UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Arquitectura Básica para el Desarrollo de Aplicaciones Gráficas 3D

TESIS

PARA OBTENER EL TITULO DE:

Ingeniero en Computación

PRESENTAN

ERIK LÓPEZ PORTILLO BARROSO

JOSÉ GABRIEL ESQUEDA MÁRQUEZ

DIRECTOR: M.I. RODRIGO GUILLERMO TINTOR PÉREZ

CIUDAD UNIVERSITARIA, MÉXICO D.F., 19 DE AGOSTO DE 2011

Me gustaría dedicar este trabajo de tesis a mi familia, porque siempre han estado presentes con su apoyo incondicional, brindándome siempre la fortaleza para salir adelante ante los obstáculos que se han presentado en mi vida.
Gracias a todos porque sin su ayuda no sería la persona en que me he convertido y no hubiera tenido la oportunidad de haber alcanzado esta meta.

José Gabriel Esqueda Márquez

Agradezco a mi familia por el apoyo y confianza que siempre me dieron para llegar a ser la persona que ahora escribe éstas palabras.

Este trabajo está dedicado a todos ustedes.

Gracias, en verdad.

Erik López Portillo Barroso

CONTENIDO

1. Introducción

1.1 Definición del problema

A lo largo de la carrera de Ingeniería en Computación se estudian una gran variedad de temas, siendo los que más llamaron nuestra atención los relacionados a la graficación por computadora. Al comprender estos temas es posible vislumbrar los alcances que pueden tener las aplicaciones de hoy en día, particularmente aquellas que sumergen a los usuarios en ambientes virtuales, además de que nos facilita el poder diseñar y desarrollar una arquitectura para la creación de aplicaciones gráficas.

El desarrollo de una aplicación con este tipo de características es relevante desde varios puntos de vista. En primer lugar el proceso descrito en este trabajo podrá servir como una pauta para aquellos compañeros que tengan el interés en conocer y entender el funcionamiento base de las aplicaciones existentes en el mercado, brindándoles una idea general del funcionamiento de estas aplicaciones, los aspectos que hay que tener en cuenta al momento de iniciar el proceso de diseño y sobre todo, la posibilidad de mejorar el trabajo que nosotros hemos llevado a cabo, por otra parte el resultado de este trabajo podrá contribuir como base para el desarrollo de otros proyectos basados en ambientes virtuales 3D.

1.2 Objetivo

El objetivo de este trabajo de tesis es diseñar e implementar una arquitectura de software bajo la cual se puedan crear aplicaciones gráficas como videojuegos en 3D, recorridos virtuales, entre otras y que éstas cuenten con las características elementales que provee cualquier otra arquitectura comercial como lo son la detección de colisiones, cargadores de modelos geométricos, dibujado en tiempo real e interfaces de usuario.

1.3 Resultado Esperado

El resultado final es una aplicación gráfica 3D demostrativa, la cual será compatible con el sistema operativo Windows y requerirá una tarjeta de video con soporte de OpenGL 2.0 o superior y que estará basada en la arquitectura básica que se ha propuesto en este trabajo escrito.

2. Tecnología de Soporte para la Arquitectura

2.1 Lenguaje C++

C++ es un lenguaje de programación de propósito general desarrollado por Bjarne Stroustrup en la década de los 80's. Las características más importantes que ofrece este lenguaje son:

- Soporte para la programación orientada a objetos.
- Portabilidad.
- Brevedad.
- Programación modular.
- Compatibilidad con el lenguaje de programación C.
- Velocidad.

De acuerdo a Stroustrop (1997, p. 7), el criterio de diseño más importante de C++ fue la simplicidad pero también lo es que haya sido desarrollado teniendo como base C lo cual permite que exista una correspondencia muy cercana entre sus tipos de datos, operadores, sentencias y la forma en que la computadora trata los números, caracteres y direcciones.

Adicionalmente a las características antes mencionadas brinda a los programadores otra gran ventaja, y es que cuenta con una decena de implementaciones independientes, centenares de bibliotecas, libros de texto y manuales brindando un gran soporte a la comunidad de desarrolladores.

Es importante recalcar que C++ no fue diseñado para el cómputo numérico aunque hoy en día es una opción muy popular para el desarrollo de aplicaciones en las que el manejo de gráficos y las interfaces de usuario son una parte importante, por lo que cuenta con un gran respaldo en este ámbito y es una de las principales ventajas por las que se optó usar este lenguaje de programación.

2.2 STL de C++

La Standard Template Library (STL) es una colección de estructuras de datos contenedoras[1], algoritmos e iteradores que están escritos en C++. Cabe señalar que la STL no es la primera implementación de este tipo, ya que anteriormente la mayor parte de los compiladores disponían de bibliotecas similares o estaban disponibles varías bibliotecas comerciales, aunque su principal desventaja es que no eran compatibles entre sí, lo que suponía para los programadores la obligación de aprender las nuevas bibliotecas para migrar de un proyecto a otro o bien de uno a otro compilador.

Sin embargo, la STL fue adoptada por el comité ANSI de estandarización de C++, lo que significó que actualmente todos los compiladores lo tuvieran como una extensión más del lenguaje.

La STL provee algunas de los tipos de contenedores más generales y útiles permitiendo al programador seleccionar una estructura que mejor se ajuste a las necesidades de su aplicación con la certeza de que cualquiera de ellas está respaldada por una gran flexibilidad, eficiencia y bases teóricas.

[1] Una estructura de datos se dice que es contenedora si puede contener instancias de otras estructuras de datos.

Resumen de los contenedores estándar	
vector<T>	Un vector de tamaño variable.
list<T>	Una lista doblemente enlazada.
queue<T>	Una cola.
stack<T>	Una pila.
deque<T>	Una cola de doble punta.
priority_queue<T>	Una cola ordenada por valor.
set<T>	Un conjunto,
multiset<T>	Un conjunto en el cual un valor puede repetirse varias veces.
map<key,val>	Un arreglo asociativo.
multimap<key,val>	Un mapa en el cual una llave puede repetirse varias veces.

Figura 1. Contenedores estándar más importantes que ofrece la STL.

De acuerdo a las características de los contenedores estos se pueden dividir en tres categorías:

- Contenedores lineales. Son aquellos que almacenan los objetos de forma secuencial, permitiendo además el acceso a los mismos de forma secuencia y/o aleatoria.
- Contenedores asociativos. Son aquellos que almacenan los objetos asociándolos a una clave. Su principal ventaja es la rapidez con la que se pueden almacenar o recuperar los objetos del contenedor.
- Contenedores adaptados. Permiten cambiar un contenedor en un nuevo contenedor modificando la interface del primero.

Al hacer uso de la STL se obtienen varias ventajas, de entre las cuales podemos mencionar las siguientes:

- Al ser estándar, está disponible para todos los compiladores y plataformas, lo cual permite usar la misma biblioteca en todos los proyectos.

- Al ser una biblioteca de componentes reutilizables incrementa la productividad y reduce el tiempo de desarrollo.
- El desarrollo de las aplicaciones se hacen rápidamente ya que se construyen a partir de algoritmos eficientes.
- Proporciona su propia gestión de memoria, de tal forma que el programador podrá ignorar problemas tales como las limitaciones de la memoria del PC.

2.3 Interfaz de Programación de Aplicaciones OpenGL

OpenGL es un ambiente para desarrollo portable de aplicaciones gráficas interactivas 2D y 3D. Fue introducido en 1992 y desde entonces se ha convertido en la interfaz de programación de aplicaciones o API (por sus siglas en inglés) más utilizada a nivel mundial. OpenGL promueve la innovación y el rápido desarrollo de aplicaciones incorporando una amplia colección de herramientas de renderizado, mapeo de texturas, efectos especiales y muchas más funciones de visualización, por lo que es una API gráfica ideal sobre la cual basar nuestra arquitectura.

Rendimiento y Calidad Visual

Cualquier aplicación gráfica que requiera un rendimiento superior, de animación 3D a CAD[2] o hasta simulación visual puede explotar las capacidades de alta calidad que ofrece OpenGL. Estas capacidades permiten a los desarrolladores de diversas áreas como CAD/CAM[3]/CAE[4], entretenimiento, visualización médica y realidad virtual, producir y desplegar gráficos 2D y 3D convincentes.

[2] Computer-Aided Design que se refiere al diseño asistido por computadora.
[3] Computer-Aided Manufacturing que se refiere a la manufactura de productos asistida por computadora.
[4] Computer-Aided Engineering que se refiere a la ingeniería asistida por computadora.

Ventajas para el desarrollador

a. Estándar en la industria

La junta de revisión de la arquitectura (ARB por sus siglas en inglés) de OpenGL, es la encargada de llevar su especificación además de que es el único estándar gráfico multiplataforma y abierto.

b. Estable

Las implementaciones de OpenGL han estado disponibles por más de 7 años en una gran variedad de plataformas. Adicionalmente, la especificación lleva un buen control que permite anunciar actualizaciones en tiempo para que los desarrolladores adopten estos cambios. La retro compatibilidad asegura que las aplicaciones existentes no se vuelvan obsoletas.

c. Confiable y Portable

Todas las aplicaciones producen resultados consistentes para cualquier hardware compatible con el API de OpenGL, sin importar el sistema operativo o el sistema de ventanas.

d. Constante evolución

Debido a su diseño cuidadoso y vanguardista, OpenGL permite que la innovación en hardware sea accesible a través del mecanismo de extensiones de su API. De esta forma, las innovaciones aparecen de manera oportuna permitiendo a los desarrolladores y a los vendedores de hardware incorporar las nuevas características en sus ciclos normales de publicación.

e. Escalable

Las aplicaciones basadas en el API de OpenGL se ejecutan en sistemas desde PCs, estaciones de trabajo hasta supercomputadoras. Como resultado, las aplicaciones son escalables hacia cualquier tipo de máquina que elija el desarrollador.

f. Facilidad de uso

OpenGL está bien estructurado con un diseño intuitivo y comandos lógicos. Las rutinas eficientes de OpenGL generalmente resultan en aplicaciones con menos líneas de código que con otras bibliotecas o paquetes gráficos. Adicionalmente, los controladores de OpenGL encapsulan la información acerca del hardware subyacente liberando al desarrollador de tener la necesidad de diseñar para un hardware específico.

g. Documentado

Existe un gran número de libros publicados acerca de OpenGL junto con muchos ejemplos de código disponibles haciendo que la información acerca de OpenGL no sea costosa y sea fácil de obtener.

Pipeline o tubería de programación de OpenGL

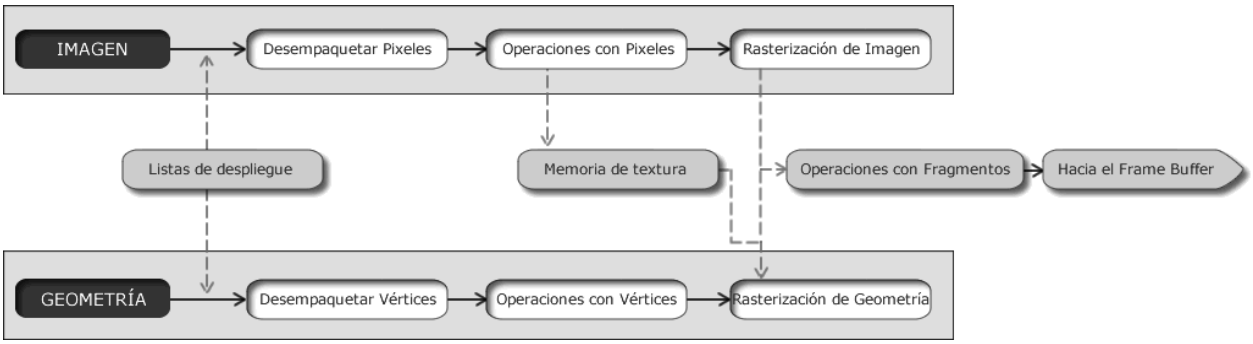

Figura 2. OpenGL opera sobre datos de imagen así como sobre primitivas geométricas

Simplificación de desarrollo de software

Las rutinas de OpenGL simplifican el desarrollo de aplicaciones gráficas, desde dibujar una simple geometría de un punto, línea, polígono relleno, hasta la creación de mapas de iluminación y texturizado más complejos. OpenGL da acceso a los desarrolladores a primitivas de imagen y geometría, listas de despliegue, transformaciones de modelo, iluminación y texturizado, anti-aliasing[5], blending[6] y otras más características.

[5] Anti-aliasing son técnicas que permiten obtener imágenes con bordes suavizados.

El estándar de OpenGL tiene relación con los lenguajes C, C++, Fortran, Ada y Java. Todas las implementaciones licenciadas de OpenGL vienen de una especificación única que es requerida para pasar un conjunto de pruebas de compatibilidad. Las aplicaciones que hacen uso de las funciones de OpenGL son fácilmente portables a un amplio rango de plataformas para maximizar la productividad del programador y reducir el tiempo de comercialización.

Todos los elementos del estado de OpenGL, inclusive los contenidos de memoria de textura y frame buffer, pueden ser obtenidos por una aplicación basada en OpenGL. También soporta aplicaciones de visualización con imágenes 2D tratadas como primitivas que pueden ser manipuladas como cualquier geometría 3D. Como se muestra en la Figura 2, las imágenes y los vértices que definen primitivas geométricas son pasadas a través del pipeline de OpenGL hacia el frame buffer.

La base para APIs avanzadas

Las empresas líderes en desarrollo de software utilizan OpenGL, junto con sus bibliotecas robustas de dibujado, como la base para APIs para gráficos 2D y 3D de alto nivel. Es por tal motivo que muchos de los desarrolladores recurren a las capacidades de OpenGL para entregar sus soluciones al mercado.

[6] El blending consiste en la mezcla de 2 colores para formar uno, permitiendo efectos como transparencia, filtros aditivos y más.

2.4 Shaders con GLSL

Los shaders son programas que permiten calcular efectos de renderizado directamente en el hardware gráfico con un grado alto de flexibilidad. Los shaders sustituyen los estados correspondientes al procesamiento de vértices y/o fragmentos con áreas programables que pueden realizar menor, igual o mayor número de operaciones que la funcionalidad fija. Los shaders fueron diseñados para permitir a los programadores realizar descripciones del comportamiento en estos puntos y por ello es imprescindible hacer uso de ellos para mejorar la calidad visual de los ambientes virtuales que serán dibujados por nuestra arquitectura.

Existen diferentes tipos de shaders:

a. Shaders de vértice (Vertex Shaders)

Se aplican sobre cada vértice que procesa la GPU. Se pueden manipular propiedades como la posición, el color, la coordenada de textura, pero no se pueden crear nuevos vértices.

b. Shaders de geometría (Geometry Shaders)

Permiten añadir o quitar vértices de una malla. Se utilizan para añadir detalle a volúmenes de forma no tan costosa como si se procesara en la CPU.

c. Shaders de pixel (Fragment Shaders)

Procesan el color para cada pixel individual. Generalmente se utilizan para calcular iluminación, tonalidad de color o efectos como bump mapping[7].

[7] Bump mapping es una técnica que permite crear la ilusión de relieves a través del uso de una imagen difusa (textura) y un mapa de normales.

¿Por qué usar shaders?

Los shaders permiten describir efectos de manera más simple y eficientes que la funcionalidad fija de OpenGL, también se pueden obtener mejores resultados e inclusive realizar efectos que serían imposibles con la funcionalidad fija.

Algunos efectos que se pueden obtener:

- Materiales realistas: metal, madera, piedra, etc.
- Iluminación más realista: luz por área, sombras suaves, etc.
- Ambientes naturales: fuego, agua, cielo, nubes, etc.
- Materiales no realistas: lápiz, pluma, carbón, etc.
- Nuevos usos de memoria de textura.
- Procesamiento digital de imágenes
- Animación

GLSL

La reciente tendencia en el hardware de gráficos ha sido remplazar la funcionalidad fija con programación en áreas que han crecido de manera excepcionalmente compleja (por ejemplo procesamiento de vértices y procesamiento de fragmentos). El lenguaje de shaders de OpenGL (GLSL por sus siglas en inglés), fue diseñado para permitir a los programadores modificar el procesamiento que ocurre en ciertos puntos del pipeline de OpenGL.

El lenguaje GLSL está basado en ANSI C y muchas de los rasgos han permanecido a excepción de cuando entran en conflicto con el rendimiento o la facilidad de implementación. El lenguaje C ha sido extendido con tipos de dato como vectores y matrices para hacerlo más conciso con las operaciones típicas que son usada en gráficos 3D. Algunos otros mecanismos del lenguaje C++ fueron tomados, como la

sobrecarga de funciones basada en los tipos de dato de los argumentos, y la habilidad de declarar variables donde sea necesario en lugar de ser al inicio de los bloques de código.

Procesadores Programables

Existen 2 tipos de procesadores programables:

- Vértices
- Fragmentos (pixeles)

Cuando los procesadores están activos se integran al estado de OpenGL e inactivan la funcionalidad fija. Los procesadores programables ejecutan shaders de vértices y fragmentos respectivamente. Fueron diseñados para trabajar con OpenGL por lo que tienen salidas y entradas que convergen en el pipeline estándar.

Las aplicaciones pueden pasar datos a los shaders a través de atributos y variables *uniform* definidas por el usuario. Los shaders de vértices pueden comunicarse con el procesamiento subsecuente utilizando variables de salida especiales y variables *varying* integradas. Los shaders de fragmentos obtienen datos del procesamiento previo a través de variables de entrada especiales y variables *varying* integradas.

a. Procesador de vértices

Es una unidad programable que opera sobre los vértices de entrada y sus valores asociados (un solo vértice a la vez).

Está diseñado para realizar operaciones tradicionales como:

- Transformación de vértices
- Transformación y normalización de normales
- Generación de coordenadas de textura
- Cálculos de iluminación

- Aplicación de color de material

Cuando se habilitan los shaders de vértices la siguiente funcionalidad de OpenGL no se aplica:

- La matriz de vista de modelo no se aplica a los vértices entrantes
- La matriz de proyección no se aplica
- La matriz de texturas no se aplica
- Las normales no son transformadas a coordenadas de vista
- No se generan coordenadas de textura
- No se aplica iluminación por vértice
- No se aplica color de material
- No se aplica color indexado

Por lo tanto el shader de vértices debe implementar las operaciones sustituidas de la funcionalidad fija dependiendo de las necesidades de aplicación.

El shader de vértices no reemplaza la funcionalidad que requiere del conocimiento de más de un vértice:

- La división de perspectiva por coordenadas de corte
- Mapeo al puerto de vista
- Rango de profundidad
- Recorte
- Determinación de cara frontal
- Ajuste de color, textura, normales
- Procesamiento de color final

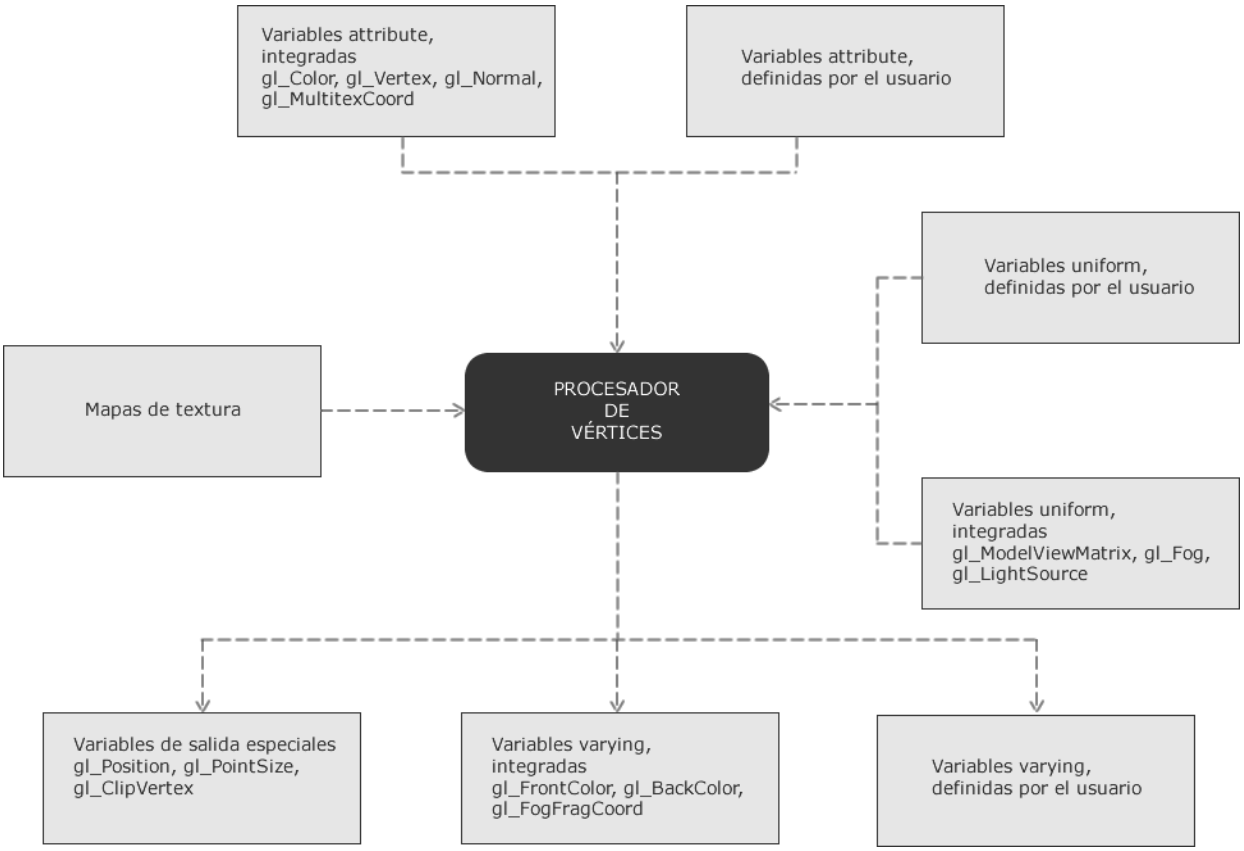

Figura 3. Procesador de vértices

b. Procesador de fragmentos

Es una unidad programable diseñada para operar fragmentos y sus datos asociados. Puede realizar las siguientes operaciones tradicionales:

- Operaciones en los valores interpolados de los vértices
- Acceso a texturas
- Aplicación de texturas
- Aplicación de niebla
- Suma de color

Ejecutan shaders de fragmentos y no pueden cambiar la posición (x, y) de un fragmento.

Cuando se habilitan los shaders de fragmentos la siguiente funcionalidad de OpenGL no se aplica:

- El ambiente y funciones de textura no son aplicados
- No se mapean las texturas
- No se realiza la suma de color
- No se aplica la niebla

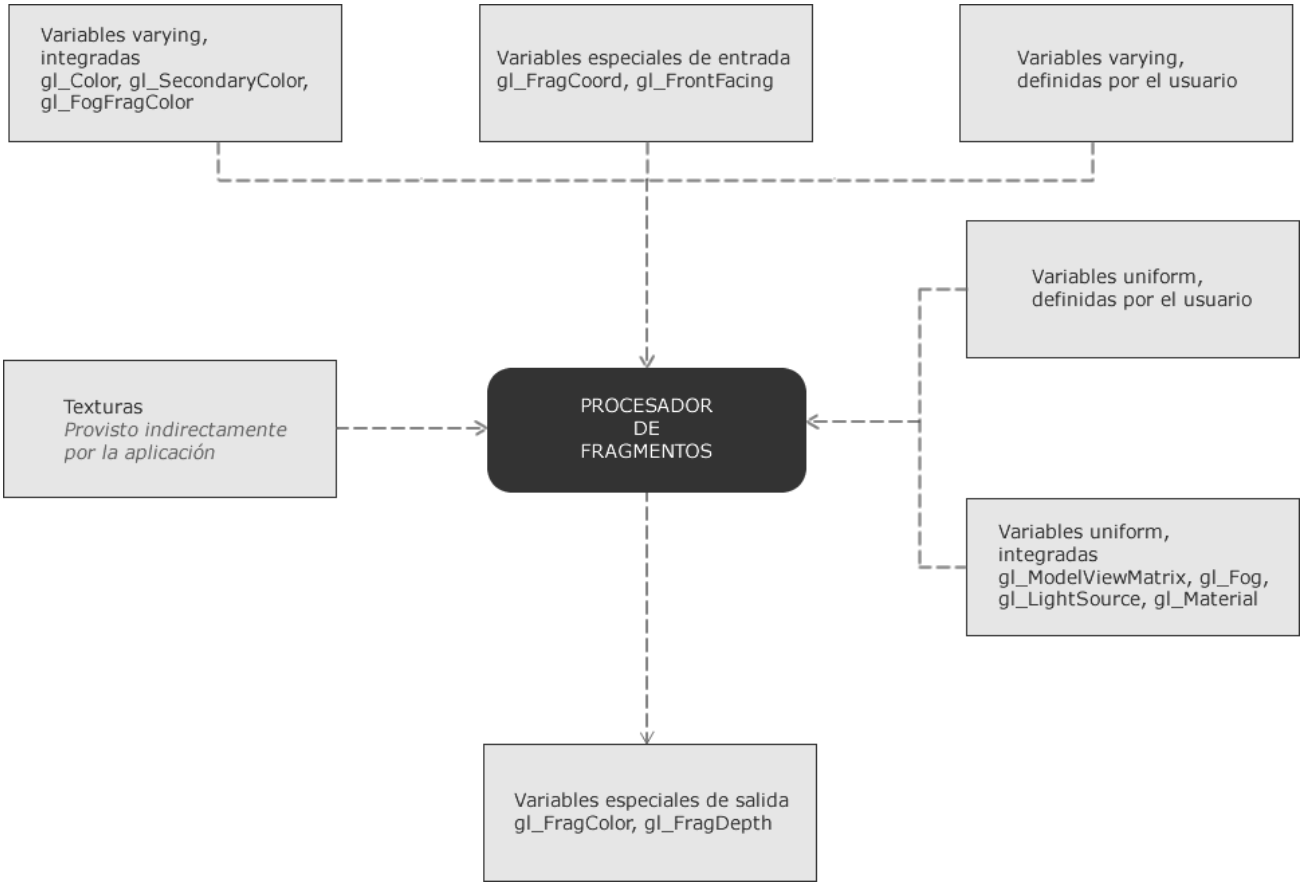

Figura 4. Procesador de fragmentos

2.5 Animación con Cal3D

Cal3D es una biblioteca de animación escrita en C++ con licencia LGPL[8], que es independiente tanto de la API gráfica como de la plataforma donde es utilizada.

Originalmente fue diseñada para ser usada como un cliente 3D del proyecto Worldforge[9] pero finalmente se convirtió en un proyecto independiente que puede ser usado en cualquier otro tipo de proyectos. Al momento en que el proyecto dio inicio, la tecnología más prometedora y flexible para la animación de los personajes era la basada en esqueletos, la cual proporcionaba mucha libertad en el proceso de animación y fue elegida como el elemento central de la biblioteca Cal3D.

De acuerdo a los creadores de la biblioteca, los objetivos contemplados en el diseño de este proyecto fueron:

- Funcionalidad.
- Facilidad de uso.
- Portabilidad.
- Escalabilidad.
- Flexibilidad.
- Rendimiento.
- Interoperabilidad.

Las características más relevantes que fueron consideradas para elegir usar esta biblioteca dentro de nuestro proyecto fueron las siguientes:

[8] En inglés, Lesser General Public License, permite hacer uso del código fuente de manera libre sin necesidad de pagar por él siendo así una licencia de software libre.
[9] Ver http://www.worldforge.org/ para más información

- Provee un sistema de control que permite manipular la secuencia y la mezcla de las animaciones, además de que le permite soportar diferentes tipos de ellas, tales como ciclos, acciones o poses.
- Cuenta con un manejo flexible de los materiales y las mallas lo que permitiría cambiar completamente la apariencia de un modelo.
- El diseño de la biblioteca está orientado a que ésta sea simple, lógica e intuitiva.
- Dado que biblioteca esta codificada en C++ y únicamente depende de la STL, puede ser usada en diferentes plataformas pudiendo utilizar tanto OpenGL como DirectX sin ningún problema, ya que no lleva a cabo el manejo del renderizado y la administración del texturizado por sí misma.
- Debido a que su diseño estuvo contemplado para formar parte de un cliente MMORPG[10] (Massively Multiplayer Online Role-Playing Game) el rendimiento global para manejar varios modelos al mismo tiempo es muy importante, lo cual se logró al optimizar los cálculos dentro la propia biblioteca.

Cal3D puede ser visto como dos piezas: la biblioteca de C++ y el exportador. El exportador como tal nos permite crear los archivos de los modelos elaborados bajo el formato Cal3D, el cual es el único que puede ser cargado por la biblioteca. Actualmente el software de modelado que tiene soporte para el exportador de Cal3D es 3ds Max y Maya. Por otra parte, la biblioteca de Cal3D provee a los desarrolladores la interfaz para la manipulación de las animaciones.

10 Son videojuegos de rol que permiten a miles de jugadores introducirse en un mundo virtual de forma simultánea a través de la Internet e interactuar entre ellos.

2.6 Audio con FMOD Ex

El sistema de sonido FMOD Ex es un motor de audio creado por Firelight Technologies para impulsar los límites creativos de la implementación de audio en videojuegos y similares, usando los mínimos recursos y manteniendo la escalabilidad. FMOD Ex está pensado para el uso de desarrolladores de videojuegos, desarrolladores multimedia, diseñadores de audio, músicos e ingenieros de audio. Es importante mencionar que se utilizó éste motor bajo la licencia no comercial, la cual establece que el producto derivado no estará destinado a fines comerciales y que tampoco se usará la biblioteca de FMOD para su venta o distribución comercial. Es por ello que las características provistas por FMOD Ex es ideal para enriquecer con audio los escenarios desplegados por nuestra arquitectura.

Arquitectura de procesamiento digital de señales (DSP)

La arquitectura para mezclar señales de FMOD utiliza cálculos de punto flotante con interpolaciones completas de 32 bits para proveer la máxima calidad de audio en la suma de señales. Utiliza una arquitectura basada en nodos que permite una flexibilidad de acceso, sub mezcla y salida en canales de elección para el programador. Los canales de entrada pueden ser mapeados para cualquier canal de salida mediante una simple matriz de dos dimensiones.

API de FMOD Ex

Una de las ventajas de este motor de audio es su integración con el lenguaje C++ que permite acceder a su funcionalidad a través de clases (programación orientada a objetos) como la clase del sistema, de sonido, de canal, de DSP y más.

2.7 Manejo de archivos XML con TinyXML

TinyXML es un parser XML escrito originalmente en C++ por Lee Thomason que puede ser fácilmente integrado dentro de otros programas bajo la licencia zlib, la cual permite que sea usado tanto en código abierto como en código lucrativo.

La funcionalidad que TinyXML provee para la manipulación de archivos XML es básica y consiste en parsear un documento XML y construir a partir de él un Modelo de Objetos del Documento (DOM[11]) que puede ser leído, modificado y guardado. Dada la simplicidad y facilidad de uso, TinyXML se adapta a nuestros requerimientos para carga y lectura de archivos de configuración usados por nuestra arquitectura.

Se debe tener en cuenta que este parser tiene ciertas limitaciones, entre las que podemos enumerar:

- No usa ni procesa entidades DTD[12] o XLS[13].
- En términos de codificación, TinyXML da soporte a UTF-8, permitiendo manipular archivos XML en cualquier lenguaje. Además soporta “Legacy Mode”, la codificación usada antes de UTF-8 y que se podría describir mejor como “ASCII extendido”.

La ventaja de usar el formato XML es que su formato está bien estructurado, permitiendo crear cualquier clase de etiquetas en el documento. De esta forma todos aquellos formatos de archivos creados para almacenar datos de una aplicación pueden ser totalmente reemplazados con XML y usar un único parser para todos ellos.

[11] En inglés, Document Object Model
[12] Ver, Document Type Definition
[13] Ver, Extensible Stylesheet Language

Para finalizar, es importante mencionar que no se aplicó ningún tipo de encriptación en los archivos XML, ya que son usados como archivos de configuración para el usuario y no supone ningún riesgo para el funcionamiento de la arquitectura.

3. Desarrollo e Implementación de Tecnologías

3.1 Introducción

Como se mencionó anteriormente, el objetivo es diseñar una arquitectura que permita un desarrollo sencillo de aplicaciones gráficas 3D, orientado principalmente a proyectos tales como recorridos virtuales o pequeños videojuegos.

Antes de comenzar con el diseño es importante definir las limitaciones de las aplicaciones a las que estará orientada ésta arquitectura, no pueden ser genéricas ya que esto implicaría una tarea inmensa de análisis tratando de cubrir todas las características que puede llegar a contener una aplicación gráfica, por tal motivo se plantean las siguientes restricciones bajo las cuales el funcionamiento de la arquitectura será óptimo:

- La cámara podrá usarse únicamente en primera y tercera persona.
- Los escenarios deberán ser preferentemente interiores para proveer un mejor manejo de las escenas, aunque esta restricción no es obligatoria.
- Los modelos geométricos deberán estar optimizados para tener el menor número de vértices posibles.

A continuación comenzaremos a detallar cada uno de los módulos que se consideraron como indispensables para conformar la arquitectura. Estos son:

- *Sistema de eventos con volúmenes especiales:* Este sistema se encargará de accionar tareas previamente programadas cuando exista una colisión con un volumen especial. Las tareas que se pueden ejecutar van desde traslaciones, rotaciones, escalamientos hasta habilitar luces o dibujar nuevos modelos geométricos en la escena.
- *Texturizado avanzado:* Ésta característica permite mapear texturas sobre cualquier modelo geométrico cargado en la escena logrando un mayor realismo de nuestro mundo virtual.

- *Efectos de texturizado con Normal Mapping y Light Maps:* Con éstas técnicas se pretende dar mayor realismo a los objetos visualizados por el usuario. Normal mapping es una técnica que permitirá aparentar un detalle más elevado de los objetos planos sin incrementar la cantidad de vértices que se dibujarán en la escena en tiempo real. Por otra parte, el uso de Light Maps permite simular la existencia de luces sin necesidad de calcular algoritmos para iluminación de los objetos en escena.
- *Efectos de Post-processing:* Además de las técnicas anteriores, estos efectos permiten procesar imágenes en tiempo real para añadir efectos tales como bloom, blur y otros de los que se hablará más adelante.
- *Sistema de colisiones:* Este sistema se encargará de que los objetos que están en escena no puedan ser atravesados por otros objetos como sucede en el mundo real.
- *Sistema de partículas:* Las partículas permiten añadir efectos como lluvia, fuego y otros, enriqueciendo gráficamente la escena renderizada.
- *Sistema de Audio:* El sistema de audio es esencial para que el usuario puede tener una mayor inmersión en el mundo virtual.
- *Animación con esqueleto:* Las animaciones por esqueleto brindar movimientos realistas sobre los modelos geométricos en escena.
- *Sistema de Inteligencia artificial:* Algunos objetos no manipulados por el usuario deben tener un comportamiento propio y éste sistema es el encargado de llevar a cabo estas acciones automatizadas.

3.2 Funcionamiento general de la Arquitectura

La arquitectura está desarrollada bajo el lenguaje C++ y OpenGL. La programación orientada a objetos permite tener una organización adecuada de las clases, métodos y funciones que se ejecutan para efectuar operaciones de carga de objetos 3D, imágenes, audio así como el procesamiento de archivos externos de configuración, shaders y más.

Flujo de Ejecución

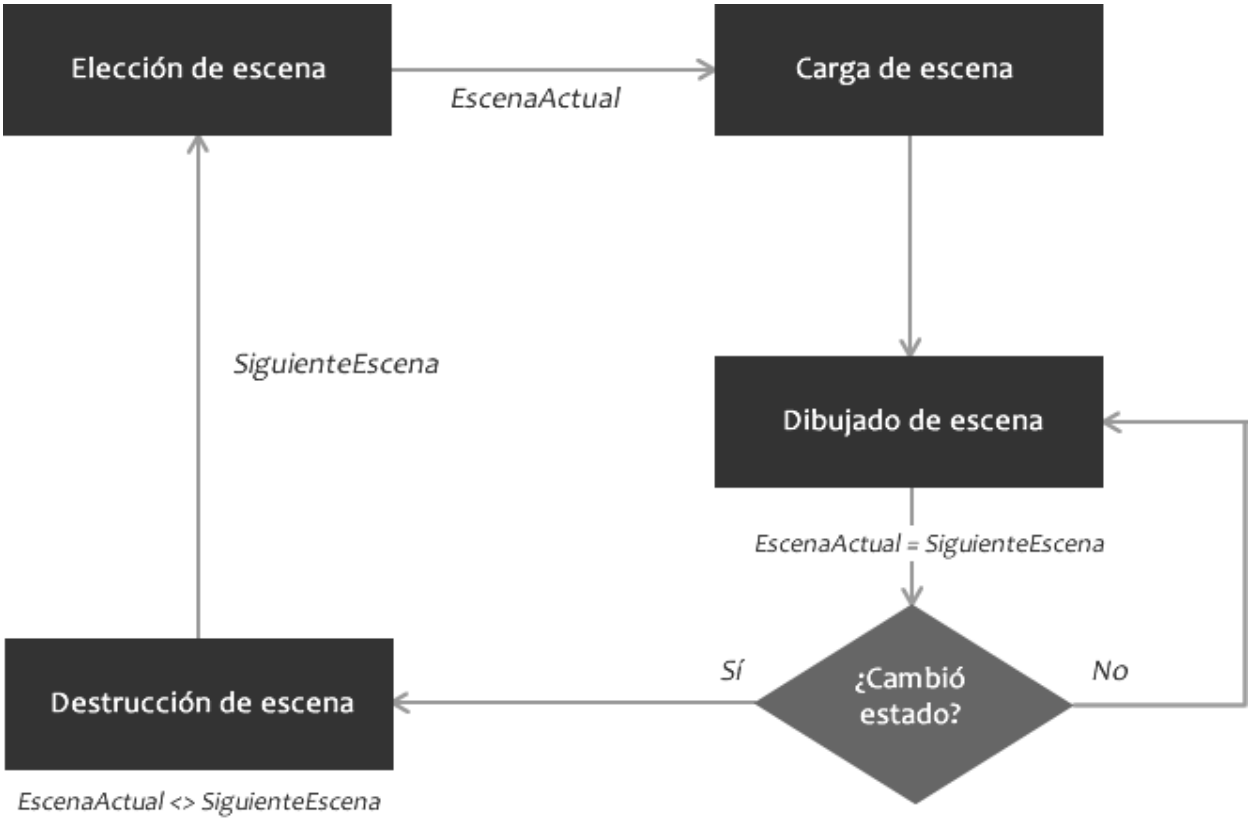

Figura 5. Flujo de arquitectura

El flujo de arquitectura mostrado en la figura anterior, describe la ejecución entre diferentes escenas. La eficiencia de este ciclo permite que el número de escenas no esté limitado ya que al momento de realizar un cambio de escena, se lleva a cabo la destrucción de la misma, la cual libera de memoria la información utilizada ejecutando tareas como:

- Liberar memoria usada por imágenes
- Liberar memoria usada por audio
- Liberar memoria usada por shaders y otros componentes de OpenGL

Es importante señalar que la arquitectura permite guardar información global del usuario y del juego en curso como puntajes, nivel actual, resolución de pantalla, configuración de controles, etc., de modo que los datos que no necesitan ser liberados en cada cambio de escena permanecen sin cambios.

Estructura

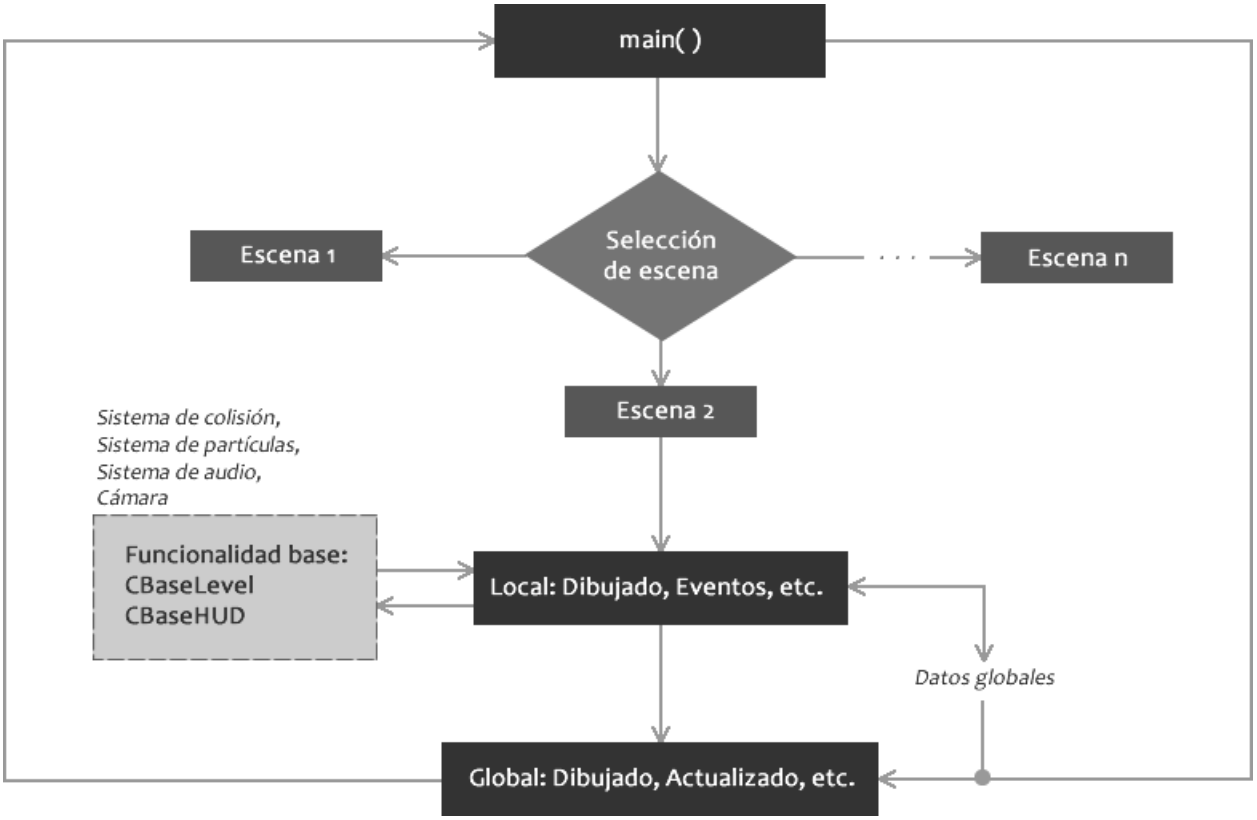

Figura 6. Estructura básica donde se muestra la inicialización de los componentes esenciales para cada escena como la interfaz de usuario, partículas, audio, etc.

La estructura de la arquitectura utiliza dos propiedades de la programación orientada objetos, en este caso del lenguaje C++: herencia y polimorfismo.

La arquitectura cuenta con clases base para crear escenas con la inicialización básica de componentes como el sistema de colisiones, sistema de partículas, sistema de audio, etc., pero al mismo tiempo permite sobrescribir o añadir funcionalidad diferente para estos sistemas y otros métodos como carga de archivos de imagen, audio, shaders, etc. Esto es posible gracias a la propiedad de polimorfismo.

Funcionalidad heredada

Ahora sabemos que las escenas creadas heredan su funcionalidad básica de las clases base de la arquitectura. Ésta funcionalidad contiene lo siguiente:

a. Inicialización de framebuffer objects (FBO)

Un framebuffer object o FBO nos permite dibujar a destinos de renderizado distintos a los buffers tradicionales de OpenGL, como por ejemplo a una imagen (render-to-texture) logrando realizar off-screen rendering. Anteriormente se utilizaban las funciones glCopyTexImage2D() y glCopyTexSubImage2D() la cual tenía limitaciones al crear imágenes de resoluciones en potencias de 2 siendo que para una ventana de 800x600, la máxima resolución sería de 512x512. Posteriormente se implementaron los Pixel Buffers (PBuffer) pero su baja eficiencia demandaba una mejor solución para el proceso de renderizar a una imagen.

Es así que llega la extensión GL_EXT_framebuffer_object (mejor conocida como los FBOs) que se caracteriza dentro de otras cosas por:

- Integración directa con texturas comunes
- No requiere contextos especiales de GL
- Es independiente del sistema de ventana
- Permite a la aplicación el control de cuándo ocurre la generación de mipmaps
- Permite renderizar a un destino sin necesidad del buffer de color (color buffer)
- Permite que los buffers de profundidad (depth), stencil y color se compartan.
- No hay pérdida de los buffers.

Por lo descrito anteriormente los FBOs nos ayudarán a realizar efectos de post-processing como bloom, desaturación, blur y en general cualquier efecto HDR del procesamiento digital de imágenes actual.

b. Inicialización de iluminación

La iluminación es esencial en las aplicaciones gráficas de la actualidad. Ésta parte se encarga de habilitar el número de luces que el usuario tendrá disponibles en la ejecución de su escena.

c. Inicialización y habilitación de niebla

Si la escena lo requiere, se podrán inicializar los parámetros necesarios para crear la niebla exponencial de OpenGL incluyendo el color de la misma.

d. Inicialización y carga del sistema de partículas

En esta parte se pueden crear el número de sistemas de partículas necesarios para la escena en curso.

e. Inicialización y carga de texturas

Se podrán cargar el número de imágenes necesarias para la ejecución normal de la escena.

f. Inicialización y carga de modelos estáticos

Con esta funcionalidad básica, se podrán cargar el número de modelos estáticos que tendrá la escena actual.

g. Inicialización y carga de modelos con esqueleto

La escena puede contar de manera opcional con modelos con esqueleto los cuales se pueden animar. Se podrán cargar el número de este tipo de modelos necesarios para la escena.

h. Inicialización y carga de shaders

Independientemente de los shaders específicos que el usuario desee añadir a su escena, la arquitectura cargará los shaders básicos de iluminación necesarios para toda la escena.

i. Inicialización de sistema de colisiones

El sistema de colisiones requiere de su inicialización básica para funcionar correctamente. Este sistema se puede cargar únicamente en las escenas en donde sea necesario.

j. Inicialización de cámara

Para cada escena es necesario que se carguen los parámetros necesarios de la cámara. La arquitectura cuenta con 2 tipos de cámara:

a. Primera persona

Este tipo de cámara se convierte en los ojos del usuario logrando que exista una inmersión completa en la escena mostrada.

b. Tercera persona

Este tipo de cámara se posiciona detrás de un modelo permitiendo que el usuario vea el entorno en donde se encuentra su jugador virtual.

3.3 Modelado Geométrico

Un modelo es una representación de algo, de un objeto, una entidad, que resalta solamente algunas de sus características importantes; dichas características se eligen en función del uso que se le dará al modelo.

Un modelo geométrico describe componentes con propiedades geométricas inherentes. Estas pueden ser la estructura espacial (como la figura en polígonos de un personaje 3D), la conectividad entre elementos (sus puntos de unión) entre otros.

Dependiendo de lo que se modela, es la importancia en la precisión del modelo resultante, por ejemplo, si hablamos de una reproducción de un objeto real, pues el modelo sintético resultante debe ser muy preciso, sin embargo, si hablamos de una estructura más abstracta como un circuito eléctrico, la forma final que adopte éste no será tan relevante como la información que ofrezca su aplicación.

Herramientas de modelado

En la actualidad existen diversas herramientas de modelado. Una de las más importantes es 3ds Max de Autodesk. Al tener acceso a la versión estudiantil, optamos por hacer uso de esta herramienta.

El formato que utiliza la arquitectura para carga y renderizado de modelos estáticos es 3DS ya que guarda toda la información necesaria como vértices, texturas, coordenadas de textura, pivote del objeto, nombre del objeto, entre otras características más.

3.4 Manejo de Eventos con Volúmenes Especiales

La arquitectura permite que el usuario interactúe con la escena a través de lo que llamamos volúmenes especiales. Un volumen especial es un volumen no visible que al detectar su colisión ejecuta un evento específico.

Los volúmenes especiales son usados en los motores de videojuegos actuales ya que permiten un control de eventos fluido y realista, como por ejemplo cambiar el tipo de color de la escena cuando se entren a lugares específicos o abrir una puerta cuando se cruce algún límite.

Volúmenes Especiales en la arquitectura

Los volúmenes especiales tienen como objetivo activar o desactivar ciertos eventos en la arquitectura para permitir la interacción del usuario con la escena.

Estos volúmenes son prismas dibujados en el modelo 3D de la escena en curso que no serán visibles para el usuario final, pero que serán interpretados por la arquitectura dependiendo de su identificador.

Al cargar el modelo 3D de la escena en curso, la arquitectura realizará un análisis sintáctico (parsing) de los nombres de los objetos contenidos para saber si son volúmenes especiales y de ser así procesarlos adecuadamente.
A continuación se muestra una lista con los identificadores necesarios para configurar volúmenes en el modelo 3D de la escena:

Identificador	Tipo de Volumen	Descripción
VE{nombre}	Volumen de Evento (Event Volume)	Al colisionar con este volumen, se ejecutará el evento *nombre* en la arquitectura.
VP{índice}	Volumen de Post-processing (Post-processing Volume)	Al colisionar con este volumen se visualizará el efecto *índice* de post processing (GLSL).
VL{i, j}	Volumen de Nivel (Level Volume)	Al colisionar con este volumen se visualizará dinámicamente una nueva parte *i* y *j* de la escena actual, dejando de renderizar otras parte de la misma. Todas las partes de la escena deben de cargarse al construir el nivel.
VS{índice}	Volumen de Sonido (Sound Volume)	Al colisionar con este volumen se activará un sonido FX *índice* registrado en el sistema de audio de la arquitectura.
VC{índice}	Volumen de Colisión (Collision Volume)	Esta característica permite colisionar con un volumen *índice* no visible.

Figura 7. Tipos de volúmenes especiales en la arquitectura

Cuando se detecta un nombre de volumen especial, la arquitectura lo guarda en un mapa (map) o una colección (set) de la STL de C++ y así poder ser encontrado rápida y óptimamente en tiempo real.

Volumen de evento (Event volumen)

Este volumen especial permite accionar un evento cuando se detecta una colisión con él. El evento a ejecutar debe ser previamente programado y puede ser cualquier cosa, desde mover algún objeto del modelo 3D hasta iniciar el dibujado de sistemas de partículas. Se realizarán tantos eventos como estén programados.

En el modelo 3D es necesarios especificar el nombre del evento con el cual lo reconocerá la arquitectura de lo contrario no se ejecutará.

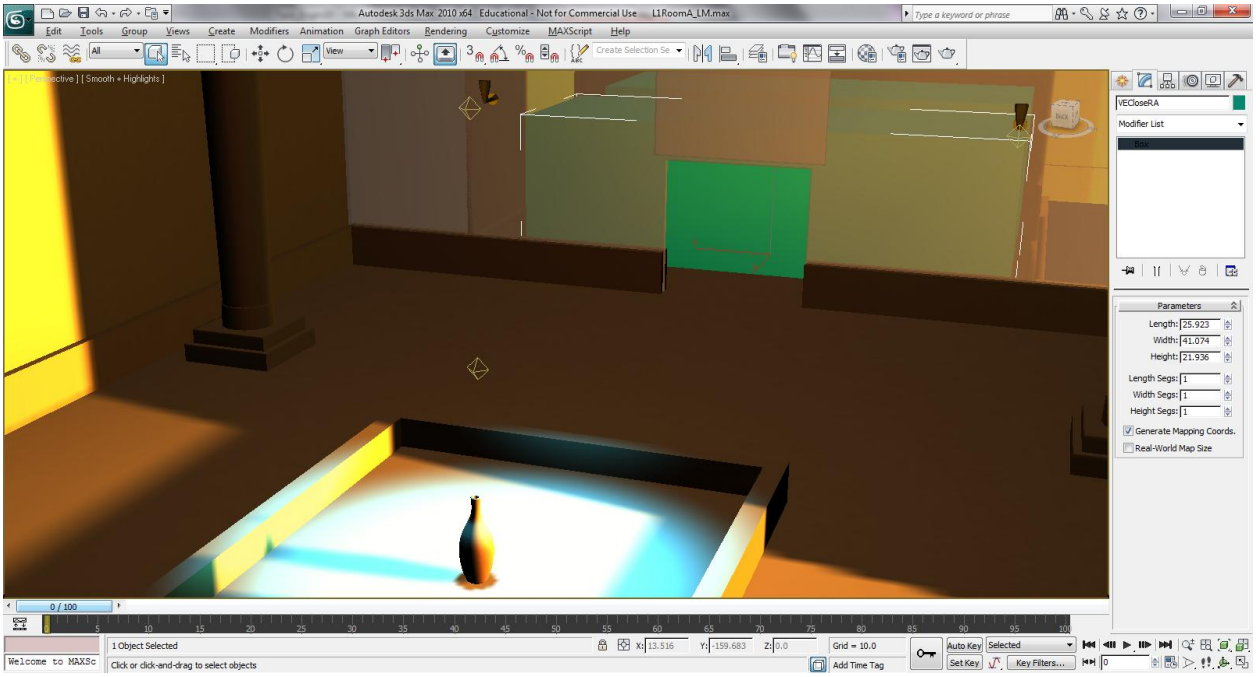

Figura 8. Volumen de evento (Event Volume)

En la figura anterior se muestran 2 habitaciones (modeladas en 3ds Max) las cuales están separadas por un muro (para propósitos de visibilidad, el muro tiene una transparencia) y conectados por una puerta. La habitación del fondo contiene un prisma (el que está seleccionado), el cual funcionará como un volumen de evento que cerrará la puerta que separa las habitaciones al colisionar con él.

Para lograr esto, simplemente se debe nombrar al prisma con el identificador adecuado, en este caso el nombre del prisma es *VECloseRA*, indicándole a la arquitectura que el prisma es un volumen especial de tipo evento, y que al colisionar con él se deberá ejecutar un evento de nombre *CloseRA* (cerrar habitación A).

Volumen de Post-processing (Post-processing Volume)

El volumen de post-processing permite ejecutar shaders con efectos de post-processing como bluring[14], bloom[15], desaturación, entre otros.
Al igual que con el volumen anterior, es necesario añadir prismas a nuestro modelo 3D y nombrarlo de acuerdo al identificador especial.

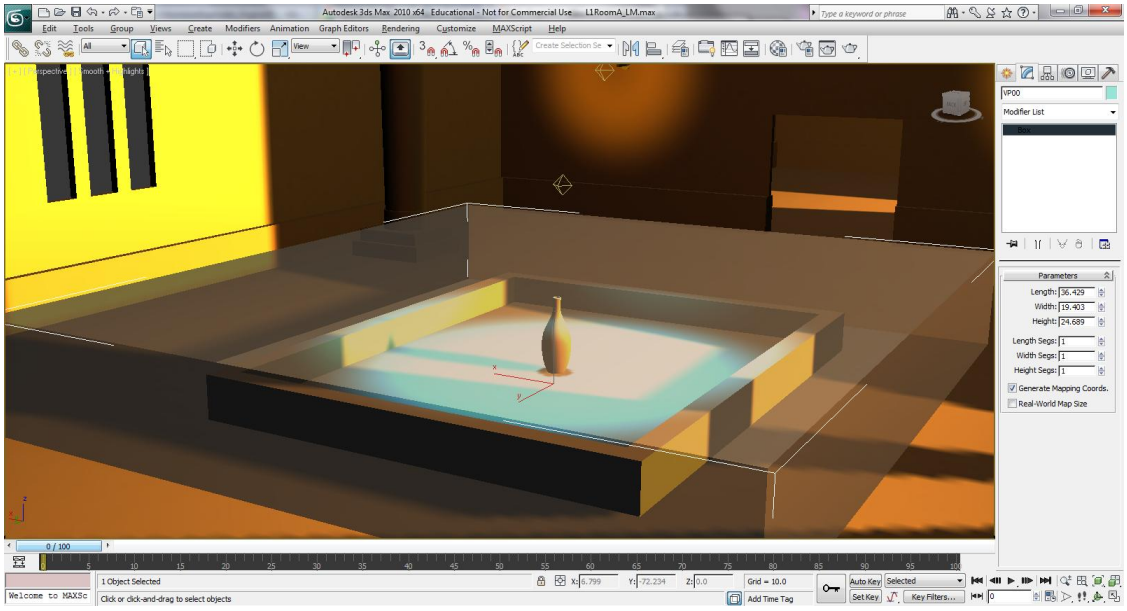

Figura 9. Volumen de Post-processing

En la figura anterior, se muestra un volumen de post-processing (mostrado con transparencia) con el nombre de VP00, lo cual le indica a la arquitectura que cuando se colisione con él, se ejecutará el efecto 0, el cual deberá estar contenido en un archivo con nombre PostProcess0.

[14] Bluring consiste en producir un efecto de borrosidad en una imagen.
[15] Bloom consiste en producir un efecto de brillantez en una imagen, parecido a los que se producen en los lentes de cámaras.

Algo que debemos destacar es que la arquitectura realizará una transición entre el efecto de la escena actual y el nuevo efecto de modo que el usuario verá el cambio de manera gradual y no brusca, dando un mayor realismo.

En la siguiente figura se muestra un ejemplo con una escena original:

Figura 10. Escena renderizada utilizando la arquitectura sin ningún efecto.

Y en esta otra, visualizamos la escena con un efecto blanco y negro, producto de la colisión con el volumen de post-processing:

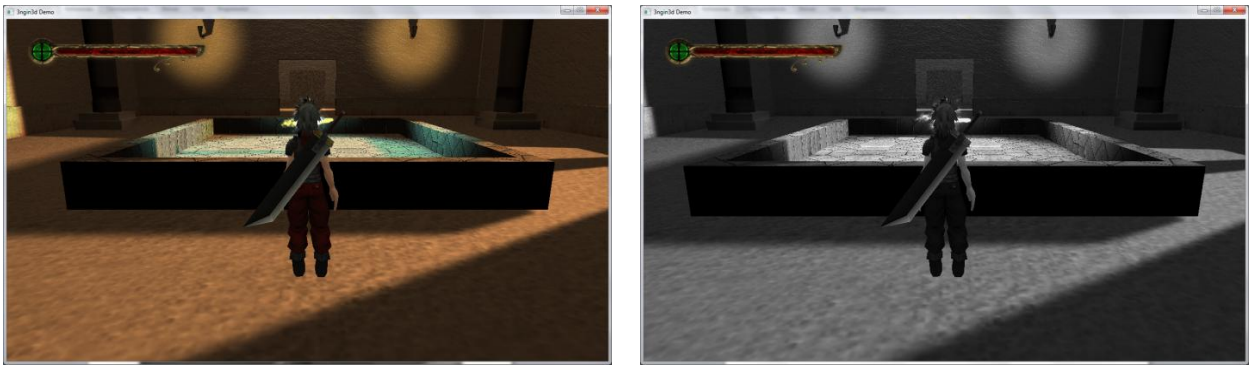

Figura 11. Escena con el efecto de post-processing, se muestra como se cambia gradualmente hasta llegar al efecto final, que en este caso es blanco y negro.

Volumen de Nivel (Level Volume)

Este volumen especial permite dibujar regiones de la escena actual en las que el usuario puede interactuar de manera real, esto es, no es necesario dibujar regiones que el usuario no está visualizando, de esta manera se realiza una optimización de los polígonos dibujados.

El funcionamiento es similar a lo mencionado anteriormente, simplemente se dibuja un prisma en el modelo 3D y se nombra con el identificador adecuado, de modo que si el volumen se llama *VL0102*, la arquitectura dibujará las regiones 1 y 2 cuando se colisione con este volumen dejando de dibujar la región 0.

Internamente cuando se colisiona con un volumen de este tipo, la lista de regiones visibles se modifica, limpiando la lista e insertando los índices de las nuevas regiones.

Esta lista de regiones es considera por todos los sistemas para saber qué objetos son los que se deben procesar en tiempo real en cierto momento del recorrido del usuario a través de la escena.

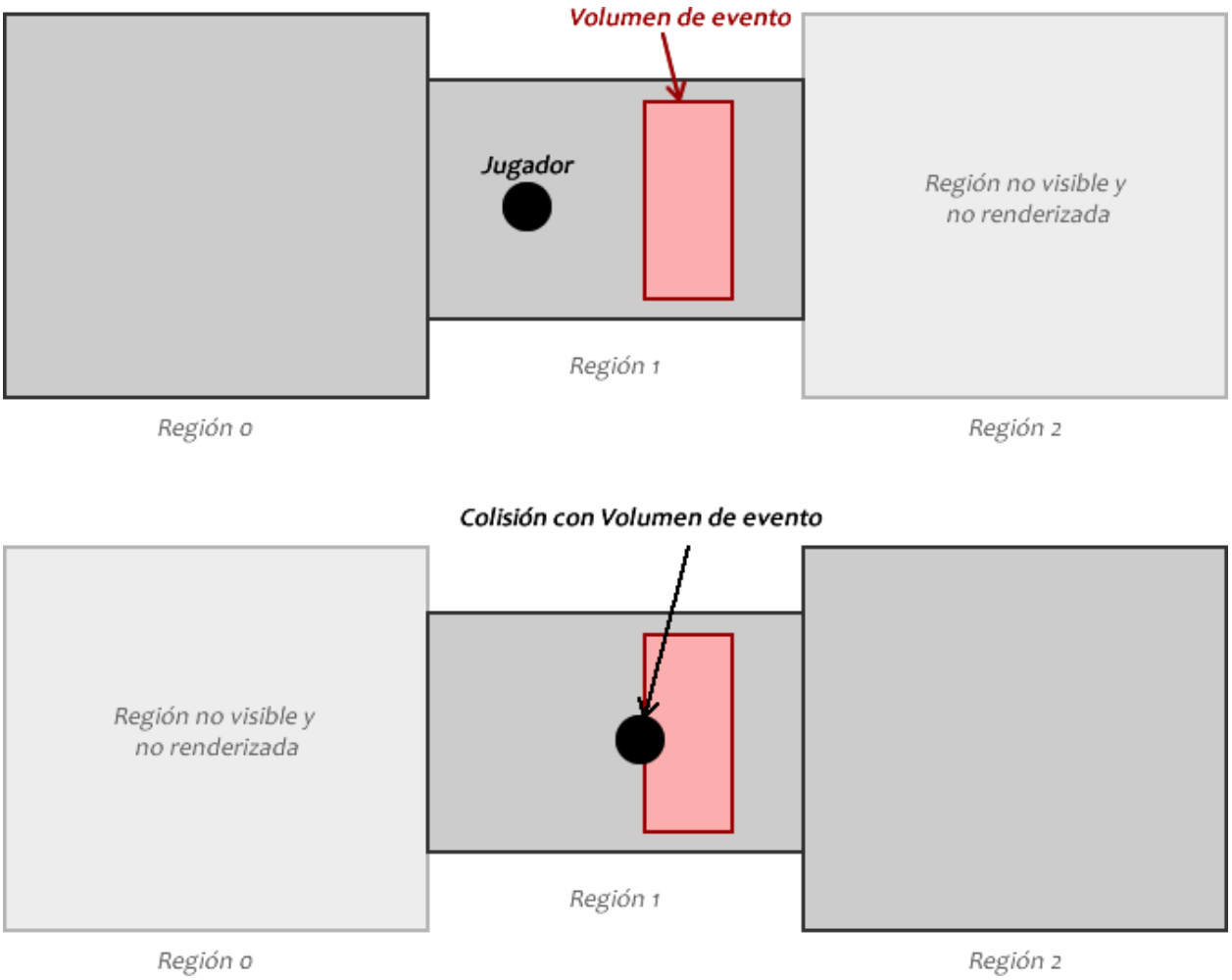

Figura 12. Funcionamiento del volumen de nivel

Volumen de Sonido (Sound Volume)

Al colisionar con este volumen, se producirá un sonido FX previamente cargado por el sistema de sonido de la arquitectura.

Volumen de Colisión (Collision Volume)

La arquitectura permite añadir volúmenes de colisión no visibles los cuales se pueden usar para no permitir el paso del jugador a ciertas áreas, por ejemplo si se tuviera una

plataforma y no queremos que el jugador pueda caer de ella, se puede utilizar un volumen de colisión, el cual no podrá verse pero no permitirá el paso a través de él.

3.5 Texturizado

El texturizado en gráficos por computadora es una parte esencial para ofrecer al usuario una experiencia más realista. En la actualidad todos los juegos y aplicaciones gráficas hacen uso de técnicas de mapeado de textura para producir efectos reales en objetos modelados y el empleo de estas técnicas en nuestra arquitectura no es la excepción.

Mapeado de Textura

Las técnicas de mapeado agregan realismo e interés a las imágenes en los gráficos por computadora. Algunas características de estas técnicas son:

- El mapeado de textura aplica un patrón de color a un objeto.
- Crear imágenes de detalle es mucho menos costoso que crear objetos con muchos polígonos.
- La mayoría de estas técnicas utilizan coordenadas de objeto en lugar de coordenadas globales o de mundo.
- Dependiendo del mapeado, se puede requerir límites para el objeto como una caja, un cilindro o una esfera. Usualmente se transforman estos objetos de límite para que sus coordenadas se contengan en un rango de 0 a 1.

El mapeado de texturas se divide en técnicas 2D y 3D. Las técnicas 2D colocan una imagen plana en un objeto utilizando métodos similares a pegar un papel tapiz a un objeto. Las técnicas 3D son análogas a esculpir un objeto de un bloque de mármol.

Mapas UVW

Si visualizamos un modelo 3D como una escultura, se puede pensar que el mapa de textura es la pintura. Los mapas llenan de color y detalle lo que la geometría no puede. El único problema es que mientras uno puedo pintar directamente en una escultura, un mapa de textura tiene que ser una imagen plana.

Esencialmente, un mapa de textura es un dibujo plano que es cortado, rotado y ajustado al modelo 3D. Un mapa UVW es una plantilla que le dice al software de modelado cómo cortar exactamente el mapa para ajustarlo al objeto.

Unwrap UVW

El software 3ds Max tiene una herramienta llamada Unwrap UVW la cual nos crea la plantilla para crear el mapa de textura de un objeto. Sobre este mapa se puede dibujar la textura para cada objeto y posteriormente aplicarla obteniendo el resultado deseado.

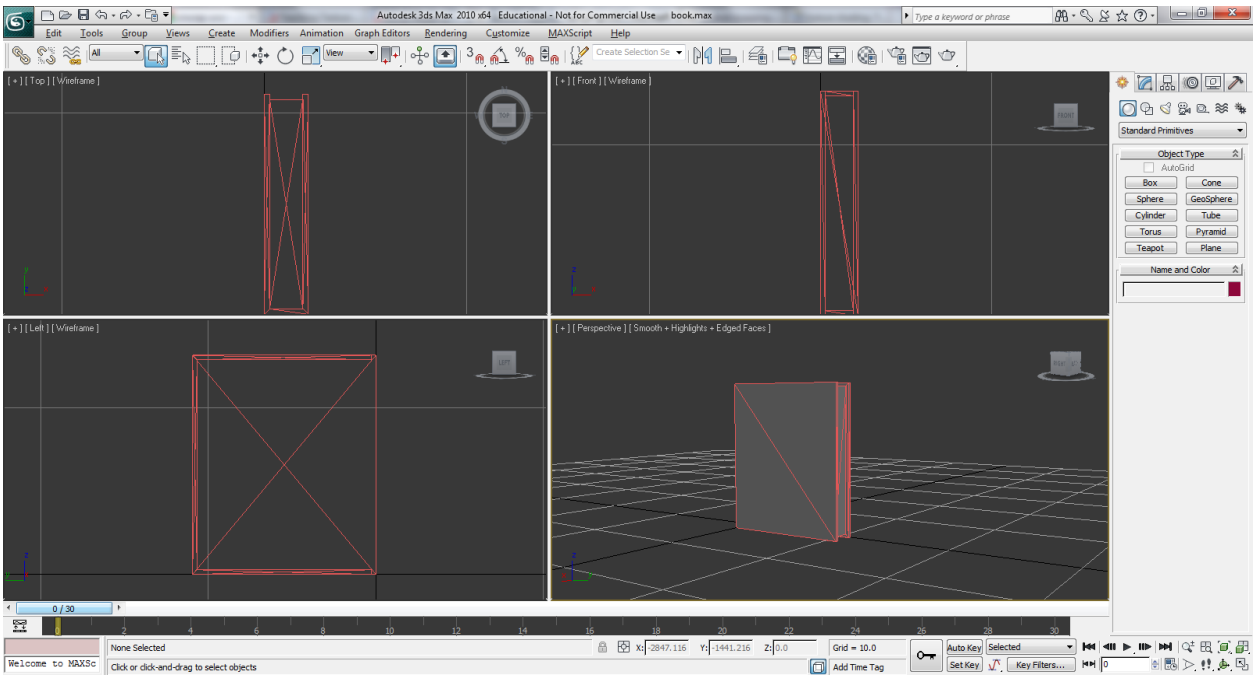

Figura 13. Geometría modelada en 3ds Max

En la figura anterior, se muestra un objeto modelado en 3ds Max. Si lo vemos no parece tener una forma, pero después de aplicar la textura tendremos el objeto de un libro.

Para ello generamos el mapa UVW usando el Unwrap UVW:

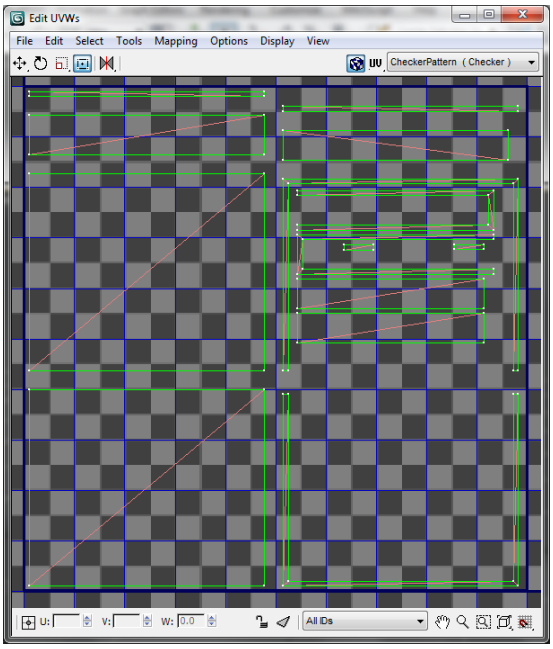

Figura 14. Plantilla generada por el Unwrap UVW de 3ds Max

Con esta plantilla, podemos pintar la textura de nuestro libro, la cual quedaría al final como sigue:

Figura 15. Mapa de textura para nuestro modelo

Una vez teniendo nuestro mapa de textura, simplemente la añadimos y la aplicamos a nuestro objeto. 3ds Max mapeara correctamente la textura con las coordenadas de textura que generó al hacer el Unwrap UVW:

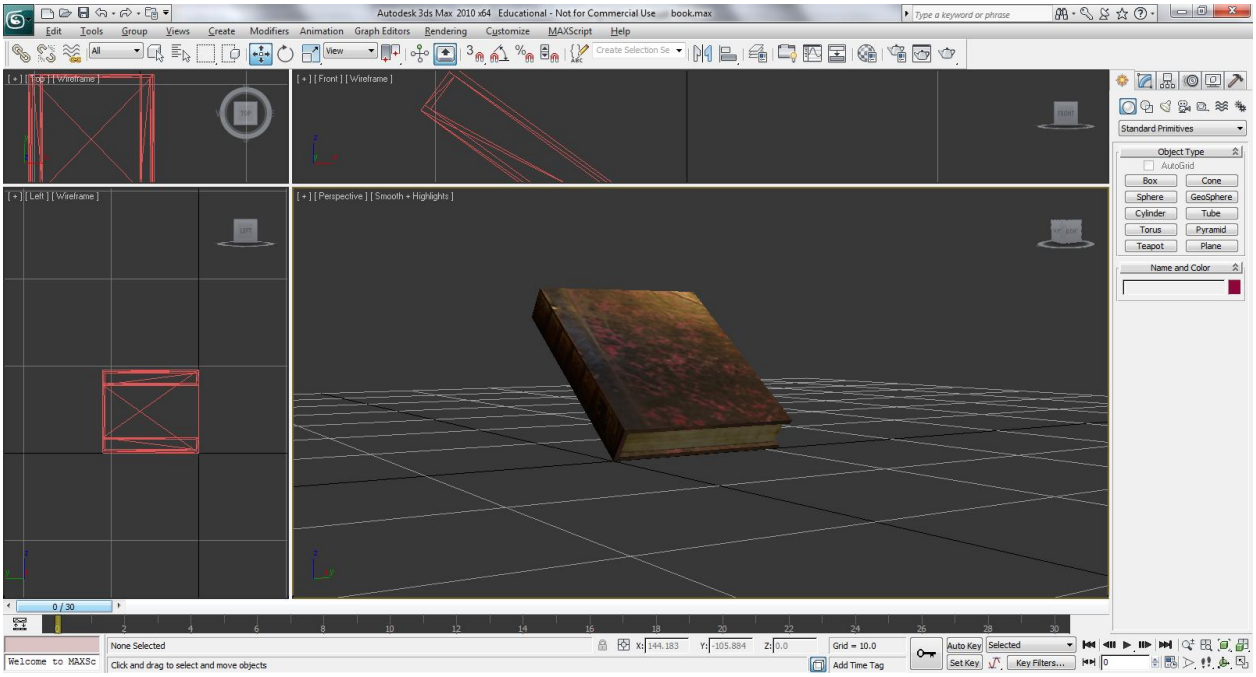

Figura 16. Geometría con textura mapeada usando Unwrap UVW

Normal Mapping

Normal mapping es una técnica usada para iluminar un modelo 3D de bajos polígonos como si fuera un modelo altamente detallado. No añade ningún detalle a la geometría original así que los bordes del modelo se verán igual pero su interior se visualizará como un modelo de alta resolución de polígonos (con relieves o hendiduras).

Proceso

Primero es importante saber en qué dirección apunta cada punto de la superficie del modelo. Esta dirección es llamada normal. La normal se puede visualizar como una línea que se extiende desde la superficie del modelo. Posteriormente es necesario saber la posición de la luz en nuestra escena. Creamos una línea desde el punto en la

superficie hasta la posición de la luz. Esta línea se llama vector de luz. Con esto tenemos 2 vectores, el de luz y la normal; el ángulo entre ellos nos dirá la iluminación para ese punto en la superficie.

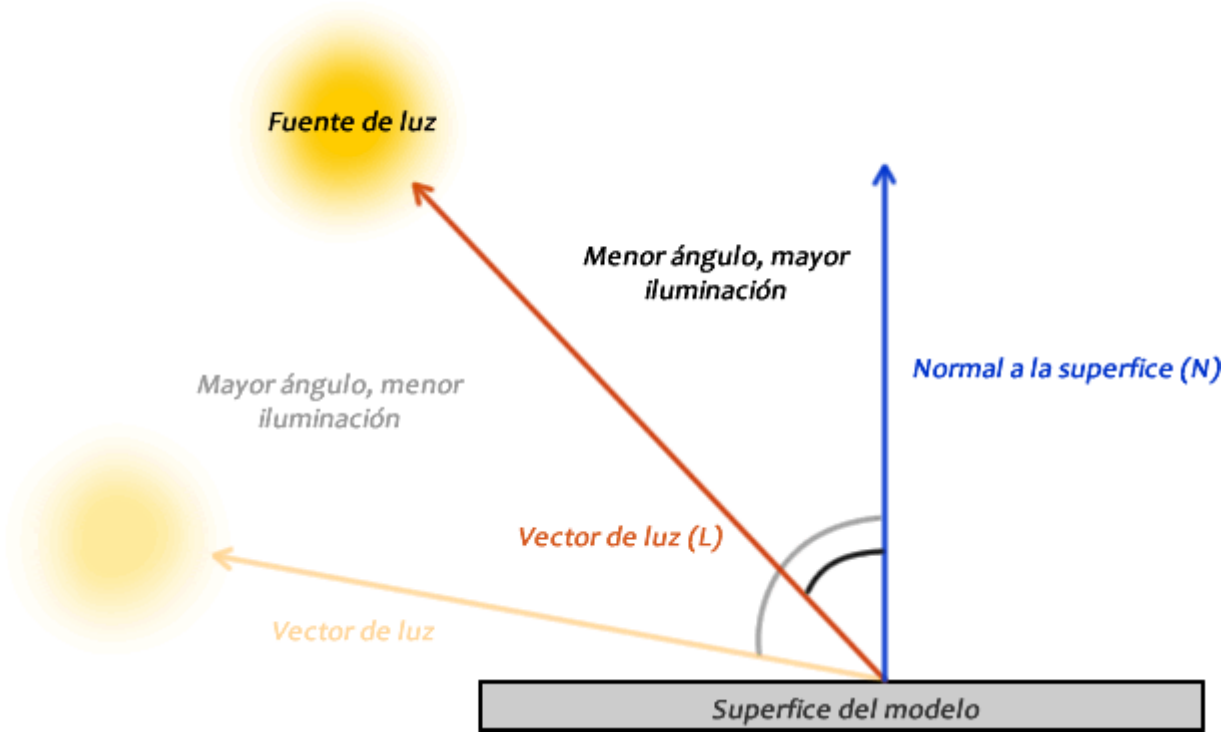

Figura 17. Vector de luz y normal

Si el ángulo es pequeño (los 2 vectores apuntan casi a la misma dirección) entonces sabemos que el punto en la superficie necesita estar iluminado porque apunta prácticamente hacia la fuente de luz. Si el ángulo es grande entonces no necesita estar iluminado porque apunta en una dirección contraria a la fuente de luz.

Por lo tanto, la fórmula para obtener la iluminación para cada punto en la superficie será:

$$iluminación = N \cdot L$$

Ya que sabemos que el producto punto entre 2 vectores, nos dará el coseno del ángulo entre los mismos.

Para que nuestro objeto de superficie plana se visualice con relieves o hendiduras, podemos hacer uso de la técnica de normal mapping. Simplemente basta con generar un mapa de normales con algún software de imagen tomando como base el mapa de textura original y posteriormente aplicar el modelo de iluminación por pixel (por vértice es menos realista) para visualizar en tiempo real el efecto deseado.

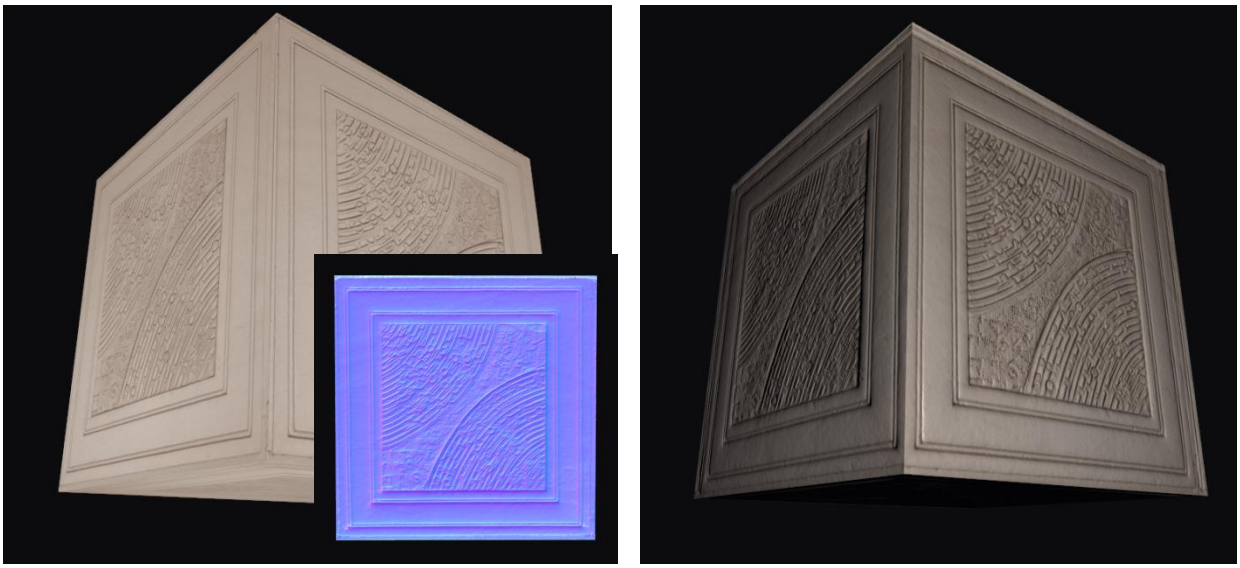

Figura 18. A la izquierda se muestra un objeto con un mapeado de textura simple, en el recuadro el mapa de normales generado a partir de la textura original. A la derecha se muestra el objeto con el mapeado de textura pero además con el mapeado de mapa de normal e iluminación por pixel.

La arquitectura realiza este efecto automáticamente, únicamente se debe añadir la textura de difusa (mapa de textura), mapa de normales y activar la iluminación por pixel.

Light Mapping

Para que una escena 3D se vea realista, se deben tomar en cuenta los efectos que producen las fuentes de luz sobre los objetos de la escena. Los modelos de iluminación no son suficientes para lograr este efecto.

OpenGL tiene la capacidad de añadir múltiples texturas para cada objeto (la tarjeta gráfica debe soportar esta característica), y la arquitectura hace uso de ésta característica para añadir light maps o mapas de luz.

La capacidad de multitextura[16] permite dibujar la textura de difusa, mapa de normal y el light map para cada objeto en un solo proceso de renderizado.

Light Maps

Un light map o mapa de luz, es una textura o grupo de texturas que contienen información acerca de la iluminación de la escena 3D. Se puede guardar la información de la iluminación en el canal alfa del light map, en los canales de colores o en ambos.

Con el uso de light maps podemos simular sombras e iluminación estática para nuestra escena. Para ello hacemos uso de las herramientas que proporciona 3ds Max para realizar esta tarea.

Figura 19. Escena renderizada sin el uso de light maps.

[16] La capacidad multitextura permite mezclar múltiples texturas en una sola superficie.

En la figura anterior, se muestra una escena cargada en la arquitectura usando únicamente mapeado de textura e iluminación por pixel, produciendo en efecto poco atractivo para el usuario.

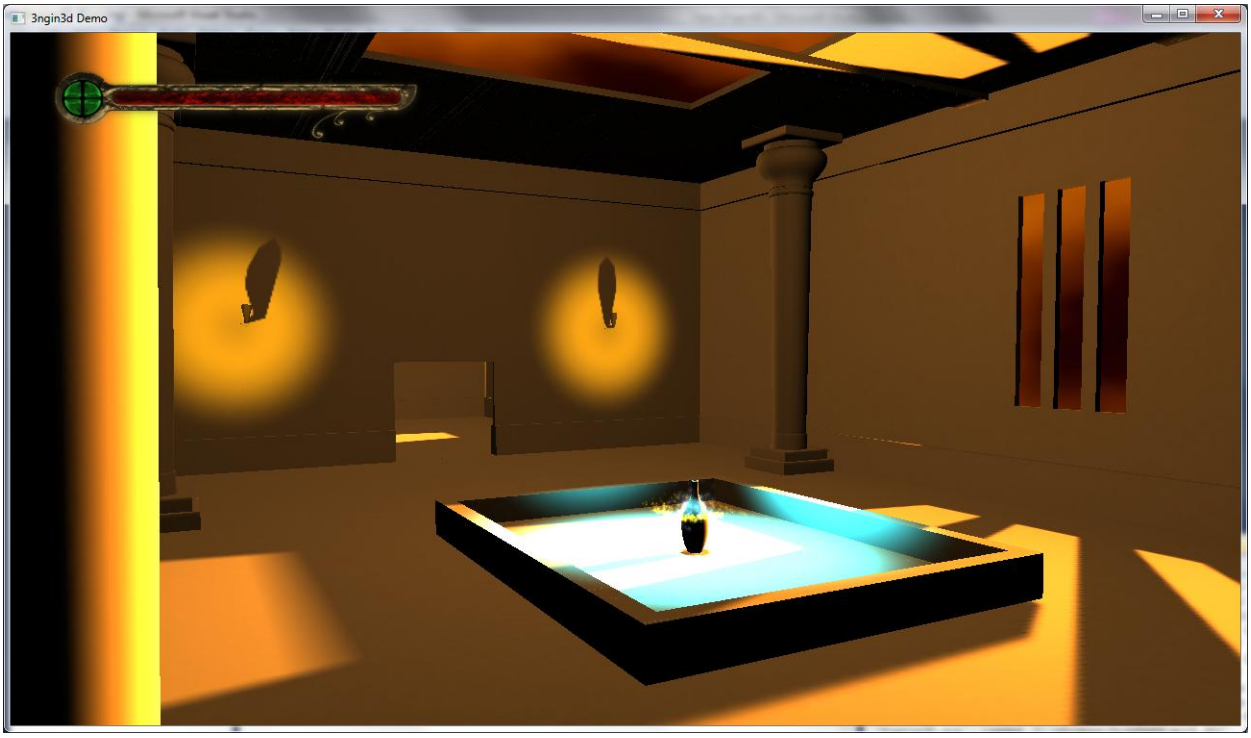

Figura 20. Escena renderizada sólo usando light maps.

En la figura mostrada, se visualiza la misma escena usando sólo los light maps para ver cómo afecta la luz cada objeto y las sombras estáticas que éstos producen.

Finalmente en la siguiente figura observamos la mezcla de ambas escenas, es decir con el uso de mapas de textura, iluminación por pixel y el uso de light maps para simular sombras y fuentes de luz:

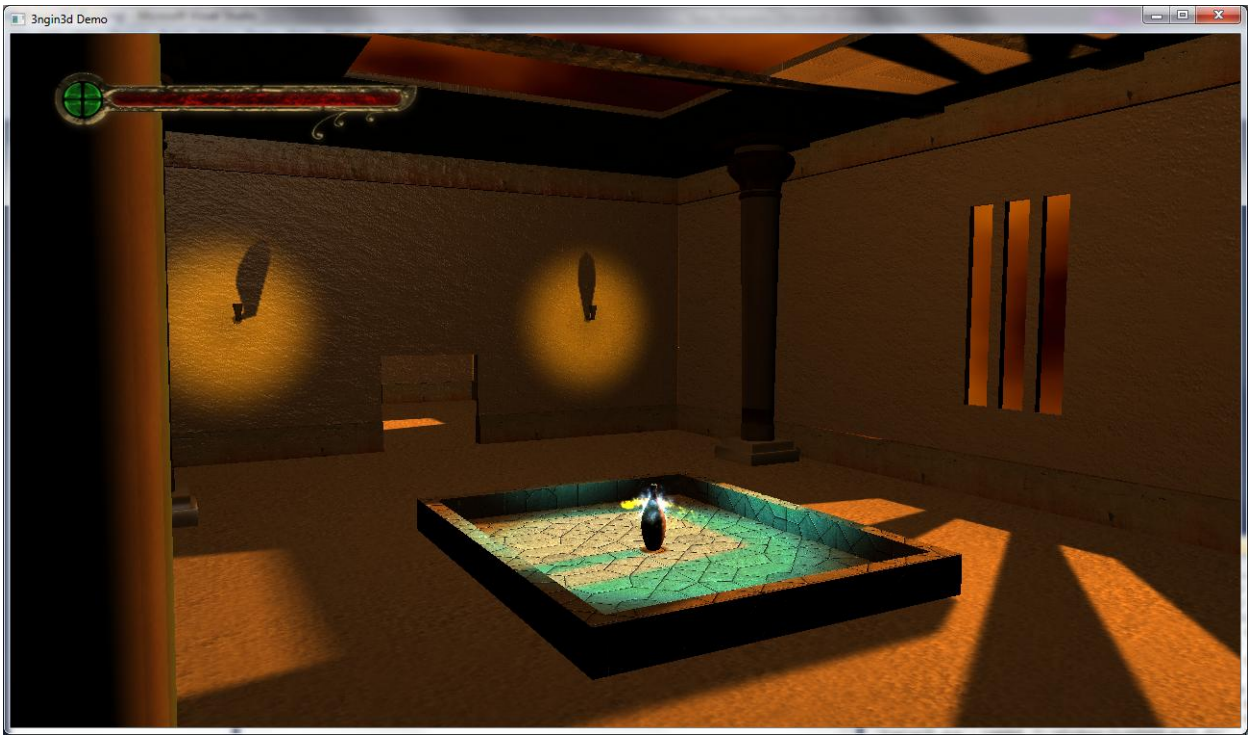

Figura 21. Escena renderizada por la arquitectura usando iluminación por pixel y light maps.

Como se puede ver, el uso simultáneo de estas técnicas de mapeado e iluminación permiten que la escena final muestre un realismo superior y creíble para el usuario.

Para aprovechar ésta técnica la arquitectura sólo requiere que se añadan los light maps generados por el software de modelado en una ruta especial y se configurará automáticamente.

3.6 Efectos de Post-processing

Como su nombre lo dice, el post-processing son procesos llevados a cabo una vez que la escena se ha terminado de renderizar. Para ello es necesario que el renderizado no se lleve a pantalla sino que se guarde en una imagen.
La arquitectura lleva a cabo ésta tarea haciendo uso de los FBOs de OpenGL de los que se habló en el tema 3.2. Con ello toda la escena se guarda en una imagen a la que posteriormente se le realizará algún procesamiento digital de imágenes (PDI) a través del uso de shaders.

Renderizado de alto rango dinámico (HDR Rendering)

En la actualidad muchos videojuegos utilizan ésta técnica que permite obtener efectos de iluminación realistas que utilizan valores de luminancia con un alto rango dinámico, es decir permiten obtener un mayor rango entre zonas claras y oscuras representando con exactitud los niveles de intensidad en imágenes de escenas reales.

La arquitectura ejecuta ésta técnica con la aplicación de filtros a la imagen que se obtiene del FBO. Para ello se promedian los valores de cada pixel de la imagen, para obtener la intensidad deseada por pixel y posteriormente dependiendo de su valor, se multiplica por un escalar para obtener la intensidad final.

Figura 22. Escena renderizada directamente del FBO

Figura 23. Escena renderizada después de aplicar el efecto de bloom.

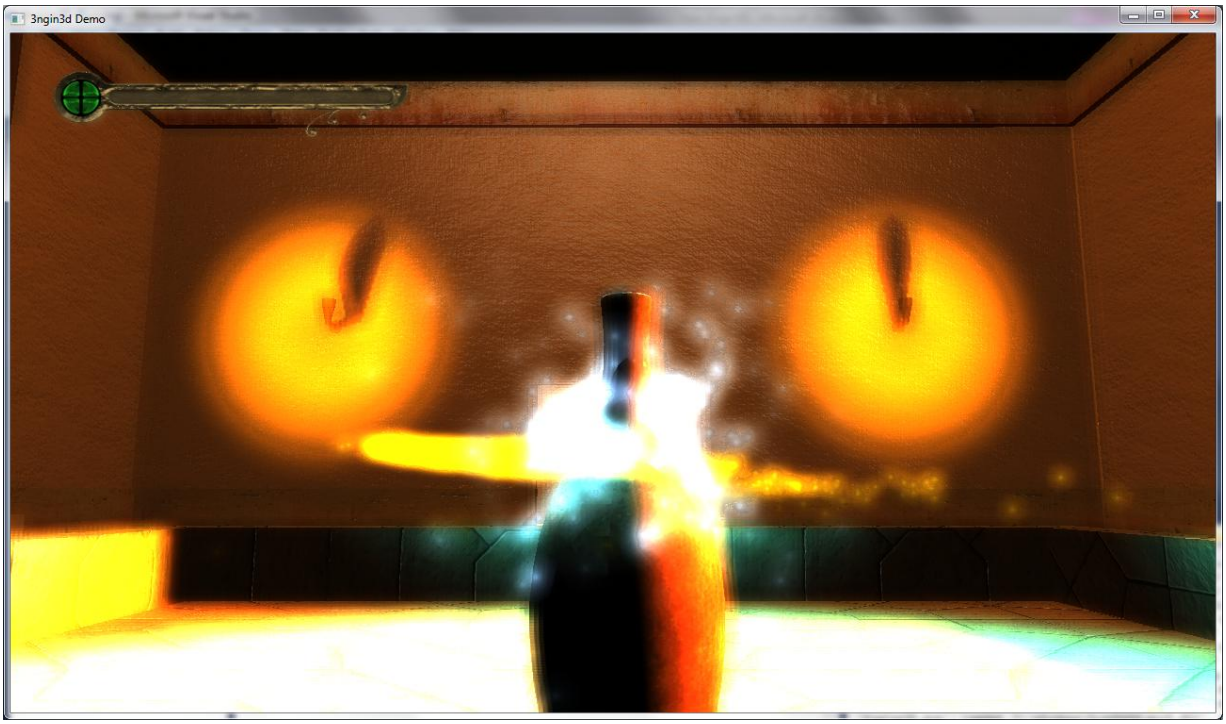

Figura 24. Escena renderizada con un factor de bloom mayor.

Bluring

Este efecto permite visualizar la imagen como si estuviera desenfocada. La arquitectura simula este efecto aplicando un filtro de tipo Gaussiano a la imagen del FBO.

Tomando como imagen original la figura anterior, aplicando este efecto de post-processing obtenemos el siguiente resultado:

Figura 25. Escena renderizada con el efecto de bluring

Desaturación

Este efecto consiste en desaturar los colores de la imagen del FBO produciendo una nueva imagen con un coloreado distinto.

Para producir un resultado blanco y negro, la arquitectura utiliza un shader que toma en cuenta el coeficiente de luminosidad relativa para obtener la intensidad en cada pixel (realizando el producto punto entre ambos) y posteriormente mezclando el color original con esta intensidad calculada. El efecto resultante es el siguiente:

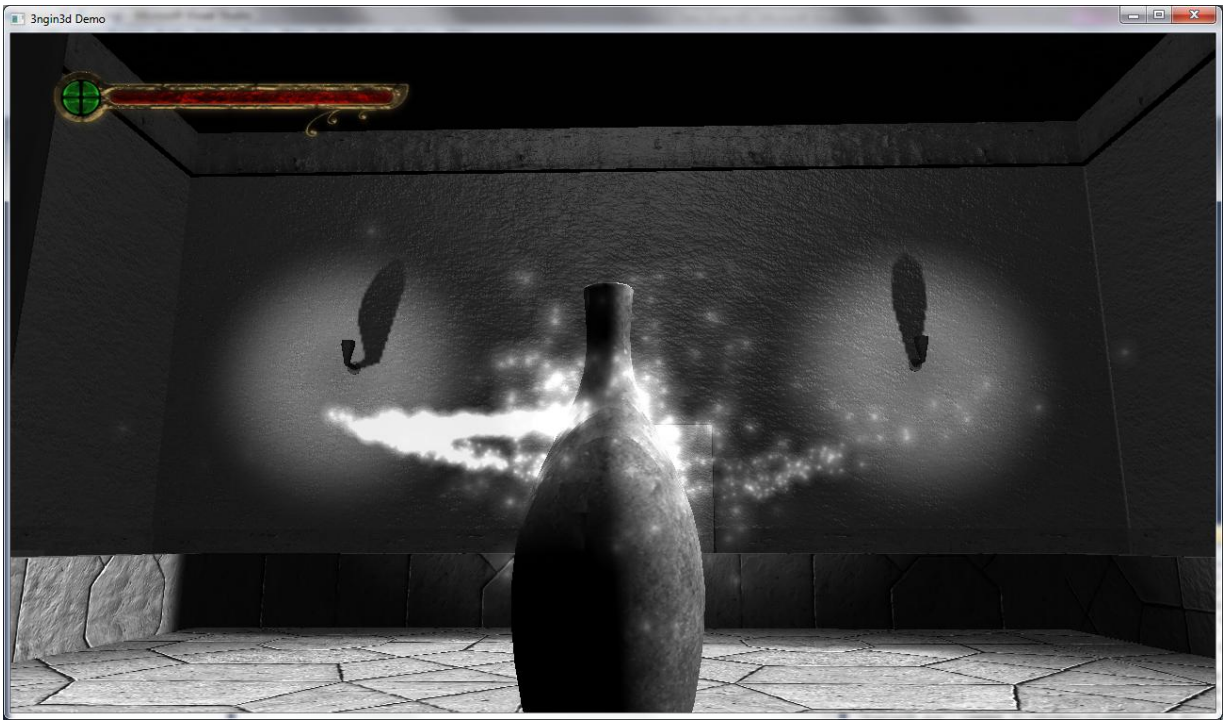

Figura 26. Efecto blanco y negro

Otro efecto de este tipo es aplicar un filtro necesario para obtener los tonos sepia. De igual manera se obtiene la intensidad calculando el producto punto entre el valor del pixel original y el valor del filtro sepia. Posteriormente se mezclan ambos valores y se obtiene el resultado siguiente:

Figura 27. Efecto de tonos sepia

3.7 Sistema de colisiones

La detección de colisiones es una parte fundamental en las aplicaciones gráficas para dar mayor realismo a las escenas virtuales con las que interactúa el usuario. Existen diversos modelos para detectar colisiones entre dos o más objetos en el espacio 3D. La arquitectura utiliza las siguientes:

Bounding sphere (BS) o esfera de colisión

Este modelo de colisión consiste en envolver completamente al objeto con una esfera de tal forma que si la suma de los radios de dos esferas es mayor a su distancia, existe una colisión.

Si tenemos dos objetos, cada uno con su esfera de colisión, y sabemos sus posiciones, lo que debemos hacer para saber si están colisionando es primeramente obtener la distancia entre sus centros:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

que también se puede reescribir como:

$$d^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2$$

Para que exista colisión se debe de cumplir lo siguiente:

$$d^2 \leq (r_1 + r_2)^2$$

por lo que sumamos los radios y lo elevamos al cuadrado. Si la distancia al cuadrado es menor o igual a la suma de los radios al cuadrado entonces hay colisión. En la siguiente figura se puede visualizar:

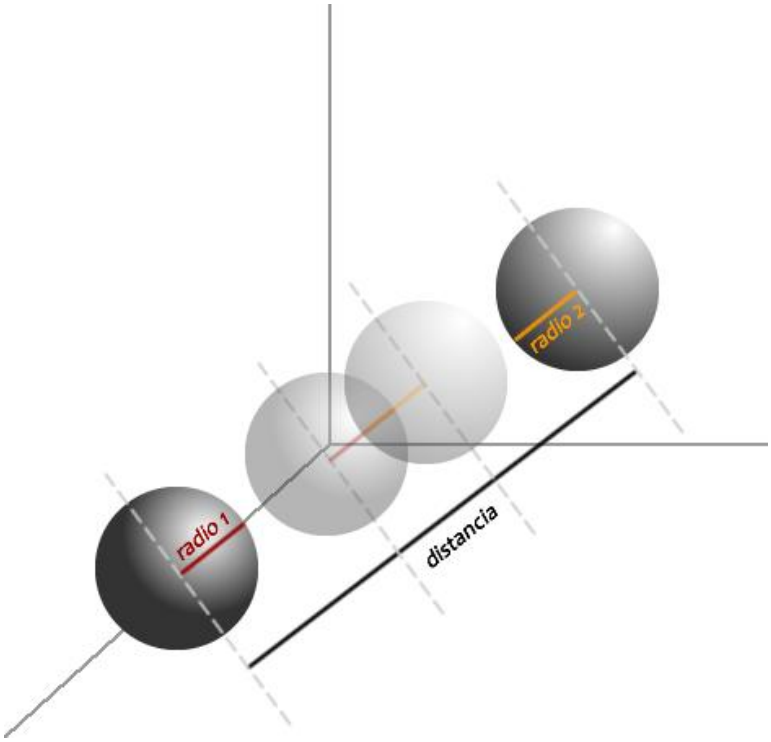

Figura 28. Colisión esfera-esfera

Axis Aligned Bounding Box (AABB) o Caja de colisión alineada a los ejes

Este modelo corresponde a una caja que se alinea a los ejes (x, y, z) y envuelve a todo el objeto. Analizando para cada eje:

$$Si\ (B_{min}\ >\ A_{max})\ o\ (A_{min}\ >\ B_{max})\ entonces\ no\ hay\ colisión$$

Figura 29. Colisión alineada a los ejes

Si para cada eje hay colisión entonces podemos deducir que existe colisión entre las cajas.

Uso del Sistema de Colisiones

La arquitectura utiliza los métodos mencionados anteriormente para su propio sistema de colisiones. La detección de colisiones se divide en 2 partes para una mayor eficiencia en el cálculo:

a. Detección de colisión BS

En una primera instancia, se realiza una discriminación de objetos con los que el jugador realmente podrá colisionar, ya que no es necesario realizar la detección para objetos que están a una gran distancia de la posición del jugador. Para ello se emplea la técnica de Bounding Sphere o BS la cual no es costosa y se calcula rápidamente. Con esto se crea un mapa de objetos con los cuales se utilizará una técnica más precisa de detección de colisiones, en este caso AABB, la cual se adapta mejor a la forma general de los objetos en la escena.

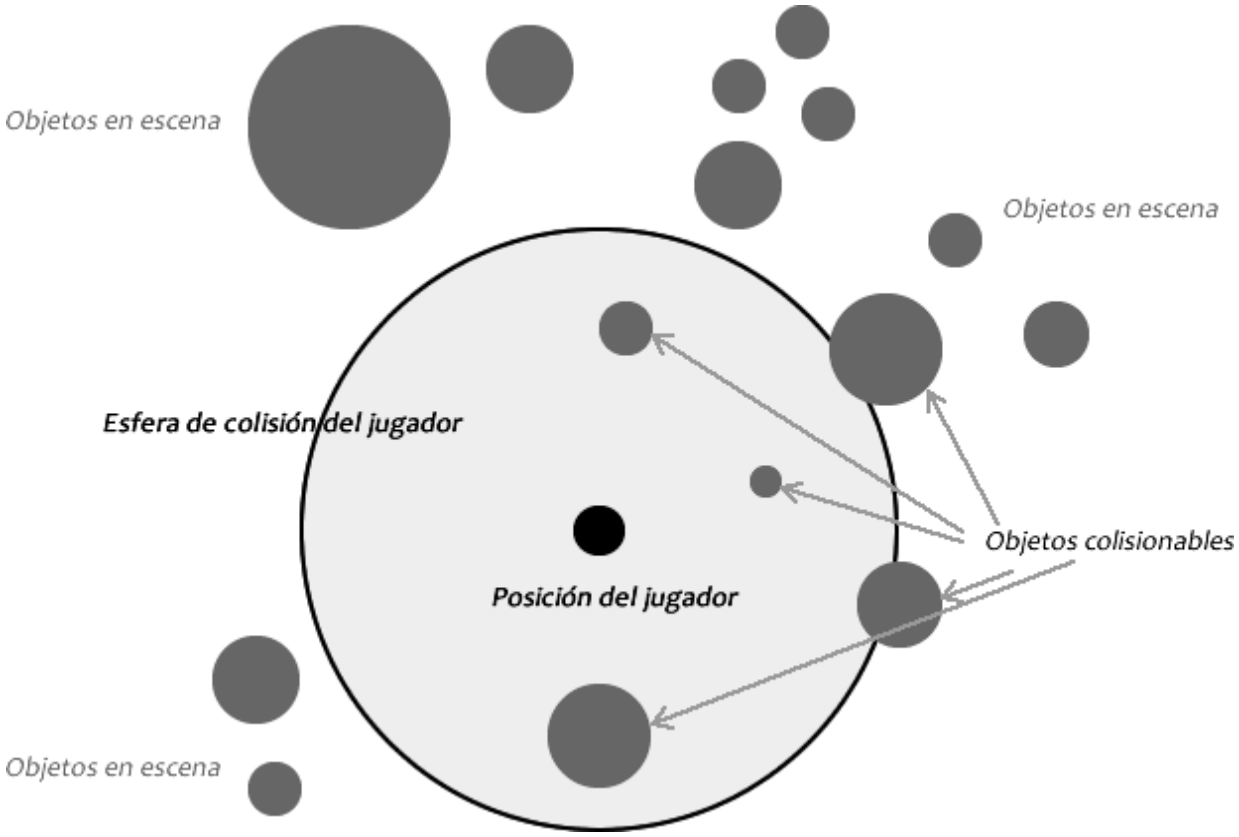

Figura 30. Sólo los objetos dentro de la esfera de colisión del jugador, serán considerados para detección de colisiones más precisas como AABB (La imagen muestra una proyección ortogonal de una escena en 3 dimensiones)

b. Detección de colisión AABB

Con el mapa de objetos potenciales para colisión, se aplica éste método de detección de colisiones y se realiza el evento requerido como por ejemplo no permitir que el jugador atraviese muros, columnas, etc., o activar eventos especiales. Se usó ésta técnica ya que no es necesario tener tanta precisión en la detección de colisiones como lo pueden proporcionar algoritmos como OBB (Oriented Bounding Box) o cajas orientadas a los ejes, colisión triángulo rayo, etc., siendo el cálculo mucho más rápido.

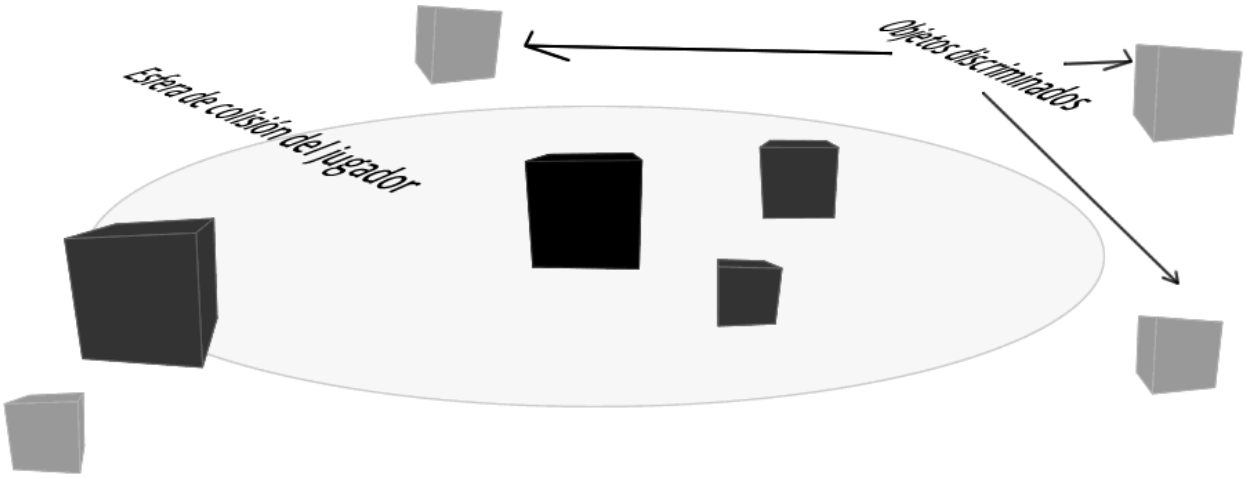

Figura 31. Con los objetos discriminados, se puede realizar la detección con los objetos con los que el jugador realmente puede colisionar

3.8 Sistema de partículas.

Hoy en día es muy común encontrar toda una amplia gama de efectos dentro de las aplicaciones gráficas tales como explosiones, efectos de armas, humo, entre otros. Una herramienta muy usada en la creación de efectos especiales dentro del desarrollo de los videojuegos son los llamados sistemas de partículas. Las partículas son sprites[17] con un tiempo de vida corto que la mayoría de las veces son generadas en grandes cantidades. Cabe remarcar que al ser vistos únicamente como efectos especiales, no es necesario hacer detección de colisión ni tampoco es indispensable que el sistema tenga conocimiento de lo que ocurre a su alrededor.

Las formas más comunes que se usan para la representación de una partícula son:

- Puntos: Los puntos son usados en numerosos tipos de efectos, especialmente en aquellos en los que no son vistos cercanamente.
- Líneas: Las líneas pueden ser usadas en efectos de estelas. Aquí las líneas conectan la posición actual de la partícula con su última posición.

[17] Es una imagen bidimensional o animación.

- Plano texturizado: Probablemente son los más ampliamente usados ya que ofrecen una gran flexibilidad. La partícula por sí misma es un plano o un par de triángulos, lo cual permite que se pueda mapear una textura sobre el mismo.
- Point sprites: Son básicamente billboards[18] que usan aceleración por hardware. La principal ventaja de usarlos es que en lugar de especificar un plano texturizado, lo cual requiere cuatro vértices y todos sus atributos relevantes, se especifican las partículas como un único punto, lo cual reduce la cantidad de geometrías enviadas a la tarjeta gráfica. Es decir, por cada punto OpenGL automáticamente calculara un billboard y aplicara cualquier textura en él.

Funcionamiento

El sistema de partículas diseñado para la arquitectura consta de 2 partes importantes, una de ellas es la partícula, que es la parte visible del sistema, y por otra lado el emisor de las partículas que es una región no visible en el espacio tridimensional dentro del cual se generan las partículas e inician su movimiento.

A continuación se describirá detalladamente el planteamiento del sistema sin abundar en los detalles técnicos, ya que estos se podrán consultar en el siguiente tema.

Comenzando con las características de cada uno de los componentes, es necesario considerar la jerarquía de los componentes el cual puede ser visto en la siguiente figura:

[18] El término billboard es aplicado al uso de la técnica bajo la cual los objetos son representados por imágenes bidimensionales aplicados a un plano el cual típicamente se mantiene perpendicular a la línea de vista.

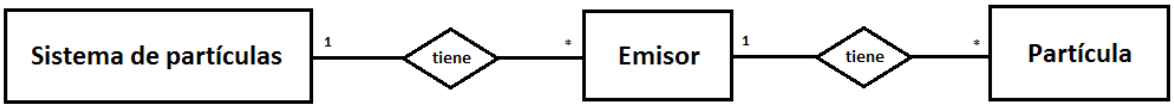

Figura 32. Diagrama Entidad-Relación del sistema de partículas.

El diagrama anterior nos representa la forma en que está conformado un sistema de partículas dentro de la arquitectura, esto quiere decir, dentro de alguna aplicación desarrollada podemos encontrar uno o más sistemas de partículas, cada sistema de partículas contará con varios emisores y cada emisor puede generar una determinada cantidad de partículas.

Una característica importante de mencionar es que los sistemas de partículas tendrán la capacidad de soportar varios comportamientos los cuales pueden predefinirse de acuerdo a las necesidades del programador, esta característica permitirá que se cuenten con varias texturas diferentes y varios emisores de partículas con características propias tales como diferentes puntos de fuga de las partículas, movimientos compuestos (traslación y rotación), entre otros.

Implementación

El sistema de partículas fue diseñado para hacer uso de point sprites. Como se había mencionado anteriormente, estos son ideales para crear sistemas de partículas con un alto rendimiento ya que solo es necesario enviar un único vértice para el punto que representara una partícula en lugar de los cuatro vértices usados por un plano texturizado. Otra de las ventajas que ofrecen es que todos los cálculos matemáticos involucrados en la alineación de la vista sobre el plano final se realizan en la GPU en lugar del CPU.

A continuación se explicara más a detalle las características de cada uno de los componentes que conforman nuestro sistema de partículas.

Sistema de partículas

En el diseño del sistema de partículas se contemplaron los atributos siguientes:

- Posición y pivote. La posición define la ubicación actual donde se encontrará el sistema de partículas. Es importante señalar que el pivote es por defecto la misma posición del sistema y fue considerado en el diseño para dar un mejor soporte a movimientos complejos que ofrecerán los emisores.
- Numero de emisores. Indica el número de emisores que compondrá nuestro sistema de partículas, las características de éstos se detallan más adelante.
- Movimientos. Indica al sistema de partículas los movimientos que podrá realizar en caso de que hayan sido definidos. Estos pueden ser traslación, rotación y escalamiento.

Emisores

Para el diseño de los emisores se contemplaron las características siguientes:

- Posición y pivote. La posición define la ubicación actual donde se localizará el emisor, para este caso el pivote de un emisor es por defecto la posición del sistema de partículas.
- Numero de partículas. Se refiere a la cantidad de partículas que generará el emisor. Las características de las mismas se detallan más adelante.
- Tamaño de las partículas. Indica el tamaño de las partículas que se generarán dentro del emisor.

- Textura. Como mencionamos anteriormente los point sprites son vistos como planos aunque solo se use un vértice para definirlos, esto nos permite asociarles una textura que otorgara una mejor apariencia a las mismas.

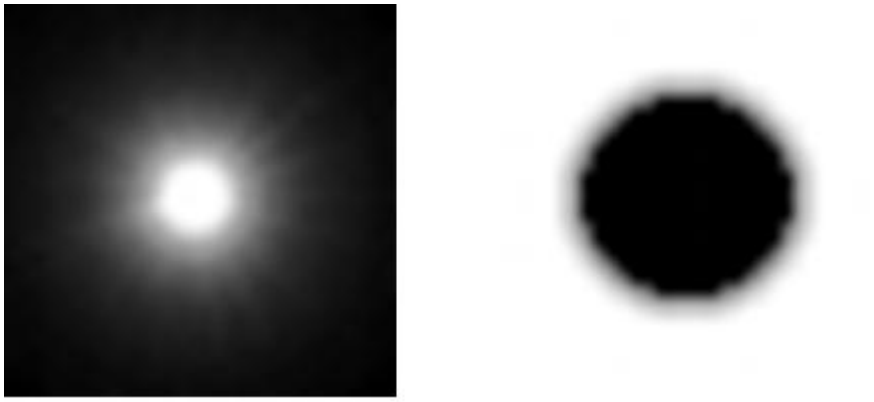

Figura 33. Características de las texturas que son soportadas por el sistema de partículas.

- Color. Cada emisor podrá generar partículas de un único color o en su defecto por un color aleatorio.
- Volumen. Indica la forma del emisor de partículas, por default siempre será un prisma rectangular al cual se le podrán definir las dimensiones deseadas.

Partículas

Cada una de las partículas cuenta con los siguientes atributos:

- Posición. Indica cuál es la posición actual de la partícula.
- Velocidad. Permite definir un vector que especifica el movimiento de la partícula si es necesario.
- Color. Define el color de la partícula, con la posibilidad de ofrecer efectos de transición de un color a otro.
- Tiempo de vida. Se refiere al tiempo en el que la partícula será visible en la escena, para lograr un mejor efecto este atributo está relacionado con la

transparencia de la misma, lo cual nos permite ofrecer un efecto de desvanecimiento al acercarse el tiempo de vida límite.

Con estas características disponibles es relativamente fácil crear diferentes efectos, ya que permite una personalización con una amplia variedad de opciones.

3.9 Audio

En la actualidad, el sonido es un complemento muy importante en las aplicaciones para computadora. Y esto es más notable si se trata de una aplicación gráfica, siendo el ejemplo más claro, el que se ofrece en los videojuegos.

De la misma forma que la calidad de los gráficos ha ido evolucionando con el paso de los años, estos cambios también se han visto reflejados en los sistemas de audio y para muestra de ellos solo basta con recordar cualquiera de los primeros videojuegos de la década de los 80's y su sonido clásico de 8-bits y compararlo con el sonido envolvente de hoy en día con capacidad para reproducir efectos 3D entre otras cosas.

Es importante remarcar que la función más importante que ofrece el sonido para estas aplicaciones es mejorar la experiencia que el usuario obtiene y el grado de inmersión en el contexto virtual que representa la aplicación.

Funcionamiento e implementación

El sistema de audio diseñado para la arquitectura tiene su funcionalidad basada en el soporte que ofrece la API de Fmod EX presentado en el capítulo anterior, como se mencionó ofrece una gran variedad de opciones para la creación y reproducción de audio interactivo.
Al igual que la mayoría de las aplicaciones gráficas, se debe contemplar el uso de dos tipos de sonido dentro de la arquitectura, los cuales son:

- Música de ambiente. Este tipo de audio es de los más importantes ya que su función es la de ir manipulando lo que el usuario entiende y siente, mientras se desenvuelve dentro del contexto virtual. Un ejemplo clásico de esto es cuando se pasa de un nivel a otro y se debe enfrentar a una nueva situación, generalmente se hace la transición a una melodía que aumenta la tensión.
- Efectos de sonido. Los efectos de sonido aunque no tienen la misma fuerza que la música de ambiente para sumergir al usuario son muy importantes ya que brindan realismo al ambiente virtual creado.

El punto más importante a considerar para el diseño del sistema de audio de nuestra arquitectura fue la sencillez en la parte de la inicialización y ejecución de cualquiera de los dos tipos de sonido explicados anteriormente.

Gracias a las características que nos ofrece la API de FMOD fue fácil de implementar, ya que en primer lugar nos brinda la posibilidad de separar las diferentes fuentes de audio en canales distintos, permitiéndonos manipular de una forma más general los efectos que se aplican a los sonidos.

Para lograr una mejor administración del sonido se contempló únicamente el uso de tres canales de audio, uno de ellos para uso específico de los efectos de sonido y los dos restantes para la música de ambiente y así facilitar efectos de transición entre dos melodías distintas.

Las características contempladas de los efectos de sonido dentro de la arquitectura no son fuera de lo común, ya que cualquier sonido puede reproducirse como tal, teniendo en cuenta que este solo se reproduce una vez, otra característica importante es que los efectos de sonido no son excluyentes por lo que se pueden reproducir varios en el mismo instante.
En el apartado de la música de ambiente la implementación requirió de un poco más de esfuerzo ya que aunque la diferencia más importante entre un efecto de sonido y la

música de fondo es que ésta última se reproduce dentro de un bucle, fue preciso diseñar el proceso por el cual se llevaría a cabo la transición de un canal a otro y que se logró de forma sencilla haciendo una atenuación y un incremento gradual del volumen en cada uno de los canales respectivos.

3.10 Animación.

La mayoría de las aplicaciones gráficas modernas giran en torno a un personaje el cual es la representación del usuario en el entorno virtual de la aplicación. Una característica muy importante de los personajes es que necesitan moverse de una forma natural y fluida.

El sistema de animación es el componente al que se le designa ésta tarea y puede dar soporte a distintas tecnologías de animación, desde técnicas muy simples que simulen el movimiento hasta técnicas más avanzadas y poderosas que con ayuda del hardware especializado logren animaciones más complejas en tiempo real.

Las principales técnicas de animación que son usadas actualmente son:

- Cel Animation. Es un tipo específico de animación tradicional en el que se usa una hoja trasparente de plástico sobre la cual las imágenes pueden ser pintadas o dibujadas sobre un fondo, sin necesidad de que este último tenga que ser redibujado en cada secuencia. El equivalente digital de este tipo de animación es una tecnología conocida como sprite animation. Esta técnica es muy usada en los juegos 2D.

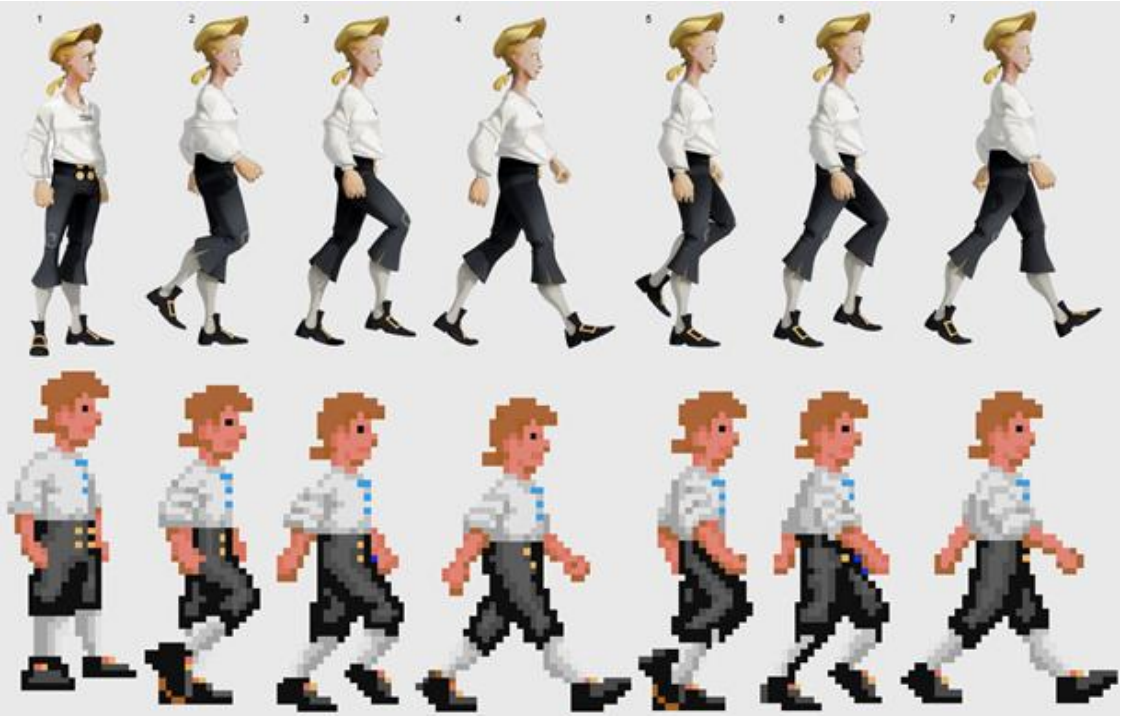

Figura 34. Ejemplo de la secuencia de una animación basada en sprites.

- Rigid Hierarchical Animation. Con la llegada de los gráficos en 3D, las técnicas basadas en el uso de sprites se perdieron y fue así que se empezaron a desarrollar las primeras técnicas de animación en 3D. En ésta representación los personajes eran modelados como una colección de piezas rígidas, las cuales estaban constreñidas una con otra de forma jerárquica, simulando las articulaciones de los huesos. El gran problema de esta técnica es el comportamiento que a menudo tiene el cuerpo del personaje debido al "agrietamiento" de las articulaciones.

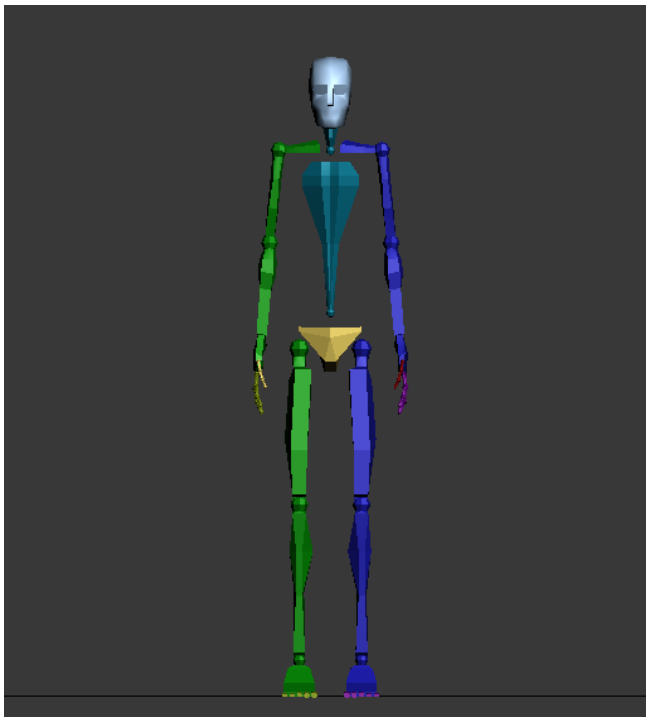

Figura 35. Ejemplo de un modelo que usa jerarquía de cuerpo rígido.

- Per-Vertex Animation. Esta técnica surge ante la necesidad de lograr un movimiento más natural y no tan rígido como el obtenido con las técnicas antes mencionadas. Una forma para lograrlo es aplicar la técnica conocida como animación por vértice, la cual consiste en que todos los vértices son animados por una artista y los datos del movimiento son exportados al motor de gráficos produciendo así la deformación deseada en la malla del modelo. Sin embargo, es una técnica con un uso intensivo de datos, desde el momento en que la información del movimiento en el tiempo es almacenada para todos los vértices que conforman la malla. Por tal motivo es una técnica que tiene poca aplicación en los juegos de tiempo real.

Skeletal Animation

La técnica elegida para implementar el sistema de animación de personajes dentro de la arquitectura fue la animación basada por esqueleto. Ésta técnica tiene los beneficios

de la animación por vértice pero además goza de las ventajas en el uso de memoria y el rendimiento eficiente de la animación jerárquica, siendo capaz de realizar aproximaciones bastante aceptables del movimiento de la piel y la ropa.

Para esta técnica es necesario un esqueleto el cual es construido de “huesos” rígidos, como sucede en la animación jerárquica. Sin embargo, en vez de renderizar las partes rígidas en la pantalla, estas permanecen ocultas, siendo visible únicamente una malla de triángulos continuos ligadas a las articulaciones del esqueleto, sus vértices se ajustan a los movimientos de las uniones y cada uno de ellos puede estar ligado a más de una unión haciendo que la malla pueda estirarse en una forma natural con los movimientos de la articulación.

Para implementar el sistema de animación se hizo uso de la biblioteca de animación Cal3d realizando únicamente la interfaz para que los usuarios puedan cargar fácilmente los archivos de los modelos y tengan acceso a las animaciones de los personajes.

En el diseño usamos 3ds Max ya que tiene el soporte de un exportador para archivos en formato Cal3d. Lo primero que se requiere es tener algún modelo de apariencia humanoide, para lo cual nosotros usamos uno creado por el artista Clawdo[19] del sitio Web Turbosquid.

[19] Para ver más trabajos de este artista, puede visitar http://www.turbosquid.com/Search/Artists/Clawdo

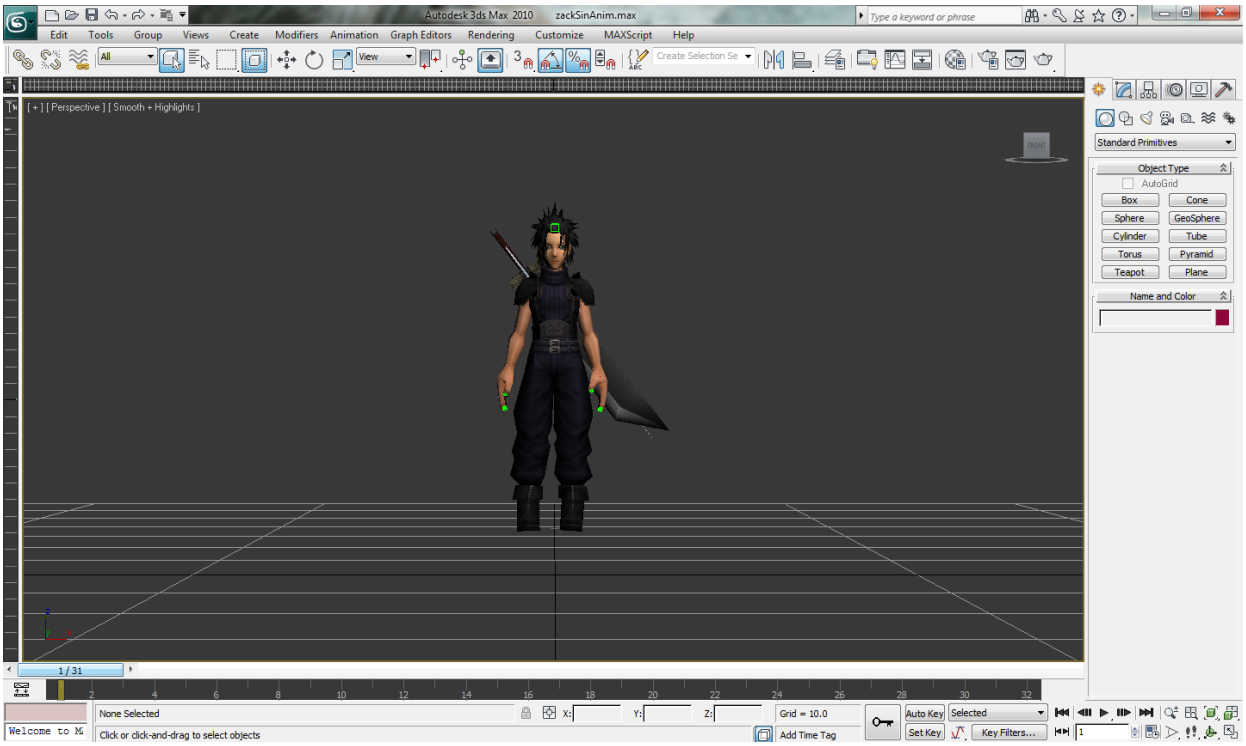

Figura 36. Modelo del personaje Zack del videojuego Final Fantasy VII.

Una vez que se tiene el modelo se llevan a cabo las animaciones, para ello usamos el entorno de desarrollo que ofrece 3ds Max, el proceso se puede encontrar de forma mas detallada en la documentación de Cal3d para el proceso de animación[20], aunque a grandes rasgos consiste en 3 pasos muy importantes:

- Crear la estructura del tipo biped que conformará el esqueleto de nuestro personaje y relacionar cada uno de los vértices del modelo inicial con los huesos de la estructura del biped.

[20] Ver http://download.gna.org/cal3d/documentation/modeling/tutorial.html

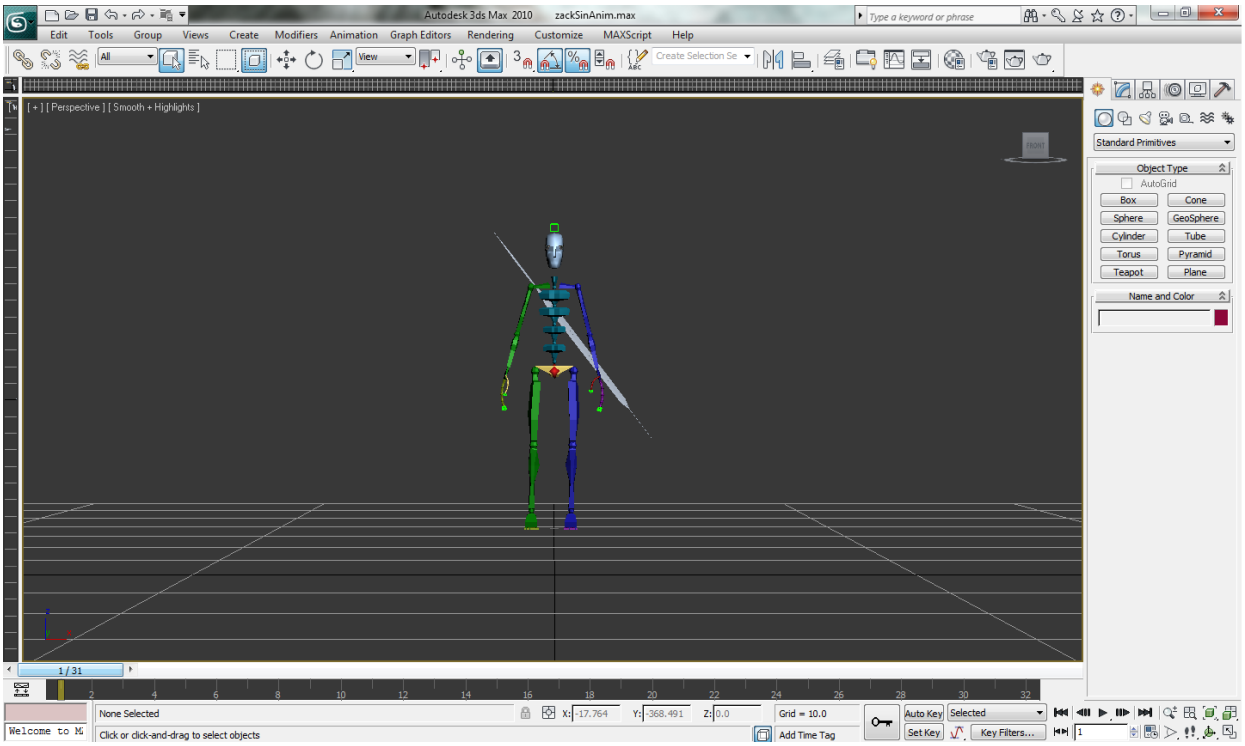

Figura 37. Estructura de tipo biped que ofrece 3dsMax para la animación basada en esqueleto.

- Cuando todos los vértices de las mallas del modelo estén relacionadas con la estructura del biped el siguiente paso consiste en animar, para ello la estructura del biped se coloca en diferentes poses a lo largo de la línea del tiempo.

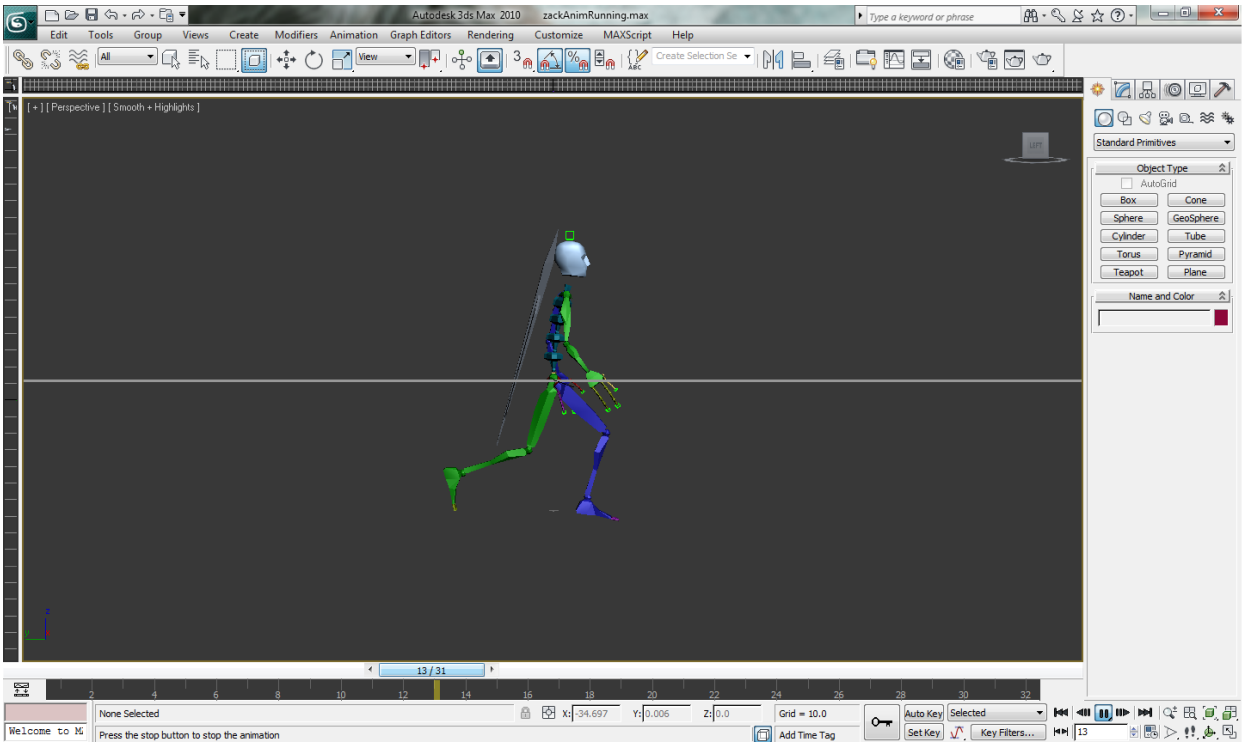

Figura 38. La animación usando 3dsMax consiste básicamente en colocar al biped en diferentes poses a lo largo de la línea del tiempo.

Figura 39. La secuencia de poses logra el efecto de la animación de caminar

- Finalmente se exportan los archivos necesarios para que la biblioteca de Cal3d pueda reproducir la animación y los cuales son indispensables para que el sistema de animación de la arquitectura funcione.

3.11 Inteligencia Artificial

La definición de Inteligencia Artificial de acuerdo a Russel y Norving (2007, p. 1) nos dice que es la creación de programas de computadoras que emulan actuar y pensar como un humano, es decir, racionalmente.

Desde el punto de vista del desarrollador de aplicaciones 3D, el principal objetivo es programar un código que permita a los personajes controlados por la computadora aparentar que toman decisiones inteligentes cuando el juego tiene múltiples opciones para una situación dada, logrando comportamientos efectivos y útiles.

Para lograrlo es indispensable considerar al menos tres procesos importantes que se deben llevar a cabo: la navegación, la percepción y la toma de decisiones. Estos puntos conforman la funcionalidad básica de un motor de Inteligencia Artificial y es indispensable que cada uno de estos procesos se retroalimenten entre sí para evaluar constantemente el estado en el que se encuentran los personajes.

A continuación se muestra un diagrama que ejemplifica la interacción que existe entre los 3 subsistemas que conforman un motor básico de IA:

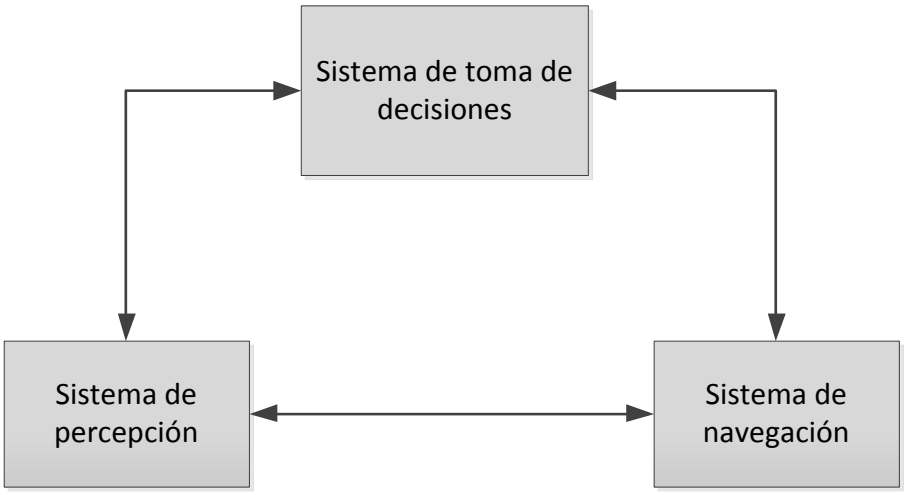

Figura 40. Capa básica de un motor de Inteligencia Artificial.

Sistema de toma de decisiones

La toma de decisiones es la tarea más importante que lleva a cabo un motor de Inteligencia Artificial. Este proceso implica que los personajes controlados por la computadora obtengan información sobre el mundo que los rodea y la procesen para responder con decisiones razonables e inteligentes. La información que puede recolectar del mundo exterior depende mucho del diseño de la aplicación desarrollada, pero su uso siempre recae en el proceso de encontrar dos tipos de soluciones para el problema planteado, una de ellas estratégica y la otra táctica.

La primera de ellas se enfoca en metas a largo plazo que involucran todo un conjunto de acciones para cumplirlas, mientras que las soluciones tácticas son a corto plazo y usualmente solo implican una acción física o una habilidad. Aunque desde el punto de vista conceptual son opuestos, en la práctica la mayoría de las aplicaciones usan ambas soluciones para lo cual aplican diferentes técnicas.

En la arquitectura desarrollada se contempla como objetivo el uso de soluciones tácticas, es decir respuestas inmediatas y que no requieren de un alto grado de inferencia por parte del motor de IA.

Sistema de percepción

El sistema de percepción es el módulo que se encarga de informar al motor de IA la situación bajo la cual se encuentra el personaje, esto significa que evalúa las cosas que se encuentran en el ambiente y bajo esa influencia será posible que nuestro personaje reaccione de alguna u otra forma.

El diseño de este módulo puede ser tan simple como se requiera, desde solo llevar un registro de las posiciones de los jugadores hasta llevar un registro completo de otros parámetros asociados como habilidad, vitalidad, cantidad de personajes, etc.

Sistema de navegación

Aunque la navegación consiste solamente en el hecho de ir de un punto A hacia un punto B, envuelve dos tareas importantes: pathfinding y evasión de obstáculos.

El pathfinding se refiere al proceso de encontrar el camino a seguir para llegar desde una posición origen a una posición destino, para ello se hace uso de una amplia variedad de técnicas, de las cuales podemos mencionar:

- Sistemas basados en rejillas: En este tipo de sistemas el mundo es dividido en una gran rejilla, ya sea de cuadrados o hexágonos y la búsqueda se lleva a cabo usando el algoritmo A*.
- Sistemas basados en campos potenciales: En este sistema el mapa también es dividido en una rejilla a la cual se le asocian valores para cada una de las regiones representando la fuerza con la que jalan o empujan a los personajes, resultando en un movimiento que va desde áreas con valores potenciales muy altos hasta áreas con valores más pequeños. Para mundos muy extensos esta técnica puede ser pre-procesada, construyendo así un diagrama de Voronoi del

espacio lo que provee una ruta bastante aceptable para llegar al destino y a la vez rápido porque las rutas son extraídas de datos existentes.

- Sistemas basados en mapas de nodos: Esta técnica es usada para mundos muy extensos donde por ejemplo los sistemas basados en rejillas ya no son tan prácticos. Al usar este método la responsabilidad recae en los diseñadores de los niveles, los cuales deben establecer una serie de puntos conectados que representen la interconexión entre los cuartos y los pasillos que conforman el nivel. La ventaja que provee este método es que el costo en memoria se reduce notablemente a diferencia de los anteriores, pero es indispensable que se haga un buen diseño del mapa de nodos. Su principal desventaja es que este sistema no contempla obstáculos dinámicos por lo que resulta indispensable la integración de un sistema de evasión de obstáculos que le permita actuar cuando eso llegue a ocurrir.
- Sistemas basados en mallas de navegación. Este sistema intenta tomar todas las ventajas que provee el sistema basado en mapas de nodos pero sin tener que crear o mantener la red de nodos, para ello usa los mismos polígonos con los que se construyó el mapa. Aunque este es un sistema mucho más poderoso puede conducir a que se creen extraños caminos si el método encargado de la construcción de la malla de navegación no es lo suficientemente inteligente o si bien los niveles fueron diseñados sin contemplar esta posibilidad.

Especificación del problema

La arquitectura enfrenta dos problemas: una de ellos es la navegación y el otro la planeación de acciones de los personajes. Para resolverlos se usan las siguientes técnicas:

Navegación basada en mapas de nodos

Esta técnica fue seleccionada dada la simplicidad con la que se puede diseñar el mapa de navegación. Dentro de esta arquitectura el diseño del mapa se lleva a cabo dentro del mismo modelo 3D del nivel, para ello se especifican los nodos mediante objetos en cuyo nombre debe llevar el indicador OPN{índice}.

Las relaciones entre los nodos se cargan de manera independiente usando un archivo XML.

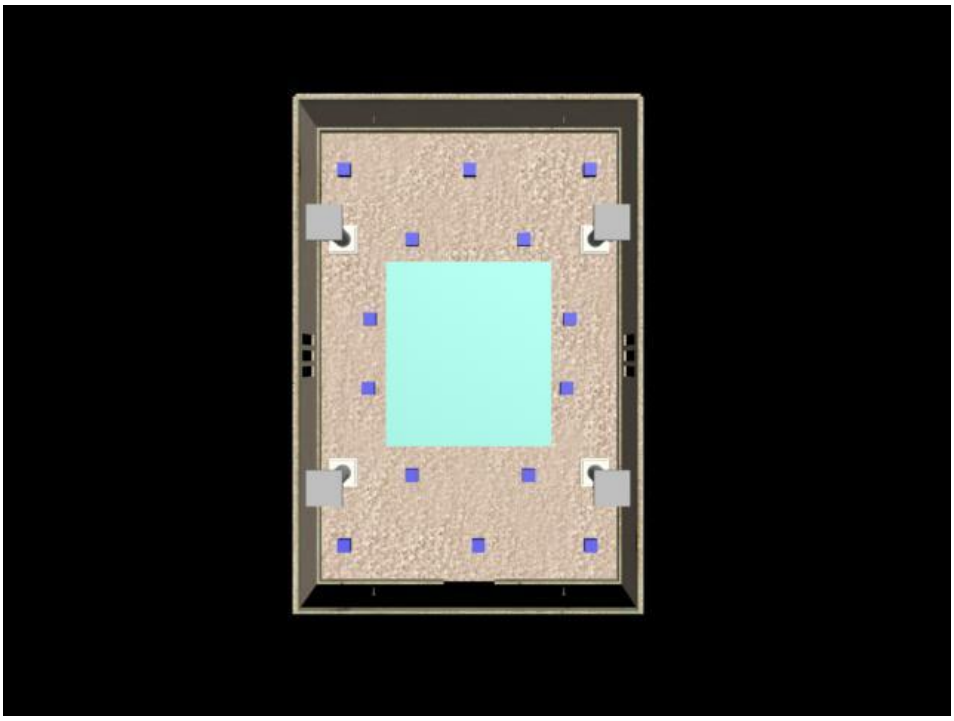

Figura 41. Mapa de nodos de uno de los niveles elaborados.

Una vez que los nodos son cargados dentro de la aplicación, se optó por usar el algoritmo A* para encontrar el camino óptimo que permita llegar de un punto A hacia un punto B dado.

Planeación de acciones usando máquina de estados

Las máquinas de estados pueden ser usadas para resolver una infinidad de problemas, desde el simple flujo del juego entre pantallas hasta las más intrincadas interacciones entre personajes. Gracias a que cuentan con características inherentes de propósito general es casi seguro que podremos aplicar una forma simple de una máquina de estados a una vasta cantidad de tareas dentro de la arquitectura, es por tal motivo que son consideradas como una de las metodologías más eficientes ya que a que a su vez separan en partes más manejables la totalidad de la funcionalidad.

La principal ventaja de un sistema basado en máquinas de estados es que es fácil de implementar. Cuenta con una naturaleza determinística inherente que permite depurarlos fácilmente ya que los errores son fáciles de replicar debido a que las conexiones entre los estados siempre son muy explícitas.

Otra característica destacable es que las máquinas de estado pueden ser fácilmente expandibles si es que durante el desarrollo del proyecto se pensara en incrementar su complejidad. Es por ello que el diseño del diagrama de estados debe desarrollarse teniendo esta posibilidad en mente porque de no ser así, esto se vuelve una gran desventaja convirtiendo el diagrama de estados en algo sumamente complejo y haciendo que el número de transiciones crezca exponencialmente y sea complicado determinar la prioridad de las acciones a resolver.

Otra desventaja que se presenta en este tipo de modelo es la oscilación de los estados. Esto ocurre cuando la percepción de datos que separa uno o varios estados es inconsistente, esto da lugar a solapamientos en los valores percibidos para llevar a cabo una transición e indudablemente conducirá a un efecto de estados vacilantes.

Como se había descrito anteriormente, el uso de las máquinas de estados para la planeación de acciones dentro de la arquitectura será implementada en un nivel básico. No se requiere un proceso de inferencia muy complejo para que los personajes realicen

una acción. De esta forma algunas de las acciones contempladas para los personajes son realmente sencillas como la ilustrada en la siguiente figura:

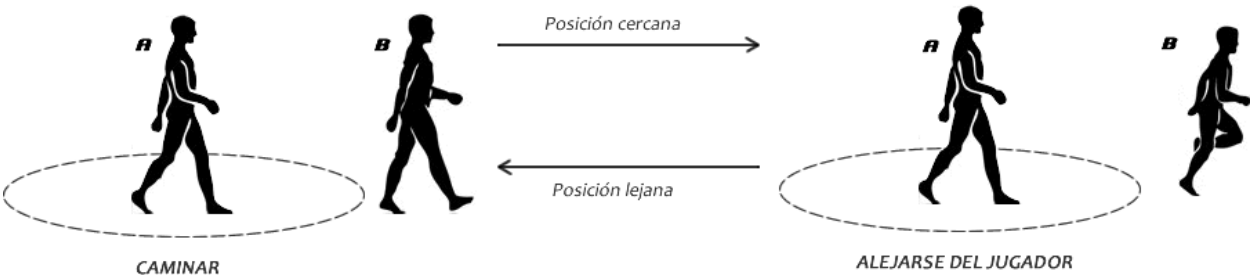

Figura 42. Ejemplo básico de una transición entre estados.

4. Conclusiones

4. *Conclusiones*

Durante la carrera como estudiantes, investigamos, leemos y aprendemos acerca de muchos temas, los cuales nos ayudan en nuestra formación profesional continua, pero es con un trabajo de investigación sobre algún tema que nos apasione cuando realmente nos ponemos en una posición de autoconocimiento y vemos la necesidad y el gusto por aprender más, y es que a lo largo de este proceso de desarrollo de nuestra tesis hemos consolidado e incrementado nuestra comprensión sobre este tópico.

El desarrollo de la arquitectura diseñada fue un trabajo arduo, de muchas horas de investigación y programación, pero sobre todo de maduración como futuros ingenieros.

Para nosotros es una pasión el programar y observar cómo se van dando los resultados esperados es una gran satisfacción y más tratándose del área de gráficos por computadora ya que ha sido el área en la que nos hemos especializado.

Para finalizar, es importante mencionar, que este trabajo de tesis dio como resultado en su implementación una aplicación gráfica en 3D que aplica todos los conceptos teóricos que conforman nuestra arquitectura diseñada y que a continuación se enumeran:

- Detección de colisiones
- Efectos especiales con partículas
- Efectos especiales con shaders
- Animación por esqueleto
- Carga y renderizado de geometría 3D
- Carga de audio
- Utilización de interfaces de usuario.

5. Anexos

Tabla de figuras

Bibliografía

C++

[1] Stroustrup, B. (1997). *The C++ Programming Language* (3ra ed.). EEUU. Addison-Wesley.

[2] C++ Information: A brief description. Consultado el 25 de abril de 2011. http://www.cplusplus.com/info/description/

STL

[3] Standard Template Library (STL). Consultado el 10 de abril de 2011. Universidad de Granada, Página del Departamento de Ciencias de la Computación e Inteligencia Artificial. http://decsai.ugr.es/~jfv/ed1/c++/cdrom4/paginaWeb/stl.htm

[4] Introduction to the Standard Template Library. Consultado el 12 de abril de 2011. http://www.sgi.com/tech/stl/stl_introduction.html

OpenGL

[5] OpenGL Overview. Consultado el 12 de abril de 2011. http://www.opengl.org/about/overview/

[6] OpenGL Shading Language. Consultado el 12 de abril de 2011. http://www.opengl.org/documentation/glsl/

Cal3D

[7] Desmecht, L., Dachary, L. & Heidelberger, B. The Cal3D User's Guide. Consultado el 12 de abril de 2011. http://download.gna.org/cal3d/documentation/guide/

[8] Cal3D. Consultado el 12 de abril de 2011. http://www.openden.com/opensource/3D+Graphics,+Animation,+Rendering/product/Cal3D

FMOD

[9] FMOD Ex Programmer's API. Consultado el 12 de abril de 2011. http://www.fmod.org/index.php/products/fmodex

TinyXML

[10] Thomason, L. , Berquin, Y. & Ellerton, A. TinyXml Documentation 2.6.0. Consultado el 12 de abril de 2011.
http://www.grinninglizard.com/tinyxmldocs/index.html

Motor de videojuegos y animación

[11] Gregory, J. (2009). *Game Engine Architecture*. EEUU. A K Peters, LTD.

Modelado geométrico

[12] Manzanilla, W. Modelado geométrico - Antecedentes. Consultado el 12 de abril de 2011.
http://www.wikilearning.com/curso_gratis/modelado_geometrico-antecedentes/29040-1

Texturizado

[13] Wolfe, R. Teaching Texture Mapping Visually. Consultado el 12 de abril de 2011.
http://www.siggraph.org/education/materials/HyperGraph/mapping/r_wolfe/r_wolfe_mapping_1.htm

[14] Section One - What's a UV map?. Consultado el 12 de abril de 2011.
http://waylon-art.com/uvw_tutorial/uvwtut_02.html

Normal Mapping

[15] Kreuzer, J. Object Space Normal Mapping with Skeletal Animation Tutorial. Consultado el 12 de abril de 2011.
http://www.3dkingdoms.com/tutorial.htm

[16] Cloward, B. Creating and using normal maps. Consultado el 12 de abril de 2011.
http://www.bencloward.com/tutorials_normal_maps1.shtml

Sistema de partículas

[17] Jaegers, K. (2009). *XNA 4.0 Game Development by Example.* EEUU. Packt Publishing

[18] Point Sprites. Consultado el 12 de abril de 2011.
http://www.codesampler.com/oglsrc/oglsrc_6.htm

[19] 2.5D. Consultado el 12 de abril de 2011.
http://en.wikipedia.org/wiki/2.5D

Light Mapping

[20] Light Mapping with Textures (Direct3D 9). Consultado el 12 de abril de 2011. http://msdn.microsoft.com/en-us/library/bb174695%28v=vs.85%29.aspx

PDI

[21] Van Wersch, R. HDR Rendering Sample. Consultado el 12 de abril de 2011. http://www.xnainfo.com/content.php?content=28

Audio

[22] El sonido en los videojuegos [1ra parte] - El jugador, la música y los diálogos. Consultado el 12 de abril de 2011. http://www.hispasonic.com/blogs/sonido-videojuegos-1ra-parte-jugador-musica-dialogos/599

Inteligencia Artificial

[24] Schwab, B. (2004). *AI Game Engine Programming. EEUU.* Charles River Media, Inc

[25] Russel, S. & Norving, P. (2003) *Artificial Intelligence: A Modern Approach.(2da ed.). EEUU.* Prentice Hall.