El ResourceManager busca y carga en memoria recursos que j3dEngine necesite en run-time. Puede cargar texturas y modelos 3D. Tiene la capacidad de cargar modelos en formato de 3D Studio (3DS), Wavefront (OBJ), ASE y Collada (DAE) entre otros.

GameAction API (GAPI)

Este módulo provee la funcionalidad para ejecutar acciones que afectan al World. Las acciones están encapsuladas los objectos *GameAction*. Hay distintos tipos de GameAction y pueden agregarse los que sean necesarios. Los GameAction utilizan la funcionalidad provista por otros módulos para implementar las operaciones.

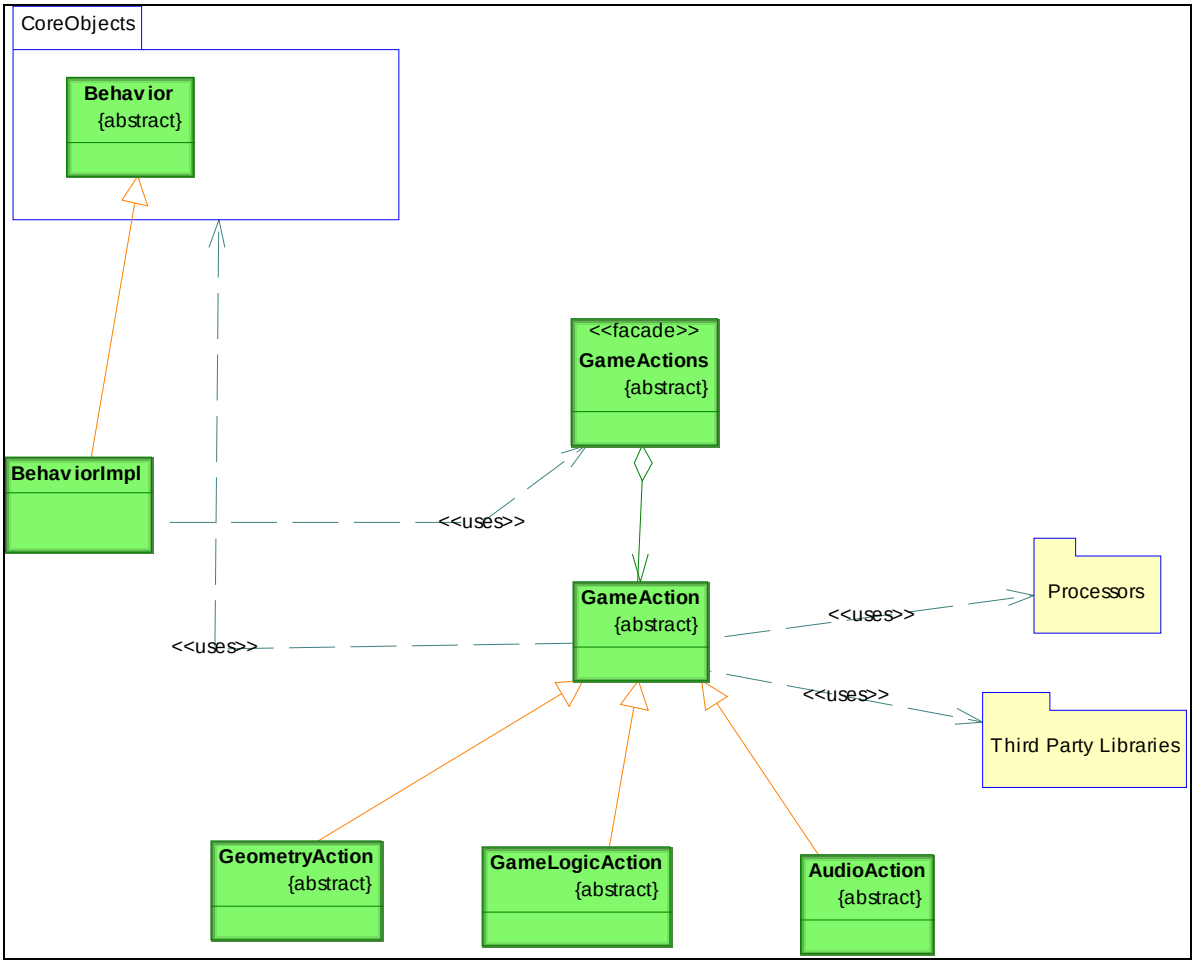

Figura 4.3.4 – Diagrama de clases del módulo GameAction API.

En la **Figura 4.3.4** pueden visualizarse las clases principales del módulo GAPI. La clase más importante es *GameActions.* Se trata de un façade[28] que provee un punto de acceso centralizado a las operaciones y delega las invocaciones en los GameAction. Esconde así la complejidad de la implementación de cada operación.

GAPI es la interfaz de programación para los Behavior de los DynamicGameObject. Accediendo a la funcionalidad de j3dEngine a través de una misma clase y con métodos simples, se logra un desarrollo mucho más sencillo y veloz.

Uno de los objetivos que pensamos y no agregamos al alcance del proyecto es el de agregar un Engine de scripting a j3dEngine. Los scripts son funciones o pequeños

[28] Façade: Design pattern que permite ocultar la complejidad de un framework o API proveyendo un punto de acceso centralizado y una interfaz más sencilla.

programas cuya lógica define el comportamiento de los objetos dinámicos. Para poder agregar la capacidad de scripting, es esencial que se pueda codificar un comportamiento con llamadas sencillas.

Por ejemplo, si desarrollo un comportamiento para un personaje 3D, el cual debe moverse hacia alguna dirección en función de una tecla del teclado que es presionada por el jugador, una forma de lograrlo sería:

```
class MyBehavior : Behavior{

      execute(){
            obj = GameObjectManager.getInstance().getObject(“myobj”);
            obj.getGeometry().getTransform3D().setTranslation(…);
            //...
      }
}
```

En cambio, utilizando las funciones de GAPI, un comportamiento podría codificarse como:

```
class MyBehavior2 : Behavior{

      execute(){
            move(“myobj”, …);
            //...
      }
}
```

ConfigurationEditor

Existen dos tipos de archivos de configuración para j3dEngine, los dos tienen formato XML. Un archivo es el de configuración de ejecución, el cual se utiliza para seleccionar el World que se va a ejecutar y la configuración de processors y threads para la ejecución.

Otro archivo que define la estructura del World. Se trata de un árbol cuya raíz es el World y que está compuesto por los objetos 3D que forman parte del mismo. En este archivo se configuran los objetos, sus nombres, su estado, comportamiento y su geometría (por ejemplo, el archivo del modelo 3D de donde debe cargarse la geometría).

j3dEngine en ejecución

Una vez configurados el archivo del World y el archivo de ejecución, puede ejecutarse el run-time de j3dEngine.

Lo primero que ocurre es la creación de un *GameEnvironment* a partir del cual se inicializan los recursos necesarios y el GameObjectManager. El GameObjectManager se encarga de cargar en memoria el World que se va a renderizar.

El *ProcessorLoopBuilder* lee la configuración de ejecución para inicializar y ordenar los Processor. Una vez terminado este proceso de inicialización, comienza el proceso del pipeline gráfico.

Es importante notar que la posibilidad de configurar más de un thread de procesamiento permite paralelizar algunas partes del pipeline gráfico, y así obtener un mayor throughput, que se traduce en mayor FPS.

Los threads ejecutan cada uno un loop de invocación de Processors. Al final de cada iteración, se obtiene un frame o cuadro en pantalla.

Tomemos como ejemplo un solo thread que ejecuta al InputProcessor, GameLogicProcessor y finalmente al RenderingProcessor. Puede verse el ejemplo en la **Figura 4.3.5**, en la cual se muestra la secuencia de ejecución de una iteración del thread.

Recordemos que en j3dEngine, cada thread está implementado a través de un ProcessorLoopThread.

En el paso **1** se ejecuta el InputProcessor. Éste invoca al KeyboardManager y MouseManager para actualizar el estado del teclado y mouse.

En el paso **4** se ejecuta el GameLogicProcessor. Accede a los objetos dinámicos a través del GameObjectManager y luego actualiza su estado. Notemos que el comportamiento que se ejecuta para cada objeto depende del estado que tiene en esa iteración, como se muestra en el paso **8**. Cada vez que se invoca al GameLogicProcessor también se actualiza el estado del World, el cual lleva cuenta del tiempo de juego o game time. El intervalo de game time entre frames depende de la diferencia de tiempo real entre iteraciones de la ejecución.

Finalmente en el paso **10** se ejecuta el RenderingProcessor, el cual invocará a la librería de rendering para interpretar el scenegraph y generar el frame para esa iteración.

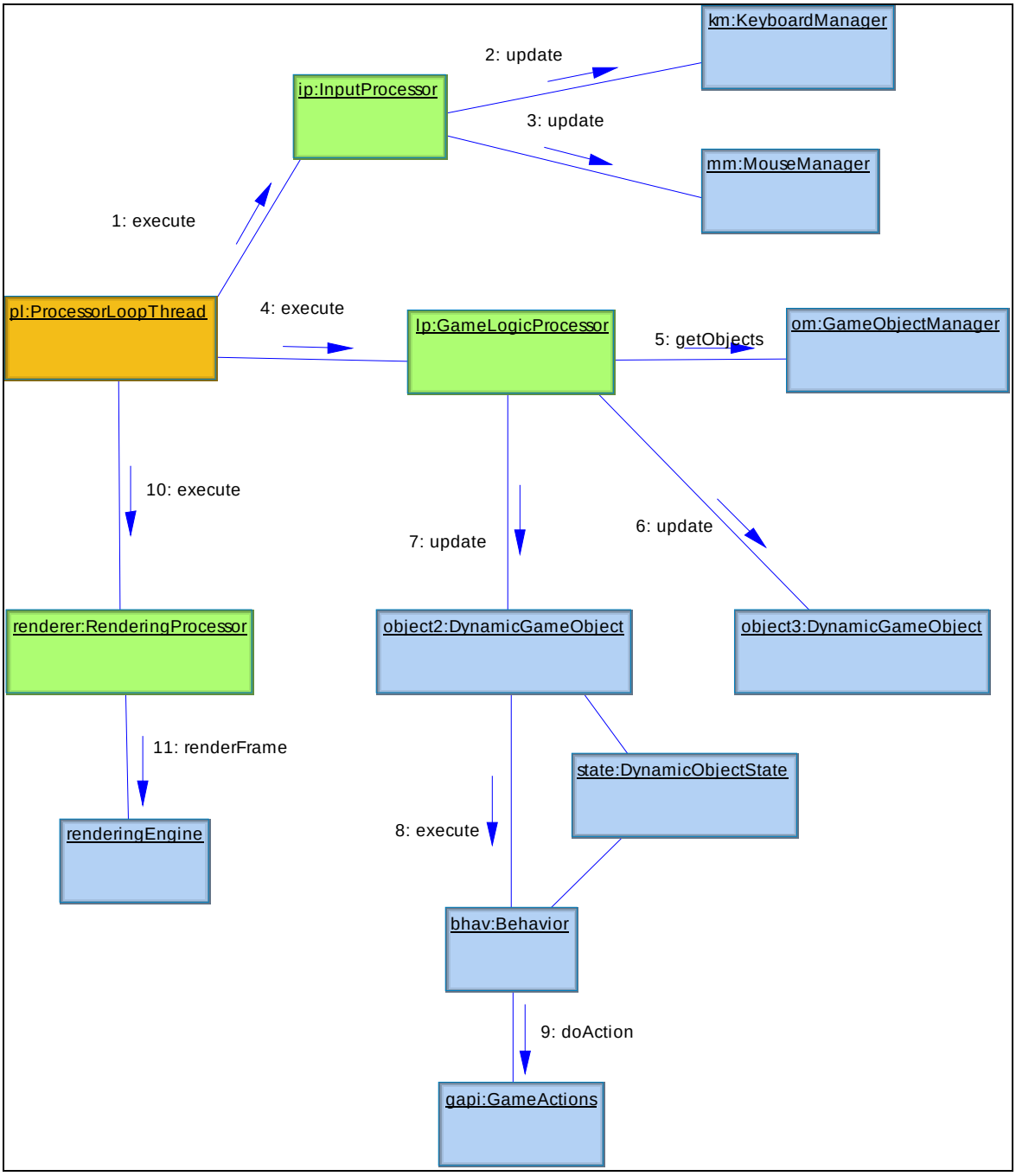

Figura 4.3.5 - Diagrama de comunicación ejemplificando la ejecución de un ciclo de un ProcessorLoopThread.

Capítulo 5 - Conclusiones

5.1 Evaluación de la Arquitectura

Para validar la arquitectura j3dEngine el método utilizado fue la construcción de un prototipo funcional. El desarrollo del prototipo nos conduciría a encontrar las fallas importantes en el diseño y nos permitiría verificar que el diseño es adecuado.

Al principio del proyecto nos fijamos una serie de objetivos que el diseño debería cumplir. Decidimos entonces evaluar los resultados finales que obtuvimos respecto de estos objetivos.

Objetivo	Resultado
Renderización de escenarios 3D en tiempo real.	Gracias a la utilización del SceneGraph de Xith3D hemos logrado visualizar escenarios 3D con animación, texturas y luces.
Es posible desarrollar un engine 3D en Java.	Hemos podido implementar un prototipo que provee la funcionalidad básica de un engine 3D. Además hemos probado otros engines 3D con buena performance, implementados en Java.
Flexible y extensible.	A través de la configuración en formato XML es muy sencillo pluggear un nuevo componente.
Interacción con el usuario.	Ha sido posible lograr esto mediante la utilización de la librería HIAL.
Configurable por el usuario.	El World puede ser configurado desde un XML. Para extender la funcionalidad el usuario debe implementar un nuevo comportamiento en lenguaje Java, pero j3dEngine brinda un API que facilita este proceso. (GameAction API)

Portable	El renderer de j3dEngine utiliza OpenGL como librería gráfica, por lo tanto no debería tener inconvenientes en ser ejecutado en cualquier plataforma que tenga una implementación de ésta.
Java + OpenGL	Hemos utilizado estas herramientas para el desarrollo del prototipo con éxito.
Soporte multithreading	El archivo XML de configuración de ejecución permite paralelizar las tareas del pipeline gráfico especificando las prioridades de cada una.

Figura 5.1.1 Tabla de evaluación de la arquitectura.

Si evaluamos la arquitectura desde el punto de vista de los objetivos planteados, hemos sido exitosos. Durante el desarrollo del prototipo la arquitectura ha sufrido algunos cambios, pero no se vio afectada en su diseño fundamental.

5.2 Objetivos a futuro

Sería bueno que j3dEngine siga siendo desarrollado como proyecto open source.

Los elementos que consideramos pendientes y de interés para el proyecto son:

- Engine de Física: El scenegraph de Xith3D es apto para aplicar cálculos de física.
- Engine de Scripting: La distribución de Java 6 tiene un engine de scripting, este módulo podría ser desarrollado fácilmente aprovechando el módulo GAPI.
- Engine de Inteligencia Artificial: Soporte para máquina de estados y árboles de decisión. Posibilidad de configurarlo mediante xml.
- Múltiples vistas.
- Soporte para mapas.
- Audio / Video.
- Networking.
- Optimizaciones de rendimiento.

Apéndice A – Estadísticas y gráficos

Ventas por unidad

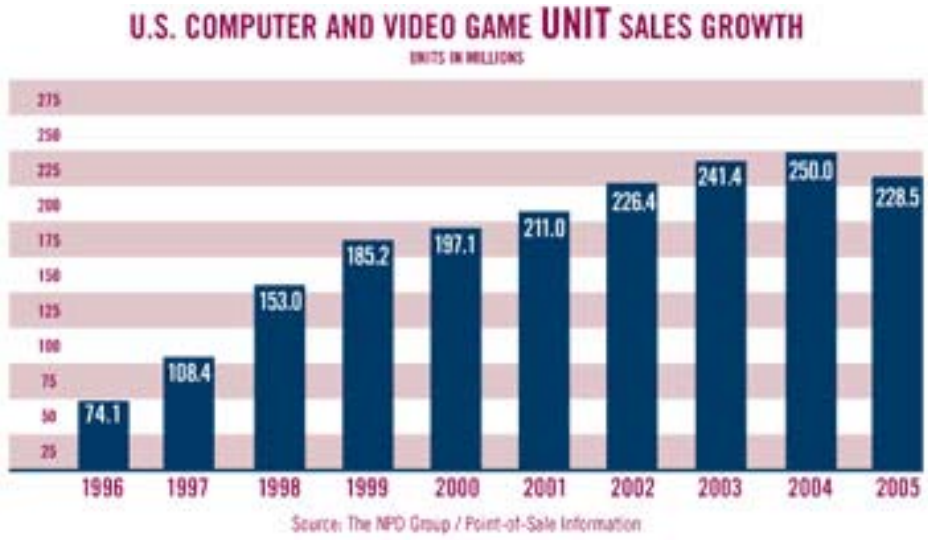

Figura A.1 Ventas de videojuegos en EEUU por unidad. Unidades en millones

Géneros más vendidos en juegos de computadoras

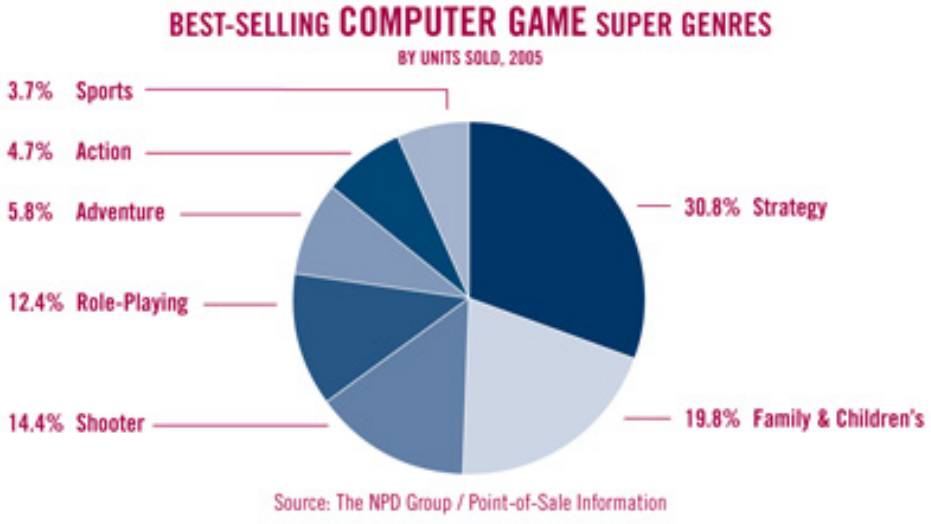

Figura A.2 Géneros de los juegos de computadora más vendidos en EEUU

Géneros más vendidos en juegos de consola

Figura A.3 Géneros de los juegos de consola más vendidos en EEUU

Géneros más jugados online

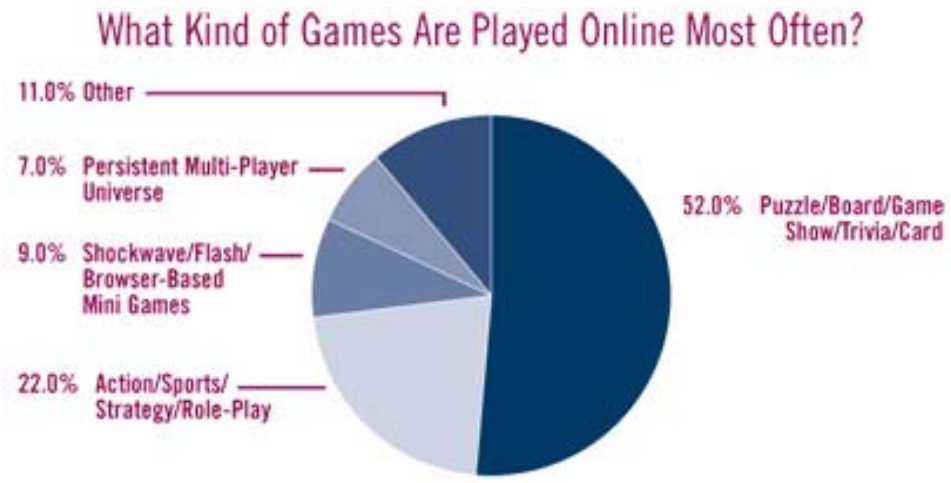

Figura A.4 Géneros de los juegos online mas jugados

Salario promedio anual en el área de programación

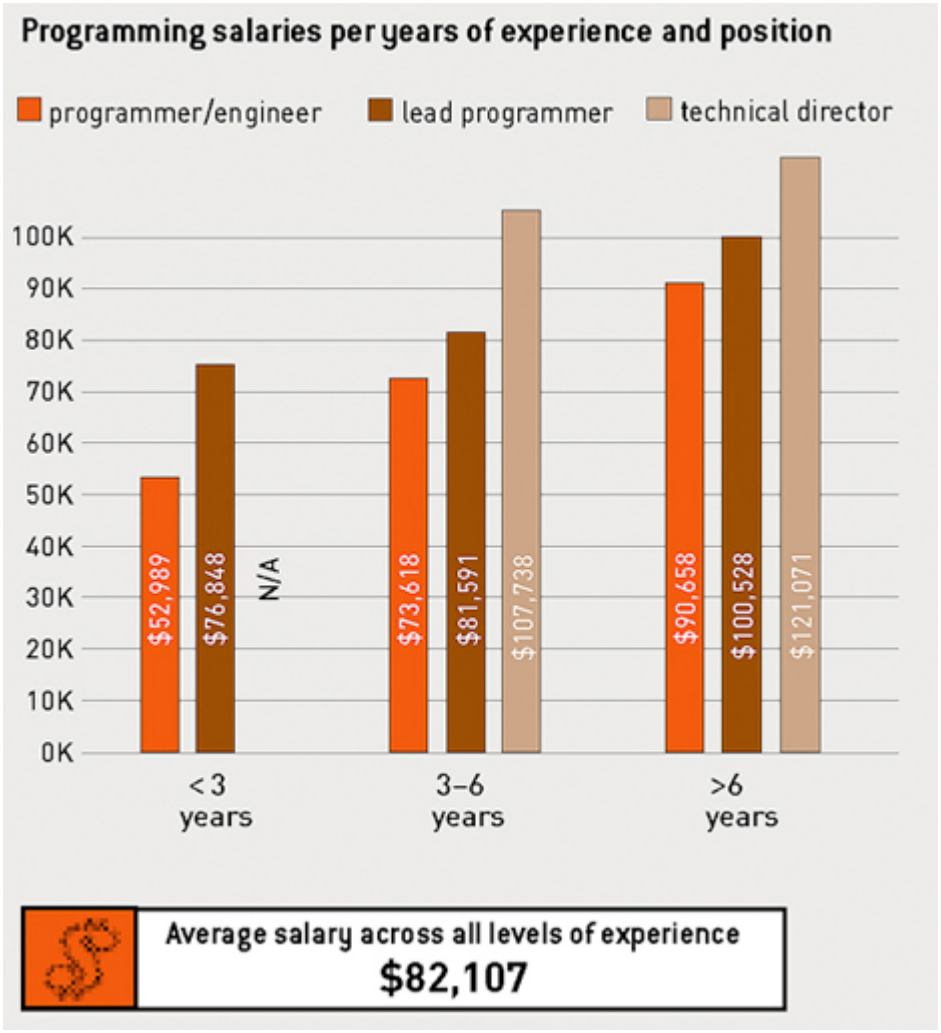

Figura A.5 Salarios promedio en el área de programación de acuerdo a la experiencia

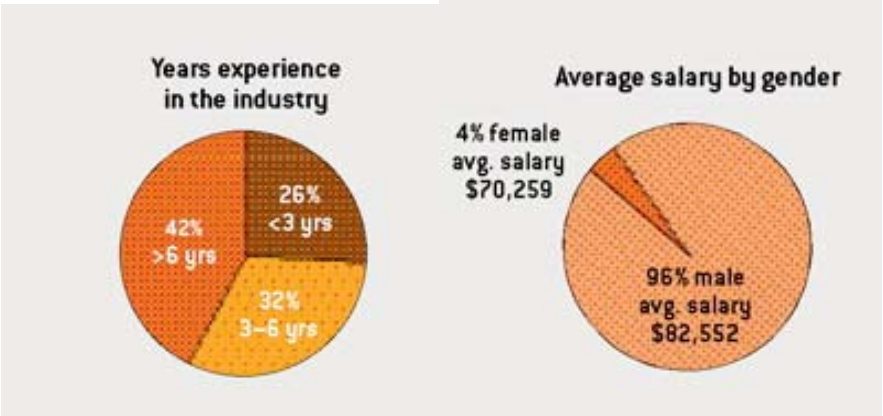

Figura A.6 Distribución en el área de programación de acuerdo a la experiencia y genero

Salario promedio anual en el área de arte y animación

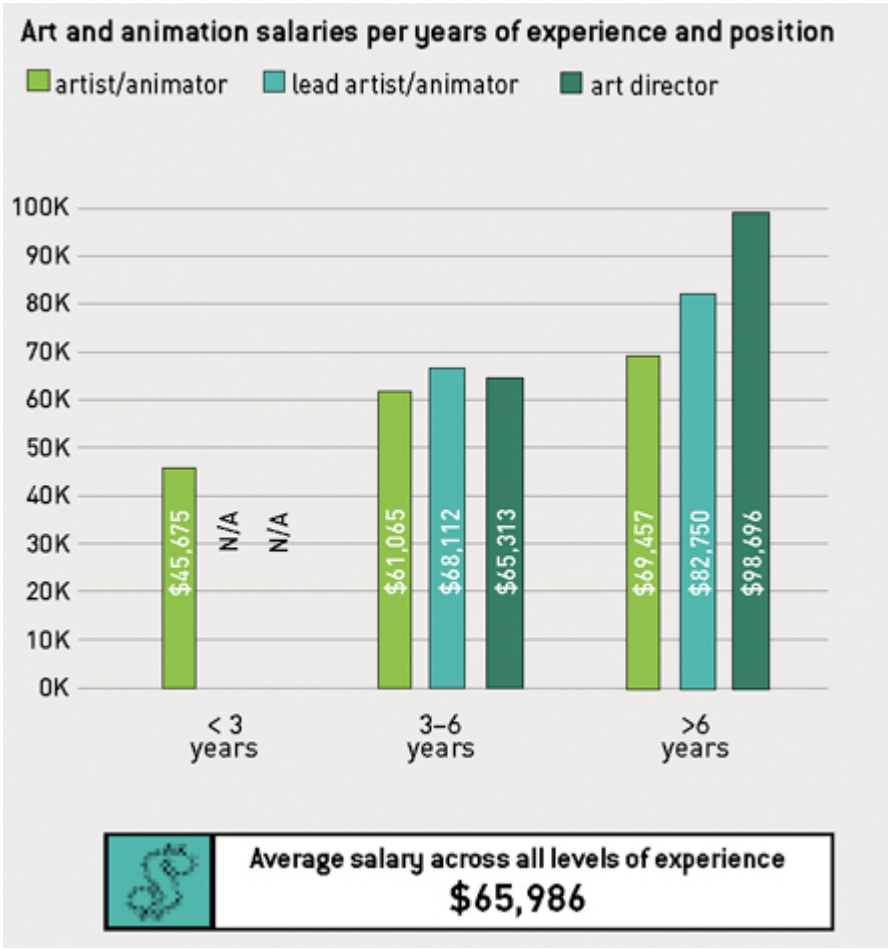

Figura A.7 Salarios promedio en el área de arte y animación de acuerdo a la experiencia

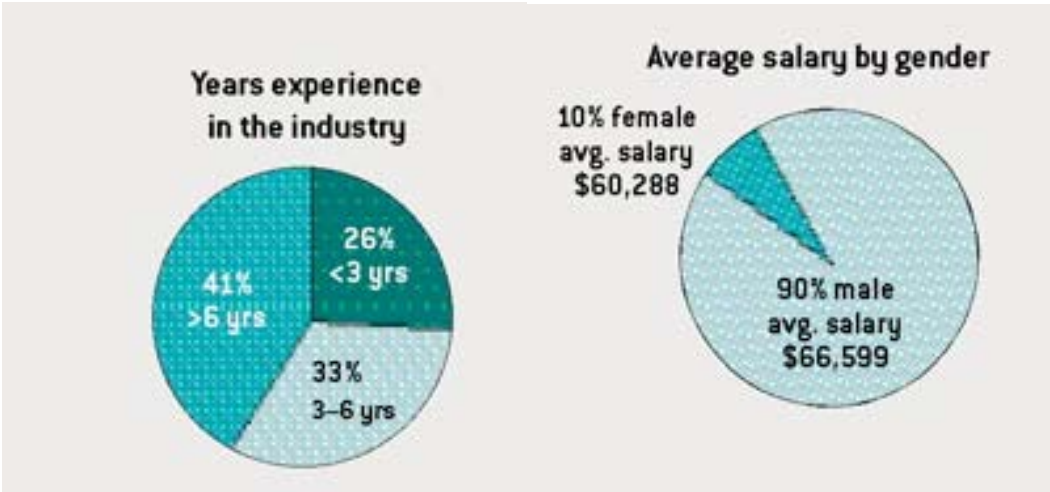

Figura A.8 Distribución en el área de arte y animación de acuerdo a la experiencia y genero

Salario promedio anual en el área de diseño

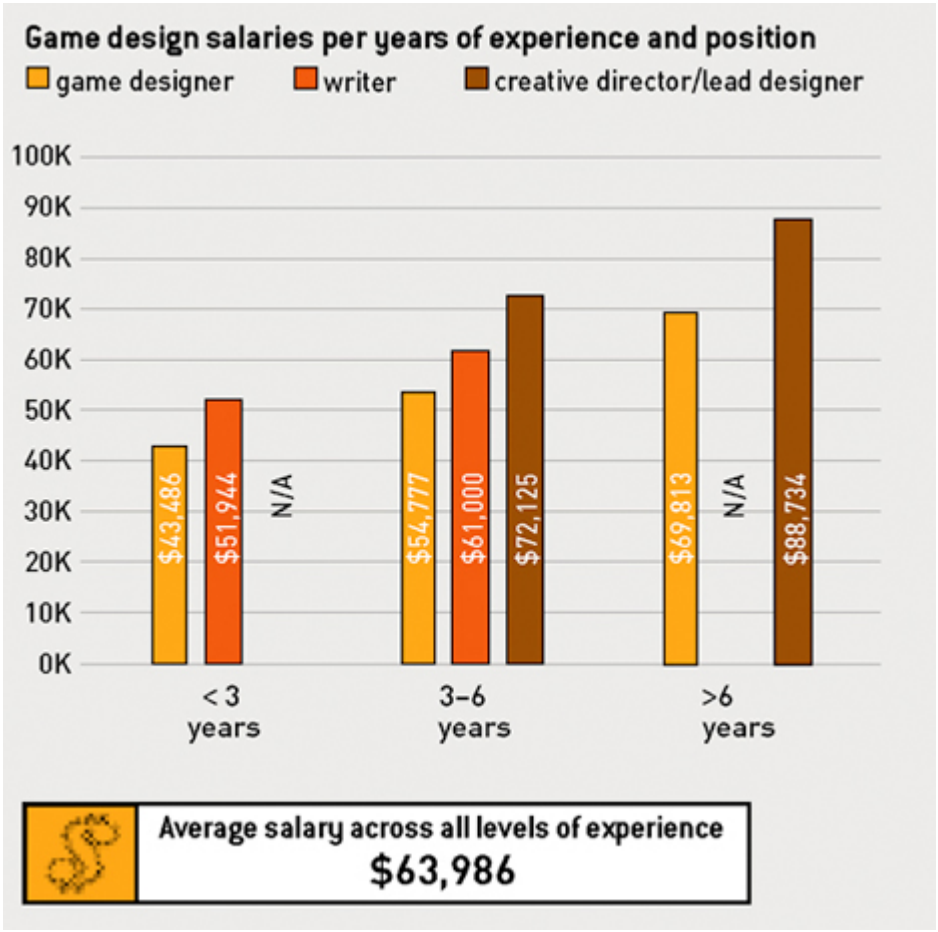

Figura A.9 Salarios promedio en el área de diseño de acuerdo a la experiencia

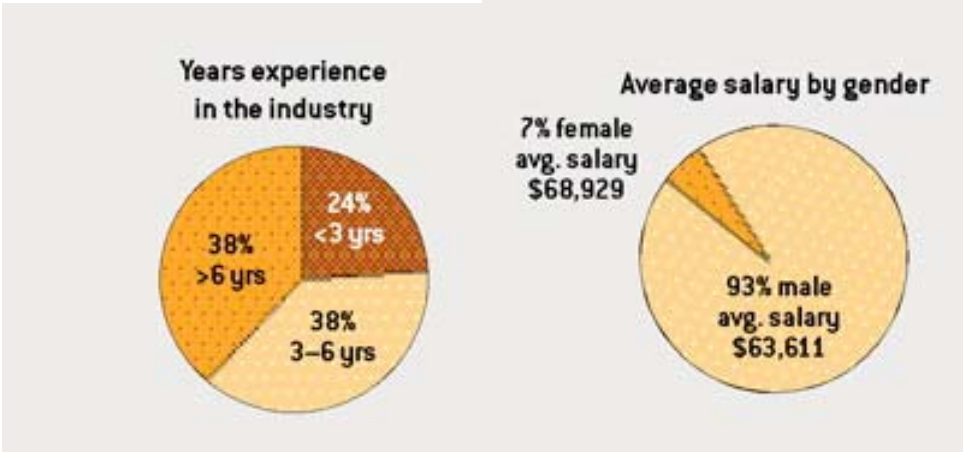

Figura A.10 Distribución en el área de diseño de acuerdo a la experiencia y genero

Salario promedio anual en el área de aseguramiento de la calidad

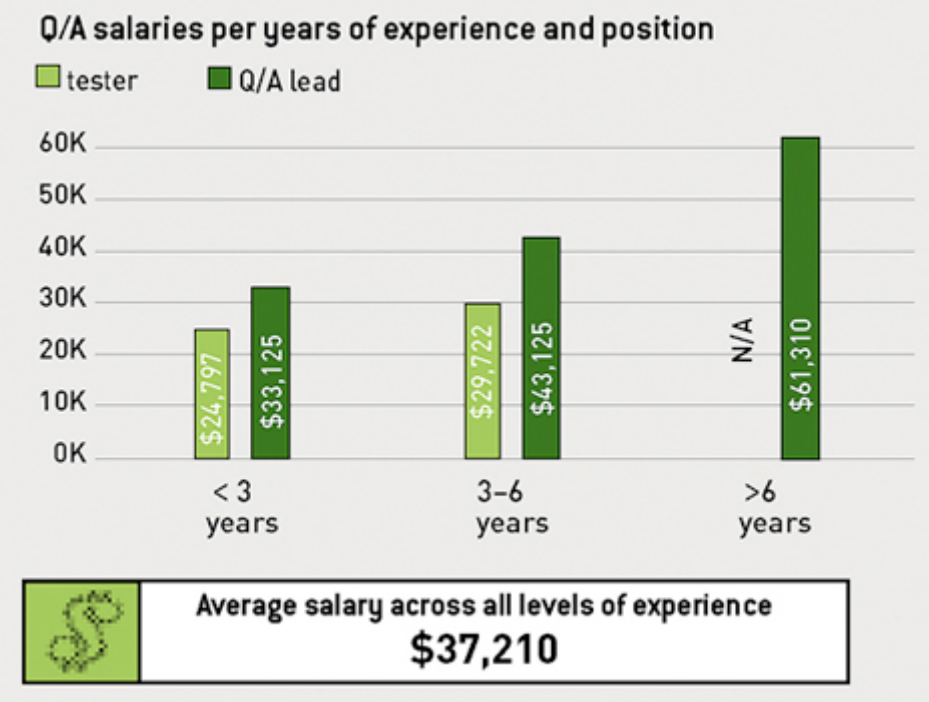

Figura A.11 Salarios promedio en el área de aseguramiento de la calidad de acuerdo a la experiencia

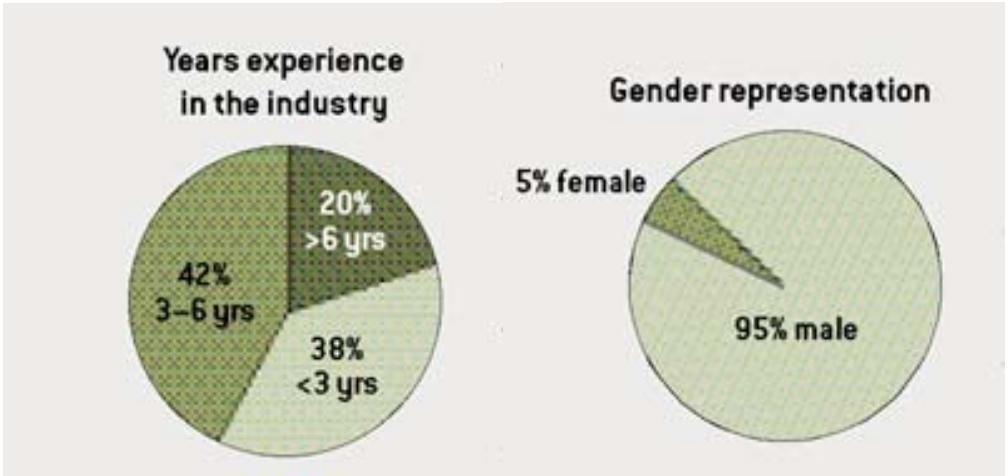

Figura A.12 Distribución en el área de aseguramiento de la calidad de acuerdo a la experiencia y genero

Glosario

Animación: Es el proceso de mostrar rápidamente varias imágenes continuadas a modo de dar la sensación de movimiento. En el caso de la animación por computadora esto se realiza modificando los modelos 3D para que cambien por ejemplo de posición, y se hace tan gradual y en movimientos más pequeños como se desee o pueda de manera de generar más cuadros y que el movimiento resulte más suave.

API (Application Programming Interface): Es una interfaz de código fuente que brinda una librería o un sistema para soportar pedidos de servicio realizados por otros sistemas. Es una definición abstracta, está desvinculada de los detalles de implementación.

Árboles de decisión: Es un modelo predictivo que modela todas las posibles decisiones a tomar dado una condición dada y el estado al que llevará tomar cada una de estas decisiones.

ATI (AMD): Importante fabricante de chips, en especial GPUs. Entre sus marcas más conocidas se encuentran: Rage™, Radeon™, CrossFire™, FireGL™ y FireML™. En octubre del 2006 fue adquirida por U$S 5400 millones por el fabricante de microprocesadores AMD.

BSP (Binary Space Partition): Es un método para subdividir recursivamente un espacio en conjuntos convexos de hiperplanos. Esta subdivisión da origen a una estructura de datos conocida como árboles BSP. Este método se utiliza en computación gráfica para aumentar la eficiencia del rendering evitando renderizar espacios no visibles.

Comportamiento: Los objetos dinámicos de un escenario cambian sus atributos en reacción a eventos generados por otros objetos o por elementos externos al escenario, como input del jugador o eventos temporales. Normalmente tienen alta interacción con el jugador y cambian su posición relativa en el escenario.

CPU (Central Processing Unit): Es el componente en un computador digital que interpreta las instrucciones y procesa los datos contenidos en los programas de computador. Usualmente llamado microprocesador.

Culling o Back face culling: Es una técnica y algoritmos utilizados en computación gráfica para determinar si un polígono de un objeto gráfico es visible o no dependiendo de la posición de la cámara, eliminando del rendering los polígonos no visibles.

Design Pattern: Estructura orientada a objetos que plantea una solución recurrente a problemas comunes en el diseño de software.

Direct3D: Es el API 3D de DirectX, es una librería para el rendering 3D (incluye también algunas operaciones para 2D). Provee a las aplicaciones que la utilizan de un nivel de abstracción respecto del hardware grafico. Su principal competidor es OpenGL.

DirectInput: API para el control de dispositivos de entrada perteneciente a DirectX. Provee un nivel de abstracción respecto del hardware de entrada, como teclado, mouse y joystick.

DirectPlay: API para el manejo de comunicaciones en juegos multiplayer. Pertenece a DirectX.

DirectSound: API de DirectX para el manejo de los dispositivos de audio. Provee una interfaz entre el hardware de audio y el software. Permite también el acceso compartido y simultaneo al hardware de audio.

DirectX: Es una colección de APIs para tareas multimedia creada por Microsoft, especialmente para el uso en videojuegos. Las librerías para su utilización y para desarrollo con ella son de acceso gratuito, pero su código fuente es privado.

Dispositivos de entrada: Es el hardware por medio del cual el usuario ingresa comandos a la computadora o consola. Los dispositivos más comunes son el teclado, el mouse y el joystick.

Engine de física: Un engine de física es un programa de computadora que simula física newtoniana, usando variables tales como la masa, velocidad, rozamiento, dureza, resistencia al viento, etc. Permite simular y predecir efectos que se aproximan a lo que ocurriría en la realidad o en un mundo de fantasía dado, tal como colisiones, caídas, lanzamientos de objetos, etc.

Engine 3D: Un engine 3D o engine gráfico, es un software que se encarga de representar gráficamente en tres dimensiones, objetos modelados utilizando algún formato de información digital. Estos motores suelen utilizar a otros programas tales como los engines de física para que el mundo virtual que grafican simule realismo.

Escenario: Es un ambiente virtual compuesto por objetos con ciertos atributos (geometría, textura, iluminación, comportamiento) y relacionados entre sí (lógica y espacialmente). Lo normal es modelarlo utilizando una estructura de datos conocida como scene-graph.

FPS (First Person Shooter): Se denomina así a los juegos en la cual la cámara del jugador es en primera persona. Y en el cual el jugador porta un arma o similar en sus manos y por lo general el objetivo implica eliminar personajes enemigos o criaturas.

FPS (Frames per second): Cuadros por segundo. Es la cantidad de cuadros que la computadora es capaz de renderizar por segundo. Esta velocidad depende de la calidad del engine3D y del hardware, principalmente del GPU.

Game engine: Es un software que además de ser un engine 3D incorpora, inteligencia artificial, audio / video, scripting, redes y otros factores que hacen a un videojuego además de la representación grafica, ya sea 3D o 2D.

GPU (Graphics Processing Unit): Una GPU es una CPU dedicada exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la CPU puede dedicarse a otro tipo de cálculos (como la inteligencia artificial o los cálculos mecánicos en el caso de los videojuegos). Una GPU implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico.

GPGPU (General-Purpose Computing on Graphics Processing Units): Procesamiento con fines generales en unidades de procesamiento gráfico (GPU). Es una tendencia reciente en computación que utiliza al GPU para realizar cálculos en lugar del CPU. El agregado de etapas programables y aritmética de alta precisión al rendering pipeline permitió a los desarrolladores de software utilizar al GPU para tareas que no se relacionan con gráficos.

Inteligencia Artificial: La inteligencia artificial en los videojuegos se utiliza para crear la ilusión de inteligencia en el comportamiento de los NPCs. Utiliza normalmente técnicas de la rama académica de la inteligencia artificial. Sin embargo la inteligencia artificial en los videojuegos incluye técnicas diferentes, tales como reducir las habilidades de la computadora para que los jugadores humanos tengan una oportunidad contra esta.

MMOG: Massive Multiplayer Online Game, juego online masivo. Son un tipo de juego multiplayer online que se caracteriza por ser jugado por cientos de jugadores a la vez en un server central. El género más común de estos juegos son los de rol (MMORPGs)

MMORPG: Massive Multiplayer Online Role Playing Game, juego de rol online masivo. Es un tipo de juego online masivo que pertenece a la categoría de juego de rol. Los MMORPGs más conocidos son: World of Warcraft, Guildwars, Lineage, Everquest y Ragnarok Online.

Modelo: Es una representación matemática de un objeto 3D almacenada en un archivo de computadora el cual describe las coordenadas de sus vertices, colores, texturas y otras características, para poder ser luego graficado por un engine 3D.

Networking: Es la disciplina de la ingeniería a la cual concierne la comunicación entre sistemas de computadoras. En este contexto, nos referimos específicamente a la comunicación entre computadoras con el objetivo de establecer una interacción multiplayer.

Nivel: Es una etapa o escenario separado en el mundo virtual de un videojuego. Como por ejemplo, una ciudad o un edificio. Este concepto se suele utilizar para fraccionar hasta donde el game engine debe procesar, y en caso de que el jugador exceda los límites de ese nivel, deberá cargarse el próximo y descargarse el anterior.

NPC (Non Player Character): Son los personajes del videojuego que son controlados por la computadora según un comportamiento prefijado o simulado.

Nvidia: Importante fabricante de microchips, utilizados en motherboards, placas de video y otros dispositivos multimedia. Junto con Ati/AMD es uno de los fabricantes de GPUs dedicadas más grandes existentes. Entre sus marcas más conocidas se encuentran: TNT™, GeForce™, Quadro™ y nForce™.

Middleware: El middleware son los programas de computadora cuyo objetivo es conectar otros componentes de software entre sí, como ejemplo software de bajo nivel con software de alto nivel. Al estar entre medio de estas capas, cumplen la función de brindar un nivel de abstracción entre ambas. Otra función importante es la de reducir esfuerzo y costos de implementación, al brindar una solución ya hecha para problemas comunes la industria del software.

Modelo 3D: Representación en memoria o un archivo de un objeto tridimensional. Esta representación se logra descomponiendo al objeto en polígonos y almacenando las posiciones es cada uno de los vértices de estos, sus normales y la textura a aplicar en cada uno de estos.

Módulo: Una porción de un programa que lleva a cabo una o un conjunto de funciones específica. Puede ser utilizado solo o combinado con otros módulos.

Objetos dinámicos y estáticos: Un escenario está formado por objetos de forma recursiva, o sea, los objetos se componen de otros objetos en una relación espacial (de ubicación). Hay objetos que están fijos en su ubicación en el escenario, estos se llaman estáticos. Hay otros objetos que poseen comportamiento y por lo tanto los denominamos dinámicos.

OpenGL (Open Graphics Library): Especificación estándar de un API multi-lenguaje y multi-plataforma para la programación de aplicaciones que producen gráficos 2D y 3D. La interfaz consiste en más de 250 funciones primitivas que se utilizan para dibujar complejos escenarios 3D. Fue desarrollado por Silicon Graphics y es popular en la industria de video games, donde compite con Direct3D de Microsoft.

Out of the box: Sacado de la caja. Se denomina así a los productos que pueden ser puestos en utilización inmediatamente después de haber sido adquiridos, sin necesidad de personalización o configuración alguna.

Pathfinding: Son unos algoritmos utilizados para encontrar la mejor ruta entre dos puntos. En los videojuegos la inteligencia artificial los utiliza para rodear obstáculos principalmente.

Portales: Es una técnica utilizada en computación gráfica para simplificar el renderizado de escenarios basándose en la visibilidad. El método consiste en dividir el escenario en sectores los cuales se conectan el uno con el otro a través de portales, desde la posición donde se encuentre la cámara solo es visible una fracción del sector adyacente, dependiendo del tamaño y posición del portal. De esta manera no es necesario renderizar todo el segundo sector, sino solo la fracción que es visible.

PPU: Physics Processing Unit. Es un procesador dedicado y diseñado enteramente para la realización de cálculos de física newtoniana.

Renderer: Módulo del engine 3D cuya tarea es generar una imagen 3D a partir de un modelo almacenado en memoria o un archivo.

Redes neurales: Es un modelo de inteligencia artificial basado en el funcionamiento del cerebro. Utiliza una red interconectada de neuronas artificiales que utilizan un modelo matemático o computacional para procesar la información. En la mayoría de los casos las redes neurales son un sistema adaptativo que modifica su estructura basado en la información externa o interna que fluye a través de la red.

Shader: Son pequeños programas que utilizan un set de instrucciones de las GPUs, su utilización primaria es para realizar efectos de rendering. Este código es ejecutado directamente por el GPU.

Shader (Pixel shader): Es un tipo de programa shader que sirve para manipular un pixel, usualmente para aplicar un efecto a una imagen. Por ejemplo, realismo, sombras o efectos de explosiones.

Shader (Vertex shader): Es un tipo de programa shader que sirve para manipular la geometría de un objeto de una escena. De esta manera, la forma de un objeto cambia y esto cambia su apariencia. Con la ayuda de efectos de vexter shader tales como el vertex lighting o iluminación por vértice se pueden lograr efectos de ondas u olas por ejemplo.

Scene Graph: Un scene-graph es una estructura de datos orientada a objetos que representa la relación lógica y espacial de la representación de un escenario gráfico. Sirve para representar el modelo de un escenario y para facilitar el procesamiento sobre el mismo.

Scripting: Un lenguaje de scripting es un lenguaje de programación típicamente interpretado, o sea, no se convierte permanentemente a un binario ejecutable, sino que se mantiene en su forma original y se interpreta en el mismo momento que se ejecuta. Los 3D engines tienen en general un módulo de scripting para permitir a los diseñadores asignar comportamiento a los objetos dinámicos de los escenarios con los que el usuario interactúa.

Texel (Texture Element): Unidad fundamental de una textura. Las texturas son representadas por conjuntos de texels, los cuales son mapeados durante el rendering a píxeles en la imagen final.

Textura: Imagen 2D mapeable a una superficie 3D. En general se trata de archivos bitmap en los que cada elemento representa un color, especificando sus componentes RGB (rojo, verde, azul) y un componente de transparencia (alpha).

Referencias

Libros

- [**3DGP**] Finney, *3D Game Programming (All in One)*, Premier Press, 2004
- [**3DGEP**] Zerbst, Duvel, *3D Game Engine Programming*, Premier Press, 2004
- [**3DGEA**] Eberly, *3D Game Engine Architecture*, Morgan Kauffman, 2005
- [**KGPJ**] Davison, *Killer Game Programming in Java*, O'Reilly, 2005
- [**DGJ**] David Brackeen, Bret Barker, Laurence Vanhelsuwé, *Developing Games in Java*, New Riders Publishing, 2003
- [**3DMP**] Dunn, Parberry, *3D Math Primer*, Wordware Publishing, 2002
- [**RTR**] Dempski, *Realtime Rendering in DirectX*, Premier Press, 2002

Artículos y Papers

- [**DJ3D**] Gonzales Clua, Bittencourt, *Desenvolvimiento de Jogos 3D*, PUC Rio
- [**JPT**] Plummer, *A Flexible And Expandable Architecture for Computer Games*, Arizona State University, 2004
- [**X3DN**] Frohlich, *Xith3D in a Nutshell 1st Edition,* 2004
- [**X3DTS**] Frohlich, *Building up a SceneGraph in Xith3D*, 2004
- [**J3DT**] *Java 3D Tutorial*, Sun Mirosystems Inc., 2000
- [**FOSL**] Free And Open Source Licensing White Paper, Sun Microsystems Inc., 2006

Links

- Wikipedia (http://www.wikipedia.org/)
- Xith3D (http://www.xith.org/)
- HIAL (http://input.jtank.net/)
- Java Monkey Engine (http://www.jmonkeyengine.com/)
- Jogl (https://jogl.dev.java.net/)
- JAXB (http://java.sun.com/webservices/jaxb/)
- Game Programming Wiki (http://gpwiki.org/)
- Java Gamming (http://www.javagaming.org/forums/index.php)
- Game middleware (http://www.gamemiddleware.org/middleware/index.html)
- Concurrency and multithreading articles (http://www.jmonkeyengine.com/wiki/doku.php?id=concurrency_and_multithreading_in_game_design)
- Game engine list (http://www.devmaster.net/engines/)