DESARROLLO DE UN MOTOR DE VIDEOJUEGOS Y SU UTILIZACIÓN EN LA CREACIÓN DE UN VIDEOJUEGO

CARLOS ANDRÉS ROCHA SILVA
NITAE ANDRÉS URIBE ORDOÑEZ

UNIVERSIDAD AUTÓNOMA DE BUCARAMANGA
FACULTAD DE INGENIERÍA DE SISTEMAS
SISTEMAS DE INFORMACIÓN E INGENIERÍA DE SOFTWARE
BUCARAMANGA
2008

DESARROLLO DE UN MOTOR DE VIDEOJUEGOS Y SU UTILIZACIÓN EN LA CREACIÓN DE UN VIDEOJUEGO

CARLOS ANDRÉS ROCHA SILVA
NITAE ANDRÉS URIBE

Trabajo de grado para optar por el título de
Ingeniero de sistemas

Director
Phd. EDUARDO CARRILLO ZAMBRANO

UNIVERSIDAD AUTÓNOMA DE BUCARAMANGA
FACULTAD DE INGENIERÍA DE SISTEMAS
SISTEMAS DE INFORMACIÓN E INGENIERÍA DE SOFTWARE
BUCARAMANGA
2008

Nota de aceptación:

_____

_____

_____

_____

_____

_____

_____

_____
Firma del director

_____
Firma del evaluador

_____
Firma del evaluador

Bucaramanga 30 de mayo de 2008

AGRADECIMIENTOS

De Carlos

A Diana: por mostrarme que el mundo no es como creí que era.

A Diana: por mostrarme que el mundo puede ser como creí que era.

A Nitae: por ser un compañero de armas en la lucha por el sueño.

A Andrés: por luchar por ser la persona que quiere ser.

A mi mamá: por nunca medir el amor que da.

A mi papá: por luchar en nombre del amor.

A mi hermano: por ser una inspiración para mí.

A Yoan: por mostrarme el otro lado de la mente.

A Juan Manuel: por enseñarme que todo puede lograrse.

A Kentaro: por enseñarme la palabra "sueño".

A Eduardo: por ser la viva representación de una vida consumada.

A Stefan: por mostrarnos que nuestra ambición era alcanzable.

A los videojuegos: por ser la combinación de todo lo que amo del arte, la ciencia y la filosofía.

Gracias, porque gracias a todos ustedes soy lo que soy,
y seré quien seré.

CONTENIDO

pág.

LISTA DE FIGURAS

pág.

LISTA DE ANEXOS

pág.

GLOSARIO

API: Es el conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta librería para ser utilizado por otro software como una capa de abstracción.

API GRÁFICO: Es un API que ofrece prestaciones para desarrollos gráficos digitales a programadores y diseñadores. En el ámbito de los videojuegos, los API gráficos mas conocidos son Direct3D y OpenGL.

BUFFER: Es un espacio de memoria, en el que se almacenan datos para evitar que el recurso que los requiere, ya sea hardware o software, se quede en algún momento sin datos.

CACHE: Es un conjunto de datos duplicados de otros originales, con la propiedad de que los datos originales son costosos de acceder, normalmente en tiempo, respecto a la copia en el caché. Cuando se accede por primera vez a un dato, se hace una copia en el caché; los accesos siguientes se realizan a dicha copia, haciendo que el tiempo de acceso medio al dato sea menor.

DIRECT3D: El objetivo de esta API es facilitar el manejo y trazado de entidades gráficas elementales, como líneas, polígonos y texturas, en cualquier aplicación que despliegues gráficos en 3D, así como efectuar de forma transparente transformaciones geométricas sobre dichas entidades. Direct3D provee también una interfaz transparente con el hardware de aceleración gráfica.

DIRECTINPUT: Módulo perteneciente a DirectX utilizado para procesar datos del teclado, ratón, joystick y otros controles para juegos. DirectInput es una librería para procesar datos del teclado, ratón, joystick y otros controles para

juegos. También provee un sistema de mapeo que permite especificar las acciones del juego que originarán los botones y los ejes de los dispositivos de entrada.

DIRECTSOUND: DirectSound es un componente de software de la biblioteca de DirectX, proveído por Microsoft, que reside en una computadora con el sistema operativo Windows. Proporciona una interfaz directa entre las aplicaciones y los drivers de la tarjeta de sonido, permitiendo a las aplicaciones producir sonidos y música.

DIRECTX: DirectX es una colección de APIs creadas para facilitar tareas relacionadas con la programación de juegos en la plataforma Microsoft Windows. Las APIs son: Direct3D, DirectInput, DirectSound, DirectGraphics, DirectMusic, DirectPlay.

JOYSTICK: Un joystick o palanca de mando es un dispositivo de control de dos o tres ejes que se usa desde una computadora o videoconsola al trasbordador espacial o los aviones de caza, pasando por grúas.

LINKER / ENLAZADOR: Es un programa que toma los ficheros de código objeto generado en los primeros pasos del proceso de compilación, la información de todos los recursos necesarios (biblioteca), quita aquellos recursos que no necesita, y enlaza el código objeto con su(s) biblioteca con lo que finalmente produce un fichero ejecutable o una biblioteca.

OPENGL: Es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

PÍXEL: Es la menor unidad en la que se descompone una imagen digital, ya sea una fotografía, un fotograma de vídeo o un gráfico.

VIDEOJUEGO: Un videojuego es un programa informático, creado en un principio para el entretenimiento, basado en la interacción entre una o varias personas y un aparato electrónico llamado consola que ejecuta el videojuego. Estos recrean entornos y situaciones virtuales en los cuales el jugador puede controlar a uno o varios personajes (o cualquier otro elemento de dicho entorno), para conseguir uno o varios objetivos por medio de unas reglas determinadas.

RESUMEN

Un videojuego se compone de diversos elementos tales como modelos, gráficas, guiones, música, y sonido, elementos que se complementan para generar un producto multimedia cuya finalidad es la de ofrecer una experiencia interactiva que entretenga a los usuarios. Esta es una industria de millones de dólares con alta complejidad y costos de producción, y la reutilización de herramientas de desarrollo se ha hecho indispensable en su crecimiento. Con este propósito nacen los motores de videojuegos, aumentando la eficiencia y capacidad de los videojuegos. Sin embargo, el costo de los motores profesionales excede el presupuesto de empresas en crecimiento, y el desarrollo de herramientas libres es muy limitado, careciendo de los elementos para crear paquetes multimedia integrales. Se ha propuesto el desarrollo de un motor de videojuegos llamado el CNEngine, con licencia Lesser General Public Licence (LGPL) compuesto por todos los módulos necesarios para garantizar su autosuficiencia en la creación de videojuegos, de manera que sirva como base para una industria en crecimiento en la región y en el mundo, e impulse futuros desarrollos similares. Los videojuegos deben cumplir con plazos y estándares en una industria competitiva, y existe poca documentación respecto a pautas o patrones de diseño que permitan guiar estos tipos de desarrollo, por lo que se ha planteado una investigación enfocada en el estado del arte de la ingeniería de software para videojuegos, de manera que facilite la creación de un producto que cumpla con todos los requerimientos del diseñador.

Palabras clave: CNEngine, Ingeniería de software, Lesser General Public Licence, Motor de videojuegos, Videojuego.

Línea de investigación: Sistemas de información e Ingeniería de Software.

0. INTRODUCCIÓN

El desarrollo de videojuegos es una industria multimillonaria, de estándares demandantes. Hoy en día se requieren equipos compuestos por profesionales de diversas áreas para su creación, buscando entretener un usuario con mayores expectativas. El desarrollo de estos videojuegos se hace más complejo, en especial para desarrolladores independientes que carecen del presupuesto para adquirir equipos de producción profesionales. Desarrolladores cuyo entorno es Latinoamérica, teniendo la limitante adicional del idioma, puesto que la mayoría de estos productos (motores libres) se realizan en países como Francia, Alemania y Japón.

El principal producto facilitador en la creación de los videojuegos es el motor, el cual permite la implementación del videojuego diseñado. En el caso de los motores libres existe el inconveniente de no permitir la creación de una aplicación interactiva integral, puesto que solo se encuentran disponibles algunas de las funcionalidades intrínsecas en un videojuego. Es así como se llega a la necesidad de desarrollar un motor que contenga todos los requisitos fundamentales de un programador o desarrollador de videojuegos para que puedan desarrollar productos de calidad sin requerir de un alto presupuesto para pagar y tener acceso a esta tecnología. Estos requisitos abarcan la posibilidad de poder implementar en las aplicaciones simulaciones de acciones y reacciones con el entorno que simulen la realidad (físicas), la visualización del entorno en tres dimensiones, la posibilidad de movimiento por medio de dispositivos de entrada, y la posibilidad de reproducción de sonidos; elementos que deben reaccionar en tiempo real a medida que el usuario “juega”.

En este proyecto se planteó dar solución a este problema y permitir a través de una licencia libre, la distribución de un motor de desarrollo completo, además de una demostración de las capacidades del mismo con un videojuego que permitirá interacción con el usuario a través de dispositivos como el teclado y el mouse, con realimentación gráfica y auditiva en tres dimensiones, simulando propiedades físicas de los objetos.

Adicionalmente, se encontró que la creación de videojuegos a nivel mundial no siempre cuenta con la rigurosidad necesaria para cumplir con las fechas de entrega del producto, aun en compañías consolidadas. Este es un factor predominante cuando se trata de desarrolladores independientes y estudios en formación. Por tanto se identificó la necesidad de crear un documento que resuma el estado del arte de las metodologías de desarrollo de videojuegos, de manera que se entendieran los diversos estándares usados y su relación con las metodologías más tradicionales, y en que medida estas son adecuadas. Como beneficio adicional, se presenta el documento como una guía para los desarrolladores que decidan utilizar el motor para sus productos, y permitirles que tengan lineamientos utilizados internacionalmente y cuya efectividad se haya comprobado en niveles satisfactorios.

Es importante resaltar que debido a los altos estándares competitivos que desea promover la Facultad de Ingeniería de Sistemas de la UNAB, existe la posibilidad de crear grupos de investigación estudiantiles denominados “semilleros de investigación”, los cuales se enfocan en temas de interés para los mismos estudiantes, puesto que es una propuesta que nace de su iniciativa.

Es en este contexto en el que se gestó la idea de un semillero enfocado a la investigación sobre la concepción y el desarrollo de videojuegos, tema de gran interés en la actualidad el cual es guiado por los estudiantes que desarrollan este proyecto. A través de la interacción con distintas facultades, se culminó un proyecto denominado “Proyecto Alfa” el cual es un videojuego en dos dimensiones enfocado al entretenimiento que obtuvo gran difusión a nivel regional. Luego con el desarrollo de “Misión Leonidas”, un demo en tercera dimensión desarrollado con la combinación de varios motores (gráficos, de sonido, y físicas), el semillero se consolidó como una propuesta seria de investigación. Hoy en día trabaja en conjunto con el “Grupo de investigación en Tecnologías de la Información” (GTI), grupo clase A clasificado por de Colciencias, desarrollando propuestas de investigación alrededor de los videojuegos.

Con estos precedentes: un gran apoyo de la universidad, satisfactorios resultados en la producción y en el desarrollo, etc. La opción de elaborar un proyecto que se encaminara a la creación de un software que sirviera de base para la elaboración de videojuegos se concibió como un proyecto factible y como un aporte que permanecerá en la Universidad y será de utilidad para futuros proyectos. El videojuego se usó para tener una idea de las funcionalidades que el motor permitiría a los desarrolladores de la UNAB, dejando con ambas cosas (el motor y un ejemplo de sus capacidades), un legado que perdurará y podrá beneficiar futuros desarrollos similares.

1. REFERENCIAS CONCEPTUALES

A continuación se presentan una serie de conceptos que son de gran utilidad para la comprensión adecuada del proyecto. Presentan un alto nivel de detalle de manera que la lectura del actual documento sea fácilmente comprensible.

1.1 GENERALIDADES DE LOS VIDEOJUEGOS

1.1.1 ¿Que es un videojuego? A continuación se presentan diversas definiciones de varias fuentes:

"Video games […] are a form of artistic expression involving creation from script writers, designers and directors."[1]

> "Un videojuego es un programa informático, creado expresamente para divertir, basado en la interacción entre una persona y un aparato electrónico donde se ejecuta el videojuego. Estos recrean entornos virtuales en los cuales el jugador puede controlar a un personaje o cualquier otro elemento de dicho entorno, para conseguir uno o varios objetivos por medio de unas reglas determinadas"[2].

La Real Academia de la lengua española define un videojuego como: "Dispositivo electrónico que permite, mediante mandos apropiados, simular juegos en las pantallas de un televisor o de un ordenador"[3].

---

[1] ENTREVISTA a Renaud Donnedieu de Vabres, ministro frances para el diario "The New York Times", "For France, Video Games Are as Artful as Cinema", noviembre 6, 2006.

[2] WIKIPEDIA: la enciclopedia libre. Videojuego. 2002. En: http://es.wikipedia.org/wiki/Videojuego.

Basado en las concepciones anteriores, se ha propuesto una definición que abarca tanto los conceptos plasmados anteriormente con la experiencia de los autores:

Un videojuego es un tipo de software diseñado para entretener a sus usuarios (jugadores) por medio de un entorno interactivo virtual, en el cual el jugador puede controlar o entablar distintos tipos de relaciones con diferentes personajes, ya sean controlados por inteligencia artificial o por otros jugadores. Con el fin de llevar a cabo esta tarea, se usan dispositivos de video y de audio que permiten manifestarle al usuario lo que sucede en el entorno virtual; y como él, a través de interfaces de entrada como controles especiales para mover el personaje o simplemente con el mouse, puede modificar estos eventos que suceden simultáneamente a sus decisiones y acciones. Este tipo de interactividad basada en la creatividad de un grupo de personas lideradas por un diseñador general, ha conllevado a muchas personas a definir a los videojuegos como una nueva forma de arte, que permite una mayor comunicación con la audiencia al permitirles interactuar con la obra en sí.

Su principal finalidad (en la mayoría de los casos) es entretener, pero además puede contener una gran variedad de propósitos adicionales o de distintos enfoques, ya sean proporcionar en la persona un espacio para la reflexión, para la generación de emociones de toda índole, para educarlos respecto a temas particulares. Incluso pueden servir de medios de entrenamiento para situaciones de la vida cotidiana (simulaciones).

---

[3] REAL ACADEMIA DE LA LENGUA ESPAÑOLA. Videojuego. En: http://buscon.rae.es/draeI/SrvltConsulta?TIPO_BUS=3&LEMA=videojuego

1.1.2 Clasificación de los videojuegos. [4] [5] Los videojuegos se clasifican de distintas maneras, las más comunes son por su enfoque gráfico, por el tipo de experiencia que el juego proporciona, por su propósito, por el número de jugadores que lo pueden usar, etc.

Teniendo en cuenta su enfoque gráfico los videojuegos se clasifican en dos grandes grupos, los videojuegos dos dimensiones (2D) y los videojuegos en tres dimensiones (3D). El primer grupo se caracteriza por sus gráficos planos y la limitación de tener solo 4 posibles movimientos en su forma pura (arriba, abajo, izquierda y derecha), sin embargo se este formato se ha modificado de manera que los juegos tienen una componente tridimensional, aunque esta no permita diferenciación por parte del jugador. Esto se logra añadiendo profundidad a los objetos en el videojuego, lo cual se logra utilizando la perspectiva y superposición de las imágenes.

El segundo enfoque se refiere a los videojuegos en tres dimensiones (3D), los cuales poseen gráficos más realistas debido a su manejo tridimensional, permitiendo que la inmersión de los jugadores sea mucho más profunda y cada vez más cercana a la realidad. Estos videojuegos tienen menores limitaciones de movimiento, pues cuentan (normalmente) con un horizonte de 360 grados, así como interacción con los objetos dentro del videojuego.

Recientemente distintas propuestas de videojuegos han llevado a la combinación de estos enfoques, permitiendo cambiar de un mundo en tercera dimensión a un mundo en segunda dimensión, siendo el juego Super Paper Mario el ejemplo más conocido.

---

[4] ROLLINGS, Andrew, ADAMS, Ernest. Andrew Rollings and Ernest Adams on Game Design. New Riders Publishing, 2003, Part II: the genres of games. p. 152

[5] PEDERSEN, Roger E. Game design foundations, Wordware Publishing. 2003. Chapter 5: Game Genres. p. 231

La clasificación por el número de jugadores los diferencia en los de un jugador y los multijugador. Los de un jugador se caracterizan por que solo una persona puede jugarlo en un momento determinado, esto implica que no puede tener interacción con otras personas en tiempo real. Los videojuegos multijugador permiten que dos o más jugadores interactúen en tiempo real o de manera que de forma simultánea tengan una interfaz gráfica, lo cual hace mucho más realista la experiencia de interacción.

La clasificación según su modo de juego, o por el tipo de experiencia que proporcionan al jugador consiste en ubicarlo en una o varias de los siguientes "géneros":

- **Acción.** Su función primordial es medir las capacidades del jugador por medio de tiempos de reacción y coordinación mano-ojo sometido a grandes factores de presión. Normalmente tienen objetivos simples y tienden a mantener una relación inversa entre complejidad y velocidad (cosa que recientemente ha cambiado). Los escenarios pueden contener elementos de aventura y de plataforma que permitan en el usuario distintas experiencias combinadas, esto es debido a que, siendo un género tan popular, ha adoptado distintos enfoques para diversificarse.

- **Aventuras.** Son juegos en los que el protagonista debe interactuar con los NPC (**non playable characters**) los cuales le permiten al jugador avanzar a través de conversaciones y exploración por grandes escenarios, resolviendo acertijos o resolviendo problemas. Los videojuegos de aventura son una historia interactiva acerca de un personaje controlado por el jugador.

- **Deportivos.** Es uno de los pocos géneros que permiten contraste con la vida real, debido a que están basados en experiencias de los deportes reales. Se divide a si mismo en subgéneros como el automovilismo, el boxeo, las peleas, etc. Pueden contener elementos irreales manteniendo las reglas básicas del juego. La idea primordial es permitirle al usuario una sensación de estar realizando el deporte desde su hogar, y generar el sentido de competencia, ya sea contra el mismo o contra la IA.

- **De disparos.** Son un subgénero de los juegos de acción. Se caracterizan principalmente por la interacción con el ambiente a través de armas de fuego. La trama puede centrarse en objetivos muy sencillos tales como eliminar a todos los enemigos mientras se sale de un recinto, aunque recientemente en grandes esfuerzos por mejorar la experiencia las tramas han llevado a estos juegos a mezclarse con otros géneros.

- **Educativos.** Juegos cuyo objetivo es transmitir al jugador algún tipo de conocimiento o reforzar alguno conocido. Su mecánica puede abarcar cualquiera de los otros géneros de manera que el jugador se pueda sumergir en la experiencia, manteniendo el énfasis en el factor educativo.

- **Estrategia.** Se fundamentan en la utilización de estrategias y planeación. Son juegos donde el diseñador crea reglas y objetivos, pero depende del jugador que decisiones tomar dentro del rango de posibilidades para vencer (intelectual y estratégicamente) al adversario. Pueden encontrarse con enfoques basados en turnos o en tiempo real. Normalmente permiten interacción con otros usuarios del juego.

- **De acertijos.** (“puzzles” por su nombre en inglés) Son juegos normalmente basados en juegos de mesa, permitiendo que el jugador pueda probar sus conocimientos con una inteligencia artificial especialmente diseñada para este tipo de retos. Es un género muy amplio que puede incluir juegos conceptuales o abstractos, pero de gran reto para el raciocinio del usuario.

- **Rol.** Como su nombre lo indica son juegos en los que el personaje principal toma un rol protagónico en una historia en la que más personajes se unen por un objetivo común y se encaminan en una travesía para conseguir tesoros, explorar vastos terrenos, conseguir experiencia y habilidades para derrotar más fácilmente a los enemigos que se atraviesan en su camino o inclusive tener una oportunidad en contra de un terrible mal. La trama normalmente es larga y compleja, pudiendo contener muchos elementos de fantasía o ciencia ficción que hacen de la experiencia algo mucho más significativo para el usuario pues trata de presentar opciones poco factibles. Sin embargo su trama puede suceder también en un entorno urbano actual, cuyo énfasis no sea la fantasía sino la exploración. Recientemente se han enfocado a otorgarle libertad al usuario al poder generar su propia historia con una columna vertebral base (una historia principal) que los usuarios pueden modificar de acuerdo a las decisiones que tomen y a las acciones que realicen, permitiendo de esta manera experiencias completamente diferentes para dos personas que estén en el mismo juego. Recientemente el uso del Internet ha enriquecido la experiencia al permitir a jugadores de todo el mundo explorar un universo persistente al mismo o en distinto tiempo, pero sensible a todas las acciones que realicen, trayendo consecuencias positivas o negativas.

- **Simulación.** Son juegos basados en el modelado de ciertas circunstancias reales que le permiten al jugador una experiencia similar al mundo real pero en la

seguridad de un entorno virtual. Muchos de estos juegos se utilizan para entrenamiento en el mundo de los negocios, en las prácticas militares o en la readaptación de personas con problemas cognitivos para su reintegro total en la sociedad. Muchas de las situaciones simuladas serian extremadamente costosas fuera de un simulador, y su realismo ha llegado a un punto tal que entrenamientos profesionales dependen ampliamente del uso de simuladores (por ejemplo, en simuladores de vuelo).

1.1.3 Elementos en el desarrollo de un videojuego.[6] El desarrollo de un videojuego es un proyecto complejo, en especial si se pretenden utilizar las últimas tecnologías gráficas, simulaciones de físicas, inteligencia artificial, etc. Su creación se equipara con la filmación de una película en cuanto a la magnitud del proyecto.

Debido a esto, en el desarrollo de un videojuego están implicadas una gran cantidad de personas, con una marcada diferenciación de cargos, pero de significativa importancia. Se describirán generalizadamente las ramas de este desarrollo.

En el desarrollo de un videojuego sobresalen 3 tipos de labores que serian los pilares del mismo, estos son la programación, el diseño y el arte digital.

La programación comprende todo lo que tiene que ver con el código del juego, tanto del motor como de la implementación del mismo. Este es el área de los ingenieros de sistemas, donde deben aplicar técnicas de ingeniera de software para hacer efectiva y eficientemente su trabajo. El segundo elemento es el diseño del videojuego: este comprende lo que es el trabajo intelectual, el de imaginar el

[6] ZERBST, Stefan; DUVEL, Oliver. 3D Game Engine Programming. Thomson Course Technology, 2004. Game Developers Series. p.10

videojuego. Normalmente la gente que realiza esta labor tiene extensa experiencia como desarrolladores o jugadores. Es análogo a escribir el guión de una película. El tercer elemento en el desarrollo de un videojuego es el arte gráfico y auditivo, que comprende en primera instancia todo lo que puede observarse, la forma de "ver" el mundo virtual que se ha creado. Por último se encuentra la parte auditiva, otro de los principales focos de atención de la interfaz en los videojuegos. Estos se pueden dividir en efectos de sonido, música del juego, música de los menús, y en general cualquier tipo de ambientación sonora que le pueda brindar al usuario una experiencia enriquecedora. Normalmente hay personas especializadas en cada área y trabajan en conjunto con los programadores para incluir su labor en el juego.

1.2 MOTORES DE VIDEOJUEGOS [7] [8]

Para crear un videojuego normalmente se necesita desarrollar, antes de crear el juego en sí, un conjunto de aspectos computacionales que definan cuales serán los comportamientos de cada uno de los elementos dentro del juego, cómo se verán y cómo reaccionaran ante las diferentes decisiones que el jugador decida de que forma afectan al objeto en cuestión. Este es un proceso largo que puede tomar el mismo o mayor tiempo que crear el videojuego, y se enfoca principalmente en la programación, debido a que su desarrollo es principalmente técnico y conlleva una metodología rigurosa propia de desarrollos de software clásico. Es a este conjunto de estándares ("núcleo" del videojuego) lo que comúnmente se denomina el "motor".

[7] ZERBST, Stefan; DUVEL, Oliver. 3D Game Engine Programming. Thomson Course Technology, 2004. Game Developers Series. p. 5

[8] PEDERSEN, Roger E. Game design foundations, Wordware Publishing. 2003. p 34.

El conjunto de estándares que se enfocan al comportamiento realista (o irreal, dependiendo del juego) del motor es conocido como un *motor de física*, el cual se define como el contenido en código donde se elabora todo el modelado matemático que le dará a las reacciones de los objetos, incluyendo las colisiones entre ellos, una perspectiva realista (mucho o poco) de manera que el usuario se identifique con la forma como el juego quiere presentarle un mundo virtual con sus propias reglas.

El motor del juego tiene además un motor de renderización, pues los motores de videojuegos normalmente están conformados por dos motores, el de física y el de renderización, ambos totalmente independientes. Sin embargo, deben trabajar juntos para lograr hacer que la aplicación funcione, creando así un videojuego. El motor de renderización se encarga de hacer realista el juego, pero desde la parte grafica, haciendo que los modelos 3D (los personajes, los objetos y los escenarios) sean mas suaves y detallados.

1.2.1 Motor de física.[9] Como ya se ha mencionado, el motor de física se encarga de que el juego tenga leyes de la naturaleza semejantes a las nuestras, por ejemplo la gravedad, las colisiones elásticas e inelásticas, la fricción, el movimiento de objetos debido al viento (incluyendo al mar), etc.; pero debido a que el videojuego está enfocado a audiencias que esperan ver reacciones directas de sus acciones, estas leyes deben físicas ser modificadas para generar un sentimiento de "hiperealismo", y de esta forma permitir en el usuario mayor interés en el mundo virtual. La tecnología usada generalmente son lenguajes como C y C++, debido a su facilidad de desarrollo para programar los métodos numéricos encargados de hacer los cálculos matemáticos sobre los objetos en el videojuego. Estos modelamientos deben poder aproximar rápidamente los

[9] MILLINGTON, Ian. Game physics engine development. Morgan Kaufmann Publishers. 2007. p. 12

valores requeridos por el juego, tienen que mantener un balance muy delicado entre eficiencia y precisión, para poder llevar a cabo los cálculos en tiempo real.

El motor de física debe poder soportar comportamientos “rígidos” y “suaves” de los objetos. Los comportamientos rígidos se usan para objetos poco elásticos y densos, como balas, paredes, espadas, etc. Los comportamientos suaves los presentan objetos compuestos por partículas, no muy densos, como el agua, las nubes, las explosiones y en general efectos de partículas, este último tipo requiere un alto nivel de procesamiento, ya que cada partícula tiene su propio comportamiento.

1.2.2 Motor de renderización.[10] Se encarga de la parte grafica del videojuego. Es código mediante el cual se visualizan los elementos en la pantalla. Para realizar esta función existen dos apis estándares en el desarrollo de aplicaciones graficas: opengl (open graphic library) y Direct3D.

OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Fue desarrollada por Silicon Graphics Inc en 1992, como su nombre lo dice, es de carácter libre, esto quiere decir que se puede usar para desarrollar sin tener que obtener una licencia.

Direct3D es parte de DirectX, una API propiedad de Microsoft disponible tanto en los sistemas Windows de 32 y 64 bits, como para sus consolas Xbox y Xbox 360, usada en la programación de gráficos 3D. Su objetivo es facilitar el manejo y trazado de objetos gráficos elementales (líneas, polígonos y texturas) en cualquier

[10] Renderización es el término adaptado de la palabra en ingles “render” que indica tratamiento de gráficas, y se usa habitualmente en la industria de los videojuegos.

aplicación que despliegue gráficos en tercera dimensión, así como efectuar de forma transparente transformaciones geométricas sobre dichas entidades.

Ambas APIs están especialmente diseñadas para explotar al máximo las capacidades de las tarjetas de aceleración grafica y se encargan del manejo de bajo nivel, incluyendo la comunicación con los drivers, permitiendo que los desarrolladores de motores de renderización se enfoquen en efectos avanzados como la iluminación en tiempo real, los shaders, suavizado de contornos, etc.

1.2.3 Motor de audio.[11] Es el encargado de la parte sonora del videojuego, dandole un sentimiento de realismo mayor al permitir realimentación no solo gráfica sino auditiva. Existen distintas APIs que facilitan la conexión entre los dispositivos de hardware y la programación de alto nivel, las más usadas son OpenAL y DirectX Audio. OpenAL es un estandar multiplataforma utilizado en videojuegos y otras aplicaciones auditivas. Desarrollada por Loki Software en 1999, y sostenida actualmente por Creative Labs, OpenAL (Open Audio Library) es una librería de libre utilización que permite la creación de sonido envolvente, tridimensional y capaz de reproducir cualquier tipo de extensión de audio.

DirectX Audio es el API de Microsoft que sostiene tanto a DirectMusic como a DirectSound, siendo el primero una posibilidad de programación a mayor nivel que la segunda, permitiendo mayor facilidad a costa de menor control (no es una diferencia significativa). Sus contrastes con OpenAL no son radicales, manteniendo una equivalencia en todos los aspectos relevantes de reproducción de audio y sonidos. Su mayor ventaja es la facilidad de implementación con otros estándares de DirectX (en especial DirectInput y DirectGraphics).

[11] MCCUSKEY, Mason. Beginning Game Audio Programming. Premier Press. 2003. Part One: Audio Engine Basics. p. 122.

1.2.4 Motor de Entrada/Salida.[12] Existen pocos APIs de amplia utilización que faciliten la creación de motores de entrada/salida de libre utilización, siendo DirectInput uno de los estándares más usados profesionalmente, debido a su capacidad de adquirir datos de entrada desde los principales puertos del computador. DirectInput permite la recepción e interpretación de datos desde periféricos de entrada tales como el Mouse, el teclado y el Joystick, además de realimentación de fuerza, el cual es un tipo de realimentación que permite impulsos que son percibidos por dispositivos con la capacidad de reaccionar adecuadamente (un Joystick que vibre dependiendo de las ordenes recibidas por el programador). Tampoco se hace diferenciación radical de acuerdo al tipo de dispositivo del que se recibe la entrada, por lo que depende del programador hacer las respectivas modificaciones.

1.3 OTROS CONCEPTOS

Debido a la alta complejidad teórica del proyecto, se plantea un glosario especializado que explica con mayor detalle los conceptos importantes a tener en cuenta en este proyecto:

1.3.1 Conceptos físicos.

- **Aplicación del principio de D'Alemberts.** Este principio relaciona diferenciales de tiempo en las ecuaciones de movimiento de partículas, sin

[12] ZERBST, Stefan; DUVEL, Oliver. 3D Game Engine Programming. Thomson Course Technology, 2004. Game Developers Series. p. 456

embargo este principio prueba que si se tienen mas de una fuerza actuando sobre una partícula, puede calcularse una sola fuerza neta sobre la partícula por medio de realizar una suma vectorial de todas las fuerzas que actúan en un instante de tiempo sobre ella.

- **Ley de Hooke.** La ley de Hooke indica que la fuerza desprendida por un resorte, depende exclusivamente de la elongación de dicho resorte, una elongación consiste en una compresión o estiramiento indistintamente. La formula de Hooke relaciona esta fuerza con una elongación de la siguiente manera:

$$F = -k\,(l - lo)$$

Donde f es la fuerza ejercida por el resorte, l es el tamaño actual del resorte, lo es el tamaño natural del mismo y k es su constante de resorte, esta constante indica que tan elástico es el resorte y depende de su material y forma.

- **Motor de partículas.** Un motor de partículas visto desde el punto de vista físico, es aquel que se encarga de realizar los cálculos que afectan a partículas puntuales (en algunos casos tratadas como micro esferas para facilitar algunos cálculos), estos cálculos tienen por objetivo el resolver el sistema de todas las fuerzas que actúan sobre ellas, causadas ya sea por el "mundo" virtual o por choques con otras partículas, esto con el fin de determinar su velocidad en cada momento y con esto su posición a través del tiempo.

- **Motor de masas agregadas.** Un motor de masas agregadas se caracteriza por simular objetos rígidos con base en física de partículas (usando un motor de partículas), para esto se usan vínculos especiales entre partículas, como

lo son los cables, las varillas y los resortes, de modo que una caja podría ser representada por 8 partículas unidas por varillas. Estos vínculos también pueden usarse para relacionar una partícula a un punto fijo en el espacio, de modo que se pueden lograr efectos de objetos como un puente colgante o una cámara que sigue un personaje.

- **Motor de cuerpos rígidos.** Este tipo de motor simula de forma integral un objeto como una unidad y no una unión de otros más pequeños, estos motores son muchos mas complejos debido a que incluyen cálculos de rotaciones combinados con translaciones sobre los centros de gravedad del objeto para lograr que estos se muevan, sin embargo son motores mucho mas eficientes, estables y realistas que los motores de masas agregadas.

- **Motor basado en impulsos.** Los motores basados en impulsos, son aquellos en los que los cálculos matemáticos se hacen con base en la física de impulsos, estos motores tratan las fuerzas como muchos impulsos cantantes a través del tiempo, de modo que no hay que hacer predicciones matemáticas sino que se actualizan las fuerzas en cada iteración del motor, esto les resta precisión, pero mucho mas sencillos de implementar.

- **Motor basado en fuerzas.** Los motores basados en fuerzas son más precisos que los motores basados en impulsos, esto se debe a que tratan las fuerzas adecuadamente según la mecánica clásica newtoniana para realizar sus cálculos, además tratan los impulsos como fuerzas de muy poca duración.

- **Motor de división por tiempo.** En un motor de división por tiempo, los cálculos de las físicas se hacen en base al tiempo del computador donde se ejecuta la aplicación, más exactamente el tiempo de ejecución del videojuego que lleva el procesador, esto tiene implicaciones como que si el motor grafico funciona lentamente, el mundo de las físicas en el videojuego va a ir mucho mas rápido que el mundo que se ve en pantalla, obteniendo como resultado objetos moviéndose mucho mas rápidamente de lo supuesto o saltando en el espacio.

- **Motor de división por frames.** Un motor de división por frames, realiza los cálculos de las físicas del juego, dependiendo del tiempo de duración del frame en el juego, como este tiempo varia constantemente durante todo el juego, los cálculos hechos de esta manera son poco constantes y dependen fuertemente del rendimiento de videojuego en general, si el videojuego esta corriendo muy lentamente los intervalos en el motor de físicas van a ser muy largos y esto puede tener resultados algo diferentes que en una aplicación corriendo correctamente.

- **Motor de tratamiento de contactos iterativo.** Este tipo de motor trata los contactos entre los cuerpos uno por uno a la vez, sin importar si estos se generaron simultáneamente, se trata el más representativo durante cada iteración, este modelo es fácil de implementar y en casos excepcionales el tratamiento iterativo de contactos genera comportamientos extraños en los objetos.

- **Motor de tratamiento de contactos de forma Jacobiana.** En este enfoque los se calculan los efectos de los contactos en un determinado instante del tiempo sobre los objetos del mundo de forma simultanea, sus resultados son mucho mas exactos que en el tratamiento iterativo, sin embargo las ecuaciones involucradas son muy complejas y requieren de demasiado procesamiento para

poder ser resultas (sin mencionar que en muchos casos no se pueden resolver y hay que tratar esas excepciones); por consiguiente son pocos los motores de físicas para juegos que usan este enfoque, son mas comunes en motores de simulaciones realistas.

1.3.2 Conceptos de programación.

- **Interfaz.** En c++ una interfaz es simplemente una clase abstracta que contiene solo la declaración de miembros públicos y deja que estos sean implementados en otras clases que heredan de esta.

La principal función de las interfaces en desarrollo de software son: primero la de guiar y restringir a los programadores, ya que mientras codifican una clase que implemente un interfaz, tiene que limitarse a los miembros que la interfaz tiene definidos; y segundo, esta función es específica del motor, consiste en hacer el desarrollo independiente de algún API grafico.

- **Biblioteca estática.** Una biblioteca estática, es una biblioteca que puede contener código, funciones y variables, que son copiados en el programa que la use por el Linker de un compilador, produciendo un Object File y un ejecutable independiente.

- **Biblioteca de vínculos dinámicos (DLL).** Es la versión que Microsoft implemento del concepto de librería compartida en sus sistemas operativos, y permite que se cargue en tiempo de ejecución, y existe como un archivo independiente.

- **Apuntadores a funciones.** Es muy similar a un apuntador a una variable, guarda la dirección en memoria de la función y esta puede ser llamada por medio del apuntador. También existe otro tipo de apuntadores de funciones, que no hacen referencia a una dirección en memoria, sino a una dirección dentro de una biblioteca; este concepto es fundamental al momento de exportar funciones de una biblioteca dinámica.

1.3.3 Conceptos de programación gráfica.

- **Stencil Buffer.** Es un buffer extra que se incluye en los dispositivos modernos de gráficos, su funcionamiento se realiza por cada uno de los pixeles en la aplicación y maneja valores enteros. Su uso más simple consiste en delimitar áreas de renderizado. Puede trabajar además en conjunto con el depth buffer para crear efectos de profundidad de una forma más rápida.

- **Píxel Buffer.** Contiene la información de color e intensidad de cada uno de los píxeles por renderizar.
- **Depth Buffer.** También llamado Z-Buffer. Se encarga de gestionar las coordenadas de profundidad de las imágenes en los gráficos en tres dimensiones (3-D). Este buffer se distribuye en como un arreglo en dos dimensiones (x,y) con un elemento por cada píxel de la pantalla, y si se da el caso en el que otro píxel deba renderizarse en el mismo lugar se compara el índice de profundidad de cada uno y escoge la que se encuentre más cercana al punto de observación.

- **3D Pipeline.** Un 3D Pipleline es un software que se encarga de transformar para que el hardware pueda usarla, la información 3D a un espacio 2D y ponerle color a los píxeles de salida en pantalla.

Es un concepto fundamental para diferenciar un API gráfico de un Motor gráfico, ya que en el caso del primero y ejemplos como DirectX u OpenGL realizan las funciones de un Pipeline y le agregan efectos por medio de algoritmos a la información antes de mostrarla en pantalla. Por otro lado, un motor debe ofrecer una interfaz al usuario que le permita manipular y combinar las funcionalidades de los API gráficos, sin tener que preocuparse por la interacción con los mismos.

- **Renderización.** Su connotación se basa en el término en inglés: “render”, el cual representa el proceso de generar una imagen basados en modelos computarizados. La imagen se compone de pixeles discretos, que tienen un color, una opacidad y demás propiedades asociadas.

2. ESTADO DEL ARTE

2.1 HISTORIA DE LOS VIDEOJUEGOS

2.1.1 Orígenes.[13] La historia de los videojuegos inicia un poco después que la de los computadores, a finales de la década de los 40 e inicios de los 50, momento en el que se crearon varios juegos con ayuda de un computador. Sin embargo, estos no eran videojuegos en si, ya que no usaban video para su visualización, sino elementos mas simples como son los leds o similares. Poco después (1958), se creó el primer juego con video usando un osciloscopio, este juego fue la primera versión del afamado pong, cuyo autor (Hill Nighinbotthan) nunca decidió patentar y su éxito comercial quedó en manos de Nolan Bushnell a nombre de una empresa denominada Atari en 1972.

El Massachussets Institute of Technology (MIT) realizó en 1961 “spacewar”, un juego de naves cuya sola finalidad era dispararse entre si, y fue elaborado usando la computadora PDP-1, la cual representaba un inmenso avance para la época. Esta tenía una memoria de 9 Kilobytes ampliable a 144; y su costo era de $100.000 dólares; incluía una máquina de escribir conectada para el ingreso de datos, un monitor CRT como salida y un sistema de cintas perforadas para almacenaje. Incorporaba un sistema operativo multiusuario y multitarea, lo cual la mantenía entre la élite de los computadores de la época. Gracias a estos avances nacen las primeras empresas de desarrollo de videojuegos per se.

[13] GARCIA SERRANO, Alberto. Programación de videojuegos con SDL. P. 6 2005 En: http://www.agserrano.com/libros/sdl/[ebook]Programacion%20de%20videojuegos%20con%20SDL.pdf.

2.1.2 Empresas. Es en 1889 cuando Fusajiro Yamauchi decide fundar la empresa japonesa Nintendo, que en sus inicios se dedicó a la fabricación de naipes para juegos de mesa. Sin embargo, con la aparición de los videojuegos Nintendo decidió cambiar de mercado, naciendo de esa una de las empresas más grandes en la industria de los videojuegos. A principios de los años 70 lanzaron un dispositivo de juegos para salas recreativas, basados en principio en la reproducción de video, nacería la primera consola de videojuegos de Nintendo, en conjunto con Mitsubishi Electric, se desarrolla para Japón un sistema de videojuegos utilizando reproductor de vídeo electrónico (EVR). Decide también explorar distintas alternativas, y es así como en 1977 presenta su consola doméstica TV GAME 6 y la TV GAME 15. En los años ochenta la Super Nintendo Eentertainment System (SNES) y continúan su evolución hasta 1996 con el Nintendo 64, la primera consola que funcionaba a 64 bits.[14]

Otra compañía importante en el mundo de los videojuegos nacería en los años 40, en el momento en que una empresa llamada Service Games que se dedicaba a la comercialización de maquinas de monedas, se fusionase con la empresa Rosen Enterprises para crear SEGA. En 1968 lanzarían su primer producto, un simulador de submarinos, momento desde el cual SEGA participó en la guerra de las videoconsolas, con lanzamientos como la SG1000 en 1983, la Mark III, en 1986 la Sega Master System y en 1988 la Sega Megadrive a 16 bits por primera vez en la historia de las consolas, y con un éxito bastante grande que no volvería a repetir. Su última consola fue la Dreamcast que no resultó exitosa y sacó a la empresa de la competencia de videoconsolas[15].

---

[14] NINTENDO. Historia de Nintendo Worldwide de 1889 a 1979. [online] Consultado: septiembre 20 de 2007. En: http://www.nintendo-europe.com/NOE/es/ES/corporate/index.jsp

[15] SEGA OF AMERICA. Sega History. [online] Consultado: 20 de septiembre de 2007. En: http://www.sega.com/corporate/corporatehist.php?item=corporate_history

2.1.3 Consolas.[16] En 1972 Ralph Baer creó la primera consola de videojuegos, la cual llevaba los videojuegos al entorno doméstico. Incluía diez juegos distintos (tenis, hockey, submarino, simon, ski, etc.), vendiendo mas de 100.000 unidades y creando una revolución más en el mundo de los videojuegos. ATARI también lanzaría su propia consola en 1975, en la que solo se podía jugar "pong". Dos años después, ATARI volvería a aparecer, esta vez con una mejor propuesta: el atari2600, la cual contaba con un procesador de 8 bits. Para 1982 ya había 8 millones de unidades en los hogares. En 1976 dos jóvenes estudiantes Steve Jobs y Steve Wosniak diseñaron para Atari lo que sería la primera versión de arkanoid, un juego altamente entretenido, aunque en ese momento la compañía desecho el proyecto, por lo que estos desarrolladores enfocaron sus esfuerzos en otras actividades.

En 1983 Nintendo presentó su consola: Nintendo Entertainment System (NES), que incluía la primera versión comercial de Super Mario Brothers. La consola costaba 200 dólares y vendió 3 millones de unidades antes de cumplir dos años en el mercado. Se estima que durante toda su vida comercial la consola vendió 65 millones de dispositivos y 500 millones de cartuchos.

En 1995 el Sega Saturn fue develado. Utilizaba dos procesadores de 32 bits y reproducía juegos en CD. Sin embargo, su alto precio (400 dólares) y la falta de desarrolladores interesados en el proyecto haría poco rentable la propuesta. Por otro lado, Sony presentaría su primer sistema de videojuegos: el PlayStation, que aparecería ante el mercado con un precio de 300 dólares y se convertiría en el dispositivo más popular del momento. La primera versión de PlayStation llegó a vender 102 millones de unidades[17].

[16] WIKIPEDIA. Video Game Console. [online] Consultado: 21 de septiembre de 2007. En: http://en.wikipedia.org/wiki/Video_game_console

En 2000 Sony presentó el PlayStation 2, con un procesador de 128 bits y una memoria de 32 MB. En diciembre de 2004, el PlayStation 2 alcanzó la cifra de 1000 millones de unidades vendidas[18].

Otra industria tecnológica entraría en la competencia comercial de los videojuegos en 1999; Microsoft incursionó en el mercado de las consolas de videojuegos con el lanzamiento del Xbox, el cual incorporaba un procesador de 733 MHz, una memoria RAM de 64 MB, un disco duro de 10 GB y una tarjeta de red para conectar hasta cuatro consolas, lo que permitía partidas de hasta 16 jugadores. En noviembre, Nintendo lanzaría también su aporte a la guerra de consolas, el GameCube, el cual logró vender 500.000 unidades durante su primera semana en el mercado. Casi al mismo tiempo, la misma compañía lanzaría otra consola portátil, el GameBoy Advance, la cual alcanzaría hasta 40 millones de consolas[19].

Después de esto empezaría una nueva era en la guerra de videoconsolas, cuando Microsoft lanza la Xbox 360 en 2005, que trabaja con un procesador de tres núcleos (cada uno con 3,2 GHz de velocidad) incluyendo 512 MB de memoria RAM y una tarjeta de aceleración gráfica de 500 MHz. Se han vendido más de 2 millones de unidades[20].

Nintendo y Sony también participarían en la actual guerra de consolas. A finales del 2006 Nintendo lanzaría al mercado su revolucionaria consola Wii, que cuenta con una innovación tecnológica que consta de un control inalámbrico con detección en tres dimensiones. Por último, también a finales del 2006, Sony

---

[17] SONY COMPUTER ENTERTAINMENT INC. Cumulative Production Shipment of Hardware / Playstation. [online] Consultado: 21 de septiembre de 2007. En: 2007. http://www.playstation.com/products.html.

[18] Ibíd.

[19] NINTENDO. Consolidated Financial Statements. [online] Consultado: septiembre 20 de 2007. En: http://www.nintendo-europe.com/NOE/es/ES/corporate/index.jsp

[20] MICROSOFT CORPOTATION. Quarterly Report. 2007. [online] Consultado: 22 de septiembre de 2007. http://www.sec.gov/Archives/edgar/data/789019/000119312507091808/d10q.htm

lanzaría el tan esperado PlayStation 3, que cuenta con un desarrollado por IBM, Toshiba e Intel, con un potencial mayor que el de sus competidores[21].

2.2 MOTORES PARA VIDEOJUEGOS

En la historia de los videojuegos, especialmente de los "shooters[22]" (juegos donde lo único que se ve es el arma y se tiene una vista en primera persona), las secuelas se hicieron muy relevantes comercialmente hablando, y por consiguiente aun más el código para crearlos, por ello se planteó la idea de usar los fundamentos conceptuales de los antiguos juegos para crear sus nuevas versiones, es así como se extrajo el "núcleo" del juego, y se le denominó el "motor" del juego, el cual podía ser reutilizado para secuelas o juegos similares.

De esta forma se comienza con una metodología de diseño basado en motores que reutilizaba el código de juegos de alto presupuesto a través de la comercialización de los mismos. Poco a poco los motores fueron especializándose mas, enfocándose en distintos aspectos de lo que podría contener un juego dependiendo del énfasis que se le deseara dar. Es así como nacen los motores de física, los motores gráficos (separados de los demás con tecnologías gráficas más poderosas), los motores de inteligencia artificial, entre otras tecnologías que hacen de este concepto una extrapolación de la reutilización de APIs o de librerías pero en un marco mucho más robusto.

---

[21] KUSHNER, David. The Insomniacs. En: Spectrum: The magazine of technology insiders. IEEE. Volumen 43. No 12 (septiembre de 2006) p. 18-23.

[22] La utilización de la palabra "shooter" se hace basada en el hecho de no encontrar una palabra afín en español que no se preste para ambigüedades.

El concepto de motor es exclusivo de los videojuegos, y ha permitido una gran diferenciación con otro tipo de desarrollo de aplicaciones, a pesar de mantener los mismos fundamentos y base para su elaboración. Estos motores han facilitado la elaboración de juegos tanto a empresas especializadas como a estudiantes iniciándose en este mundo artístico y a la vez técnico.

Algunos de los motores más usados durante el aprendizaje del desarrollo de videojuegos son: el Delta 3D, el Genesis 3D, el Ogre 3D, el Irrlicht Engine, el Neo Engine, el Torque Game Engine, el TrueVision3D y el C4 Game Engine.

2.3 ESTADO DEL ARTE A NIVEL REGIONAL

En el entorno regional (Latinoamérica) el desarrollo de videojuegos es una práctica muy reciente, tanto en la academia como en la industria, además existen pocos proyectos y grupos dedicados a ello, por lo que Colombia tiene una participación casi nula en el ámbito global en comparación con países europeos, Estados Unidos o Japón.

En el campo académico es de remarcar la iniciativa que se desarrolla en la UNAB, donde se formó el semillero de investigación en desarrollo de videojuegos, donde actualmente se trabaja con C# para la programación y Photoshop para la parte grafica.

En la Institución Universitaria de Envigado (IUE) existe un semillero de videojuegos que ya participo en el encuentro de semilleros Redcolsi, Nodo Antioquia, con dos proyectos desarrollados. El primero, un semillero infantil de Videojuegos para niños con capacidades excepcionales (Municipio de Itagüi), y el

segundo: Diseño de videojuegos como estrategia de promoción del programa Ingeniería de Sistemas de la IUE.

En la facultad de ingeniería de sistemas y computación de la Universidad Tecnológica de Pereira se creo el grupo GDA (grupo de desarrollo Allegro), que debe su nombre a su desarrollo en proyectos con la librería grafica Allegro.

En cuanto a empresas y desarrollo profesional encontramos a 3D Logical, la cual es una empresa en surgimiento de la ciudad de Medellín que ofrece servicios de gráficos y animaciones en 3 dimensiones, la cual ya realizó un videojuego para las EPM (Empresas Públicas de Medellín), el cual se desarrolla en la misma ciudad, aunque su calidad es altamente debatible.

Colombia Games es una empresa que se dedica al desarrollo de videojuegos para dispositivos móviles, que recibió el apoyo del Fondo Emprender y en este momento cuenta con varios títulos en el mercado de móviles.

El único ejemplo de empresa desarrolladora de juegos de consola exitoso en Colombia (y Latinoamérica en general) es Inmersión Games, que ya tiene en el mercado su primer juego AAA para Xbox 360 llamado “Monster madness”, también están desarrollando un esperado título llamado “Cell Factor”, los cuales han generado alta expectativa en la industria.

Todos estos grupos y empresas están trabajando en el desarrollo de videojuegos usando motores ya existentes, pues necesariamente deben usar esta tecnología para desarrollar sus productos ya que su construcción puede llevar años de trabajo y tener costos muy elevados. En cuanto a los semilleros, todos están enfocados al diseño de videojuegos y no propiamente al de un motor o a su metodología y no propiamente a un motor, debido a la gran complejidad de los

mismos, y cabe resaltar que todos los motores que usan han sido desarrollados fuera de Latinoamérica.

3. TECNOLOGÍAS USADAS[23]

3.1 LENGUAJE DE PROGRAMACIÓN C++

Se ha escogido el lenguaje de programación C++ debido a su efectividad y versatilidad, su velocidad de ejecución y su capacidad de alcanzar el nivel de programación mas cercano a la máquina aprovechando a la vez, las ventajas de un lenguaje de alto nivel y de estándares tan útiles como la programación orientada a objetos.[24]

3.2 HERRAMIENTA DE DISEÑO GRÁFICO MILKSHAPE 3D

El software de diseño gráfico Milkshape 3D se ha escogido debido a su bajo precio y a su versatilidad en el manejo de formatos para los modelos. Cuenta además con una gran capacidad de adaptación con diversos plug-ins que lo hacen una herramienta capaz y adecuada para desarrollos moderadamente complejos.

---

[23] WALSH, Peter. Advanced 3D Game Programming with DirectX. WordWare Publishing, 2003. p. 13

[24] STROUSTRUP, Bjarne. 2007. [online] Consultado: 22 de septiembre de 2007. En: http://www.research.att.com/~bs/homepage.html

3.3 DIRECTX

Es importante conocer las distintas API (Interfaz de Programación de Aplicaciones) Graficas que existen para el desarrollo de aplicaciones que hagan uso de las tarjetas graficas de manera adecuada, y que ademas simplifiquen la tarea de la interacción con el hardware indistintamente de la maquina en la que se corra. Los dos mas importantes son el OpenGL y el DirectX (específicamente el Direct3D para la parte grafica). OpenGL es una librería práctica, rápida de utilizar y es libre, por lo que parece una muy buena opción, aunque tiene grandes limitantes, con los que DirectX sabe lidiar como la versatilidad al saber aprovechar los distintos tipos de hardware de la forma mas eficiente posible, sin depender de que el fabricante del hardware haya desarrollado especificaciones para que se pueda usar el estándar.

4. DISEÑO Y DESARROLLO DEL MOTOR

4.1 METODOLOGÍA

La metodología de creación del motor se fundamentó en el ciclo de vida en espiral, realizando reiteradamente un ciclo de aprendizaje, implementación, pruebas e implantación en el proyecto macro. De manera general se investigó teoría adecuada sobre los fundamentos principales de cada motor, luego se elaboró la estructura del motor en general, priorizando la modularidad y facilidad de utilización. Esta estructura sería implementada posteriormente en las subdivisiones de acuerdo a los módulos que se crearon, y la realización de una demostración de cada uno estos se hizo indispensable. En una iteración posterior, se realizó la integración de todos los módulos, asegurando la facilidad de utilización tanto de los módulos por separado como su efectividad para crear aplicaciones multimedia utilizando todos los módulos en conjunto.

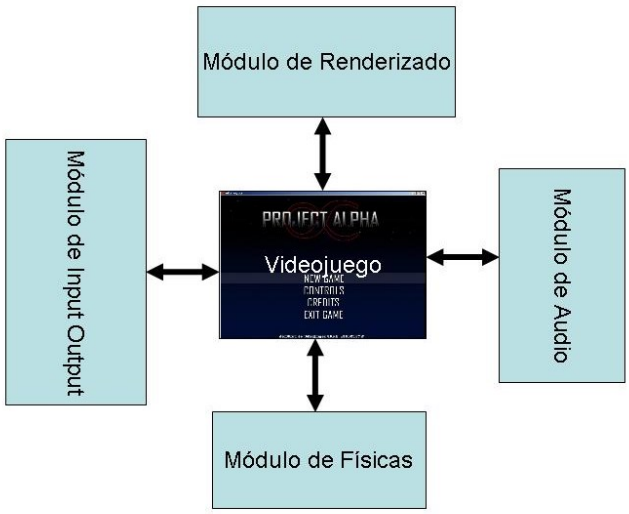
4.2 DESARROLLO DE LA ESTRUCTURA DEL MOTOR

El motor se desarrollo enfocándose en un concepto genérico modular, permitiendo la creación de videojuegos de distinta índole, priorizando los juegos de disparos en primera persona, juegos de aventura en tercera y primera persona y los juegos de deporte. El código del motor se encuentra en inglés, permitiendo abarcar una mayor población de posibles usuarios alrededor del mundo, y aprovechando el hecho de que la programación de alto nivel es un lenguaje basado en el inglés, por lo que un programador de videojuegos deberá tener la capacidad de entender las

funciones utilizadas en el mismo (los nombres de las funciones de la mayoría de las librerías de C++ están basadas en el inglés).

El desarrollo del motor se ha planteado dividiéndolo en módulos independientes entre sí, pero permitiendo una integración suficientemente sencilla para la creación de videojuegos. El modelo utilizado se puede visualizar en la figura 1.

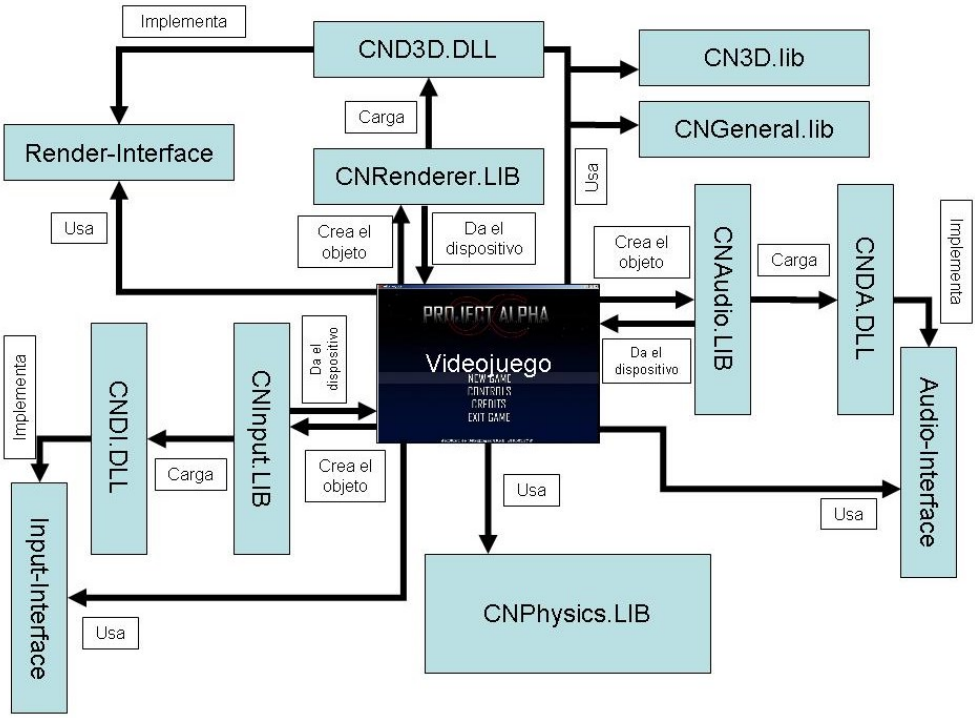
Figura 1. Modelo conceptual del motor desarrollado por módulos.



Fuente: Autores.

Existe también una estructura interna dentro de los módulos compuesta por una librería estática, una dinámica y una interfaz (con la excepción del módulo de físicas). Adicional a los cuatro módulos existen dos librerías estáticas que implementan generalidades del motor que no necesitan un módulo completo para su utilización. Un nivel de detalle mayor puede apreciarse en la figura 2.

Figura 2. Estructura interna del CNEngine.



Fuente: Autores.

El CNEngine esta compuesto por distintos submotores que pueden definirse como “módulos” del motor completo, y debido a que es un motor de juegos debe poder realizar todas las tareas que implican la creación de un videojuego. Cada módulo está conformado por la interfaz que declara las clases, variables y funciones que el videojuego podrá utilizar, pero puesto que solo se encuentra la declaración, su implementación queda supeditada a otro archivo el cual es el: “nombredelmodulo”.DLL, el cual es una librería de enlace dinámico (Dinamic Link Library), donde se encuentran encapsuladas todas las definiciones pertinentes. Adicionalmente existe una librería de enlace estático cuya función es la de cargar el DLL; esto lo logra mediante el paso de la dirección en memoria de un objeto creado de la implementación de la clase que reside dentro del DLL, que usa este

objeto mediante la definición en la interfaz (en el caso del módulo de renderizado este objeto es el “Device”).

A diferencia de los otros tres módulos, el módulo de físicas se compone por una librería estática debido a su dependencia de los otros módulos en su utilización. Esto genera que se deba incluir el archivo Physics.LIB en cada proyecto donde se deben agregar físicas, al igual que con los demás módulos, pero a diferencia de estos, la librería no cuenta con una librería dinámica que la acompaña, por lo que toda la implementación de las físicas se encuentra embebida dentro del ejecutable que se genere.

Adicionalmente a los cuatro módulos principales se encuentran dos librerías estáticas, el CN3D.lib y el CNGeneral.lib, las cuales se encargan de la parte matemática para cálculo de objetos en tercera dimensión del motor y el movimiento de la cámara, su objetivo y control del movimiento.

El motor es completamente independiente de los proyectos que se hagan con el mismo, es decir, puede implementarse un videojuego sin modificar el código del motor y el programador del videojuego solo debe preocuparse por utilizar las funciones que el CNEngine pone a su disposición. Entre otras ventajas, una razón para que el motor esté dividido en módulos es que un programador puede decidir usar determinados módulos y no utilizar otros, por lo que se brindan las facilidades para realizar distintos proyectos con enfoques completamente diferentes. Aunque en el caso del módulo de físicas, se genera una dependencia intrínseca debido a la imposibilidad de dar una realimentación gráfica.

A continuación se hará una explicación más detallada de cada uno de los módulos:

4.2.1 Módulo de Render. Es el módulo que se encarga de la renderización de los objetos en pantalla. Tiene la más alta complejidad en términos de desarrollo. Como opción de comunicación con el hardware se ha escogido (la justificación se realizó anteriormente) el Direct3D para implementar todas las opciones que puede tener la renderización de los objetos dentro del videojuego. Debido al soporte de integración que presta Microsoft entre los distintos APIs que conforman el DirectX, el Direct3D es una posibilidad viable Microsoft como parte de su paquete DirectX, pero una reimplementación con base en el modelo propuesto para otra API (tal como OpenGL) es factible.

4.2.2 Módulo de Input. Es el módulo que se utiliza para comunicar la aplicación que se vaya a crear con el teclado, el Mouse y el Joystick, de manera que sea interactiva. Una aplicación puede desarrollarse sin este módulo, lo cual la convertiría en una aplicación gráfica, así como tures virtuales guiados, o una muestra de modelos en tercera dimensión con movimientos de cámara programados con antelación.

Los dispositivos de input pueden ser: el teclado, el Mouse, o de un joystick o una combinación de los anteriores. La conexión también hará uso del DirectX en la forma del DirectInput, que facilitará al motor la conexión con los dispositivos independiente del desarrollador del hardware.

4.2.3 Módulo de Audio. Este módulo le permite a la aplicación que se planea ser una experiencia multimedia, al permitir que la persona no solo obtenga realimentación visual de sus acciones sino también sonoras. No solo se encargará de crear la música que acompañará el juego sino también los efectos de sonido. Para lograr esto el motor interactuará con el DirectAudio y con el DirectSound.

4.2.4 Módulo de Físicas Otro de los módulos de alta complejidad es el motor de físicas, el cual está implementado de manera que se puedan realizar las colisiones adecuadas en los objetos que interactúan y se pueda crear una experiencia realista para el usuario. Se crean funciones y modelos de distintos comportamientos de objetos y sus reacciones, de manera que puedan añadirse a los modelos que se incluyen en la aplicación, de esta forma se crean mundos virtuales con reglas que dependen del programador del videojuego.

Por otra parte, se han desarrollado interfaces que permiten utilizar de manera adecuada la renderización de objetos y modelos en tercera dimensión y la aplicación de texturas sobre ellos. Las interfaces tienen un segundo propósito que es el de encapsular la implantación de las clases que utiliza el motor para realizar los procedimientos pertinentes, por lo que el concepto de programación orientada a objetos se utiliza en toda su rigurosidad, además de que previene el inconveniente de que la persona modifique el motor durante la creación de su juego y convierta el código en dependiente de ese proyecto en particular (si desea modificar el código fuente debe recompilar el motor previamente a su utilización en un proyecto).

4.3 MOTOR DE RENDERIZADO

4.3.1 Diseño. Es el de mayor complejidad, por medio de este se renderizan todos los modelos, texturas, iluminaciones, efectos, etc. Abarca campos tan amplios como el 3D Pipeline, los shaders (píxel y vertex), manejadores (de materiales, de modelos, de Skins) y los distintos buffers (píxel, depth y stencil). Para la creación de este motor, se utilizó el API Direct3D que pertenece a DirectX, de manera que la interacción con los distintos dispositivos de hardware se hiciera de manera transparente.

En primera instancia se determinó que el programador del videojuego debe tener la opción de escoger el tipo de enfoque que desea tener en su aplicación. Por tanto, se presenta una lista de posibilidades de los diversos hardware de acuerdo al equipo que se esté utilizando, lo que le daría total control respecto a las especificaciones sobre las que quiere desarrollar su aplicación.

Para permitir esto, se debió crear una enumeración de todos los dispositivos de hardware encontrados en el equipo, que se realiza en el archivo CND3DEnum en el proyecto CNRenderer (del que se hablará a continuación).

Existen dos proyectos dentro del motor de renderización, el primero es el CNRenderer, que conforma la librería estática que permite al programador hacer uso del objeto de tipo RenderDevice, el cual utiliza todas las funciones necesarias para la renderización. Esta librería es una interfaz que permite ver la declaración de las funciones que puede utilizar el programador, pero no su implementación. El otro proyecto es el CND3D, que contiene la implementación de las funciones que se utilizan en el CNRenderer y otras que no pueden ser accedidas por el usuario y están embebidas en el la librería dinámica, puesto que el objetivo de este proyecto es encapsular toda la información del Direct3D de manera que el programador no deba preocuparse por como se ejecuta el renderizado sino simplemente buscar su mejor forma de aplicación.

El motor de renderizado permite también tener distintas ventanas hijas, las cuales pueden renderizar cualquier parte del mundo virtual dependiendo del Viewport que se les asigne, y se utilizarán en la medida en que las swap-chains estén asignadas a ese Viewport. Pueden asignarse varias swap-chains a distintas ventanas, de manera que puedan verse distintas vistas de un mismo objeto. Esto permite que aplicaciones como Generadores de Niveles sean factibles de desarrollar con el motor puesto que las distintas vistas permitirían al desarrollador tener un control general de la escena.

4.3.2 Desarrollo. En el desarrollo del motor de renderización se implementaron las funciones y estructuras necesarias para su funcionamiento básico, pero debido a las altas necesidades de optimización y opciones de mejoramiento y efectos gráficos, se desarrollaron una diversidad de tecnologías que le permiten al motor una ejecución más fluida, y capacidades de renderización predeterminadas y abiertas de manera que el programador pueda hacer su propia implementación. A continuación se hará referencia a las tecnologías desarrolladas más importantes.

- **Manejadores.** Debido a los grandes requerimientos del procesamiento de imágenes, su adecuado manejo puede evitar que una aplicación responda de manera lenta o ineficiente. Un método adecuado puede lograrse administrando lo que se va a procesar. Esto se logra creando listas que carguen todas las texturas necesarias, y cuando algún objeto la necesite se llama de la lista, así no se debe volver a almacenar en memoria, simplemente se instancia al objeto adecuado.

El mismo concepto se aplica para las Skins, las cuales almacenan tanto las texturas como los materiales en forma de índices. Dependiendo del enfoque del programador, se pueden almacenar listas dependiendo de lo que el considere, optimizaría más el código, en el caso del CNEngine se almacenan listas de acuerdo a los objetos que tienen la misma textura. Estas listas se conocen como “manejadores”, puesto que manejan el uso de memoria adecuadamente.

Un buen ejemplo de la optimización que permiten los manejadores lo tiene el manejador de vertices (vertex manager), que le permite a Direct3D renderizar triángulos de forma rápida. Lo hace retardando la renderización hasta que sea absolutamente necesario (que hayan muchos datos almacenados o sucedió un cambio drástico) antes de cambiar de modo realiza la renderización.

- **Luces.** Otro factor importante del motor de renderizado es la posibilidad de agregar luces a la escena. Existen distintos tipos de luces que pueden añadirse:

  - **Luz direccional:** es una luz que proviene de un punto distante.

  - **Luz de punto:** es un punto el cual emite luz en todas las direcciones con la misma intensidad.

  - **Luz localizada:** es una luz que (como la de una lámpara) se enfoca en un punto específico, con un origen y una dirección determinados. Contiene dos conos, el primario enfoca una luz intensa, mientras que el secundario crea un efecto de desvanecimiento.

  - **Luz emisiva:** es una luz que puede emitir un objeto dentro del mundo virtual, pero esta luz no afecta los otros modelos en el mundo.

El motor de Render también permite la utilización de un color alfa que define la opacidad de un objeto manejando rangos entre 0 y 1; mientras que con un valor de 1 el modelo se hace completamente visible, un valor de 0 lo hace invisible en el proceso de renderización.

- **Visualización.** La visualización en la escena determina que es lo que está viendo el usuario, pues pueden existir una gran cantidad de objetos en el mundo virtual, pero lo más probable es que no se estén visualizando todos debido al la posición y dirección de la cámara. Además, cómo se ve la escena también depende de distintos factores, como el tipo de proyección.

Existen dos tipos de proyecciones, proyección ortogonal y proyección de perspectiva. La última se usa para factorizar la distancia desde la visión de la

cámara hasta el objeto y determinar las coordenadas de las otras dos dimensiones. Lo que hace es factorizar los valores de visión del objeto dependiendo de su profundidad para hacerlo ver en el plano de proyección más pequeño de lo que realmente es, así como por medio de la vista vemos los objetos lejanos más pequeños.

La perspectiva ortogonal es la que se usa en menús y herramientas en dos dimensiones en el espacio 3D. La idea de la proyección es que se pase de una imagen en 3D a una imagen en 2D, de manera que sea más fácil renderizar y no se necesite tener toda la información de profundidad.

Para poder visualizar los objetos, también es de importancia la ubicación del observador es la visión, lo cual se determina a través de la matriz de visión, en la que se determinan su posición y orientación. Aparte de la matriz, existe un rectángulo que permite observar la escena, el Viewport.

También existen los planos de clipping, que determinan el punto máximo y mínimo de distancia que debería ser capaz de ver la persona.

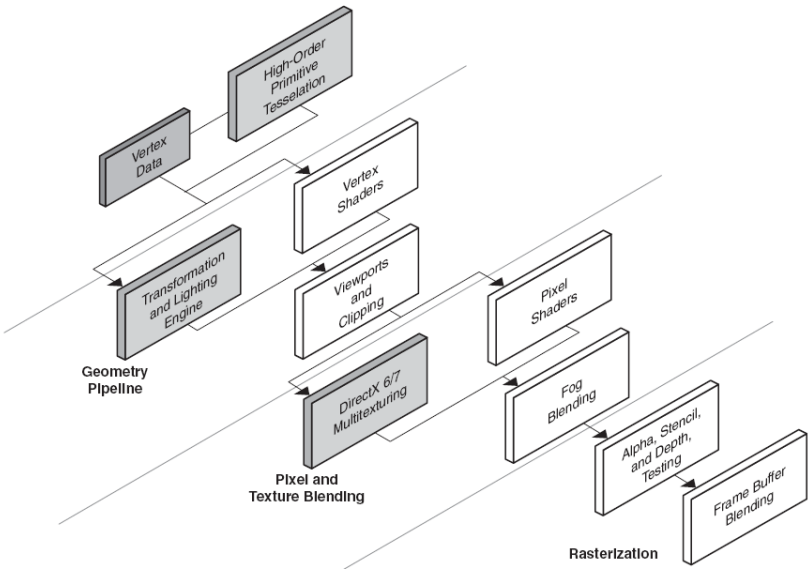
Es importante para poder proyectar los objetos que se pase de la matriz del mundo a la matriz de visión y por último a la matriz de proyección, de manera que se pueda pasar un objeto del mundo a la proyección para que sea visible, esto se hace multiplicando las distintas matrices formando un combo de matrices.

- **Pipeline.** El Pipeline es la posibilidad de transformar de las coordenadas del mundo (world coordanates) a las coordenadas de visión (view coordinates), de 3D a 2D, puesto que los objetos en pantalla solo se ven en dos dimensiones, con una opacidad diferente para manejar la profundidad. Este es el Pipeline de

función definida, el cual utiliza el procesado de imágenes a través de el TnL (Transformation and Lightning Engine, motor de luz y transformaciones).
El camino del 3D pipeline se establece se la siguiente forma:

Se envían los datos a procesar (vértices y datos), entonces se busca si está habilitado el vertex shader. Si no está habilitado se utiliza el TnL, y de esta forma se cambie el 3D pipeline de función definida (point fixed pipeline). Luego se hacen las transformaciones del viewport (se transforman las coordenadas del objeto desde la matriz de coordenadas del mundo hasta las del viewport) y luego se realiza el clipping momento en el cual ambos pipelines son iguales. Luego, si no hay shaders, se hace multitexturizado, y si sí se cuenta con la opción de shaders, se utiliza el pixel shader. Luego se aplica color de niebla y por último se aplican el alfa y profundidad, para poder ser enviados al mezclado del frame (frame buffer blending), que consiste en la renderización de todos los objetos (ver figura 3).

Figura 3. Camino de los dos pipelines.



Fuente: ZERBST, Stefan; DUVEL, Oliver. 3D Game Engine Programming. Thomson Course Technology, 2004. Game Developers Series, p. 337.

El Pipeline de función definida permite al programador modificar algunas características a través del envío de distintos parámetros, y no permite crear nuevas funciones o tener acceso al hardware gráfico. Contrario al Pipeline flexible, que permite aplicar cualquier función a los datos a través de una nueva que se exprese en una fórmula matemática establecida por los shaders.

- **Shaders.** Los shaders son programas que se ejecutan directamente en la GPU utilizando un lenguaje de programación de bajo nivel. Permiten crear efectos variados en la escena (agregar efectos a las texturas, utilizar múltiples texturas en un mismo modelo, índices de reflexión de la luz, etc.).

Existen dos tipos de shaders, los vertex shaders y los píxel shaders. Los primeros se utilizan para la renderización y efectos enfocados a los vértices de los modelos, mientras que los segundos son programas que manipulan los colores de un pixel después de que se renderiza una primitiva.

Así como los skins, los shaders no pueden ser accedidos directamente, solo a través de su identificador (ID). Solo puede estar activo un vertex shader y solo un pixel shader.

Los vertex y pixel shaders deben ir el uno al lado del otro, puesto que hay una necesidad intrínseca de renderizar con ambas tecnologías al mismo tiempo. El siguiente es un ejemplo de un par de shaders que simplemente agregan una textura a un objeto.

Ejemplo de vertex shader:

```
vs.1.1                  -> versión del vertex shader
dcl_position v0         ->la posición del vertice
```

dcl_normal v3	-> el vector normal del vertice
dcl_texcoord v6	-> un arreglo de coordenadas de una textura
dcl_tangent v8	-> vector tangente
m4x4 oPos, v0, c0	-> la matriz de combinación almacenada en C0, y el m4x4 multiplica una matriz por un vector en 4D y el oPos contiene la posición tranformada del vertice
mov oD0, c4	->la luz ambiental se almacen en c4 entonces se pasa al registro
mov oT0, v6	

Ejemplo de pixel shader:

ps.1.1	-> versión del píxel shader
tex t0	-> pasa al registro
mul r0, t0, v0	-> multiplica el color por el color difuso y logra el color final

De esta forma se entiende que los shaders son una alternativa al método de renderizado por defecto, dando mayores posibilidades al programador de experimentar.

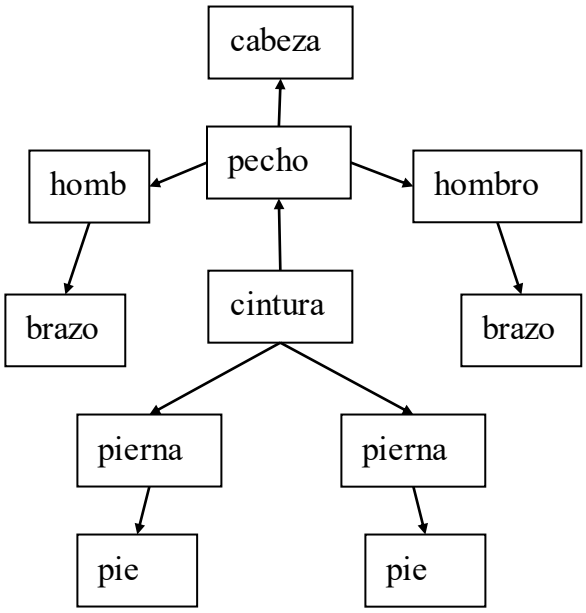
- **Renderizado.** Renderizar triángulos en varias pasadas de código es una utilización muy pobre del rendimiento, así que un cache guarda los triángulos en una lista para la función de Render solo almacena los objetos dentro de una lista, y solo cuando se llama la función Flush() Realmente se renderizan los objetos. Cuando el cache se llena se renderizan objetos dentro de la lista para sacarlos de la lista y guardar los nuevos.

Debido a que existen varios formatos de vertices por lo que el manager del cache tiene varios arreglos para manejar cada uno el forceflush() renderiza un objeto,

sus triangulos, materiales y texturas, y el forceflushall() lo renderiza todo para que todos los indices queden vacios, la idea del algoritmo del manager es que almacene todo en pots (vasijas) donde se almacenen todos los modelos que usen la misma skin, de manera que se usen los mismos pots, y cuando se llenen todos se vacía el pot más lleno.

- **Animación.** La animación actualmente se realiza utilizando el concepto de Bones, una técnica que mantiene unido al modelo a través de uniones (Joints) de manera jerarquica, estando el movimiento de algunos joints supeditados al movimiento de otros. De esta manera existen los joints hijos que se supeditan a los joints padres. El movimiento que se realiza es rotacional, no pueden trasladarse (tal y como sucede en el cuerpo humano). También existe un Joint Raiz, puesto que no tiene padre. Cuando este Joint se mueve, todos los demás se mueven.

Figura 4. Unión jerárquica de Joints.



Fuente: autores.

Dentro del motor todos los datos se almacenan en arreglos los cuales apuntan a los índices del modelo, sus vértices y sus skins.

Para realizar la rotación la información de cada vértice se almacena en una matriz que guarda las rotaciones relativas del Joint (que dependen del padre), otra que guarda la posición absoluta, y una matriz final que determina como se debe rotar el modelo. Para obtener esta matriz se debe multiplicar la matriz de posición relativa por la matriz que contiene la posición absoluta del padre, matriz a la cual se le transforma en su transpuesta, para finalmente permitir su utilización.

¿Pero a que posición debe rotar? Constantemente se está actualizando si hay algún movimiento en los vértices objetivo, si los hay se realiza una interpolación entre la matriz posición en la que se encuentra y la matriz posición a la que debe llegar.

Por último se actualizan los vértices del modelo, los cuales deben estar añadidos a un Joint, y generar la rotación pertinente.

4.4 DESARROLLO DE LA LIBRERÍA MATEMÁTICA

A pesar de los avances en velocidad y rendimiento de las máquinas actuales, las aplicaciones gráficas siempre mantendrán necesidades difíciles de satisfacer, puesto que mientras nuevas tecnologías se desarrollan, mejores y más realistas técnicas son implementadas, requiriendo aun mayor capacidades de mas máquinas, ya sea memoria, procesamiento o velocidad de transmisión de los datos.

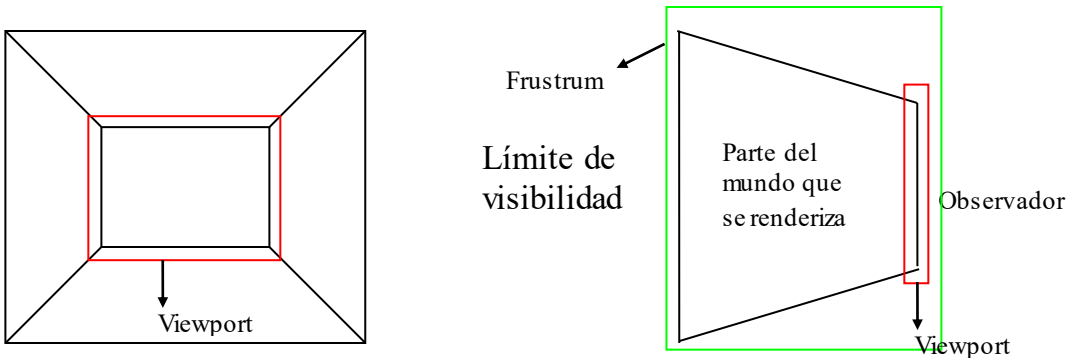
Debido a esto, la necesidad de optimizar el código a ejecutarse es indispensable para un funcionamiento eficiente del motor. Técnicas como algoritmos de eficiencia y la utilización de código Assembler son utilizadas en una librería estática llamada cn3d.LIB; nombre que se atribuye a la necesidad de realizar cálculos para objetos en tercera dimensión.

La optimización del código se logra al distribuir la carga de procesamiento entre el GPU (Graphics Processing Unit) y el CPU (Central Processing Unit), ya que mientras el primero realiza el procesamiento de los vértices e índices de los modelos, el segundo realiza los cálculos matemáticos pertinentes para organizar la información que procesara el GPU, así como todos los caculos que este ultimo no puede realizar. La librería estática se enfoca, por tanto, en la utilización de la CPU.

La utilización del lenguaje Assembler se hace a través del estándar SIMD (Single Instruction Multiple Data), el cual es un estándar propuesto por Intel y compatible con AMD y otros procesadores. La tecnología SIMD debe su nombre a que con ella se puede aplicar una sola instrucción a múltiple registros en el procesador paralelamente. Existen tecnologías superiores pero su utilización limitaría el número de máquinas que podrían utilizar el motor; además de que el SIMD contiene la capacidad necesaria para la optimización deseada.

Otra de las diversas técnicas que se utilizan en los videojuegos para optimizar el código es la evasión de cualquier exceso de procesamiento por renderizado. Es entonces cuando surge la idea del Frustrum, un poliedro similar al interior de una pirámide cortada horizontalmente por dos planos. Esta se determina como el área de renderizado, puesto que el Viewport solo permite una visualización en esta forma (la del Frustrum) en un momento determinado (ver figura 5).

Figura 5. Vista del viewport en un determinado momento.



Fuente: Autores

De esta forma nace también el concepto de Culling, el cual consiste en la detección de segmentos de modelos que se encuentren fuera del Frustrum y de esa forma no están dentro del rango de renderización (fuera del rango de vista del jugador), a pesar de que puedan encontrarse dentro del "mundo". También puede darse el caso en el que el modelo se encuentre parcialmente dentro del Frustrum, lo cual se conoce como Clipping, por lo que (también en miras al rendimiento) se eliminan las partes del modelo que no se encuentran dentro del rango (solo durante la renderización) para aprovechar aun más la memoria.

Existe también el concepto de las bounding boxes, las cuales son cajas de recubrimiento para los modelos. Se utilizan para las colisiones en vez de los modelos puesto que la cantidad de vértices en los mismos conlleva una cantidad de procesamiento innecesaria. Hay dos categorías de bounding boxes dentro del CNEngine:

- El axis aligned bounding boxes (Aabb) y se define en coordenadas globales, almacenando solo los valores mínimos y máximo para cada eje del modelo.

- El oriented Bounding Box (obb) que son más eficientes puesto que tienen coordenadas locales y se ajustan al eje del modelo.

Estos conceptos (con muchos otros adicionales) se utilizan en la optimización de la aplicación gráfica, de las colisiones entre los objetos, el procesamiento y el manejo de memoria. Por último la librería matemática utiliza su acceso al lenguaje de bajo nivel para conocer las capacidades del hardware de la maquina en la que se esta ejecutando, para elegir que técnica de optimización utilizar.

Para realizar los cálculos necesarios de matemáticas 3D, la librería cn3d.h (la cual es la interfaz para acceder a la librería estática), tiene implementadas clases para las estructuras matemáticas básicas vinculadas al proceso, así como funciones para sus operaciones básicas, estas estructuras y funciones son:

Vector de tres dimensiones

- Suma
- Resta
- Multiplicación
- División
- Producto punto
- Producto cruz
- Normalizar
- Negación
- Longitud
- Rotar
- Angulo con otro vector

Matrices 3X3 y 4X4

- Calcular Identidad
- Rotar en X
- Rotar en Y
- Rotar en Z
- Rotar por un vector
- Trasladar
- Transpuestas
- Inversa
- Multiplicación
- Apuntar a un vector

Rayo

- Transformar
- Destransformar
- Intersección con triangulo
- Intersección con plano
- Intersección con aabb
- Intersección con obb

Plano

- Distancia a un punto
- Clasificar un punto
- Intersección con triangulo
- Intersección con plano
- Intersección con aabb
- Intersección con obb

Oriented Boundging Box (obb)

- Transformar
- Destransformar
- Intersección con triangulo
- Intersección con plano
- Intersección con rayo
- Intersección con obb

Axis Aligned Boundging Box (Aabb)

- Obtener Planos
- Intersección con plano
- Contiene un rayo
- Intersección con rayo
- Intersección con aabb
- Intersección con obb
- Intersección con aabb

Quaternion

- Construir a partir de ángulos eulerianos
- Normalizar
- Conjugar
- Crear matriz de rotación
- Magnitud
- Multiplicación
- División
- Rotar por un vector
- Rotar por un quaternion

4.5 MOTOR DE AUDIO

4.5.1 Diseño. Las librerías DirectMusic y DirectSound mantienen su principal diferencia en el nivel de programación, puesto que el DirectSound es de bajo nivel en comparación con DirectMusic, costandole a este último determinados niveles de control, y dependiendo el segundo de las habilidades expertas del programador para manejar código de más bajo nivel.

Otro factor importante es la utilización de manejadores, los cuales cargan los sonidos y músicas que se vayan a utilizar dentro del juego en el principio de la ejecución, instanciandola en el momento en el que se necesite.se aplica el principio de los managers, en el que se cargan los sonidos al principio de la aplicación y se almacenan en el cache, de manera que cuando se llamen sea solo buscarlos en la memoria.

Existen dos formas habituales de guardar efectos de sonido o música en un archivo. El primer método utiliza samples (datos de intensidad de sonido y digitalización del audio almacenados) digitalizados de los sonidos, y el segundo utiliza “instrumentos”. El primer método consume gran cantidad de memoria, pero su accesibilidad es rápida y efectiva. El segundo método almacena información acerca de los instrumentos que se usan en una pieza musical e información adicional referente al momento en el que deben usarse. En DirectX se utiliza el formato de sonidos descargables (DownLoadable Sounds Format, DLS), el cual utiliza ambos métodos de la siguiente forma: en tiempo de ejecución la información de los instrumentos es generada y se debe llamar esa información para que pueda ejecutarse sin retardos.

4.5.2 Desarrollo. El desarrollo del motor de audio se enfocó en la creación de las librerías estáticas y dinámicas con la utilización del DirectMusic y del DirectSound. La librería estática se encarga de crear el objeto que inicializará el Device, de manera que puedan accederse a las funciones implementadas en la librería dinámica, donde se ejecutan funciones del DirectInput, y se denominan respectivamente Claudio.lib y CNDA.dll. Se definen de igual forma que las librerías de renderización, diferenciandose en la implementación y en el componente de DirectX que se accesa. A continuación se hablará de las particularidades del motor.

El motor carga los sonidos y música de la siguiente manera: llama a la función Download del DirectMusic para cargar en memoria un archivo de sonido, después se debe crear un "camino" (path) de audio, que si no es un sonido en tercera dimensión se crea por defecto. Si es un sonido en tercera dimensión se debe almacenar información sobre la dirección y posición del sonido, además de la posición del usuario (el listener). El resto de la librería es un conjunto de funciones

implementadas que permiten inicializar un sonido, detener un sonido, llamar al Device, etc.

4.6 MOTOR DE ENTRADA/SALIDA

4.6.1 Diseño. El desarrollo del motor de entrada/salida inició como sus predecesores, con dos proyectos que generan las librerías estática y dinámica (el CNInput, y el CNDI respectivamente). El contenido de estos proyectos también se supeditó al diseño de la estructura ya planteada, en la que la librería estática que utiliza la implementación que se encuentra en el DLL, sea este el que se crea en el proyecto adicional, o una implementación realizada con otra librería (SDL, por ejemplo).

4.6.2 Desarrollo. En la implementación del CNDI se define la forma en que los diversos dispositivos de entrada envían información a través del DirectInput, información que puede ser almacenada en estructuras o variables. Esto se realiza en la clase base que implementa los métodos adecuados para extraer la información; sin embargo, cada dispositivo cuenta con su propia clase de manera que las particularidades de cada uno puedan implementarse.

Otro factor importante es la creación del Device, que se realiza por medio de un puntero que apunta a un objeto de la clase CNDIDevice, la cual crea a su vez un objeto del tipo IDirectInput8, el cual se conecta con los diversos dispositivos de entrada de forma directa.

Hay dos modos de llamar un dispositivo de entrada, el modo con buffer y sin buffer. El primero es una forma de asegurarse de que todas las entradas

realizadas por el usuario se almacenarán, mientras que el segundo solo almacena la tecla inmediatamente anterior.

La forma de implementación para el programador de esta clase puede realizarse al obtener la información de la locación a la que apunta el Mouse o el Joystick, además de las funciones que determinan si se mantiene presionada una tecla o se presiona para luego liberarlos.

4.7 MOTOR DE FÍSICAS

4.7.1 Introducción. Los motores de físicas en los videojuegos tienen una gran importancia en el impacto que el juego tiene sobre los jugadores, ya que en este siglo los jugadores esperan que los comportamientos de los objetos en el juego sean lo mas reales posibles, de modo que la inmersión en el videojuego sea duradera y no se sienta que simplemente se esta jugando, sino que se vive una experiencia similar a las vividas en el mundo no virtual.

Sin embargo durante la investigación para estructurar el motor de físicas del videojuego, se aprendió que las físicas simuladas en un motor de videojuegos, no son simulaciones de la física del mundo no virtual, se basan en las leyes físicas que explican el comportamiento de objetos en el mundo real, pero no funcionan de la misma manera. En los videojuegos estas leyes pueden alterarse, así como las propiedades de los objetos en el mismo, para darle a los jugadores una experiencia supuestamente “realista”, supuestamente porque la mayoría de video jugadores nunca han disparado un arma, un rayo láser o manejado un tanque; de modo que para el jugador el realismo no es representativo del mundo no virtual, sino de lo que han visto en otros mundos virtuales y lo que su imaginación les dice.

Para ilustrar mejor la idea anterior, un ejemplo muy común en los videojuegos de hoy son las municiones, cuando se dispara un arma en el mundo real, las personas no pueden ver la bala saliendo de la pistola y acercándose a ellas, pero en muchos videojuegos este es el caso, para lograr este efecto mucho mas emocionante en el videojuego, la velocidad de la bala tiene que disminuirse y por ende la gravedad sobre la misma también, para que esta no caiga al suelo antes de llegar a su objetivo. Una vez esta llega a su objetivo, por ejemplo un enemigo, el jugador espera que ese enemigo reaccione a la bala y se mueva hacia atrás por el impacto, para lograr esto, la bala debe tener una masa mucho mayor a los pocos gramos de una bala real y convertirse en una bala de 2 o 4 kilogramos.

4.7.2 Diseño. Para poder diseñar el motor de físicas, se investigo en 2 frentes, el primero consistió en buscar motores de física para descarga en Internet, el segundo fue una búsqueda en libros y artículos sobre motores de físicas.

En primera instancia, se encontraron varios motores de físicas en Internet, algunos libres, otros opensource, entonces se procedió a analizar 2 motores de físicas, el Newton y el Bullet. El newton es de los más usados en desarrollos principiantes, sin embargo no es opensource por lo que no contribuyo a la estructura del motor de físicas, el Bullet es otro motor de físicas muy usado a nivel mundial en desarrollos no profesionales, y a diferencia del newton si es opensource, al analizar las capacidades del Bullet con lo propuestos para el motor de físicas, se encontraron pocas coincidencias, además el código estaba poco estructurado u orientado a objetos.

Entonces se recurrió a la búsqueda en textos, como resultado de la misma, se encontró información sobre que tipos de motores de física hay, clasificados usando su modo de funcionar. Con base en esta información se decidió realizar un

motor de cuerpos rígidos, basados en impulsos y procesando en intervalos de frames, además de tratar los contactos de forma iterativa.

También se realizo una modificación de la estructura general del motor debido a esta investigación, se decidió implementar el motor de físicas en una librería estática y no en una dinámica, esto debido a dos razones de peso, la primera, que el motor de físicas no usa otra API, como los otros 3 motores que usan DirectX, por lo que no hay necesidad de encapsularlo en una librería dinámica, la segunda, es por rendimiento, el motor en general es mas rápido si el motor de físicas se encuentra en una librería estática, ya que de esta manera se incrusta en el ejecutable de un videojuego y no hay que cargarlo en tiempo de ejecución.

4.7.3 Desarrollo. El primer paso en el desarrollo del motor de físicas, fue el desarrollo de un submotor del motor de físicas, llamado motor de partículas, un motor de partículas se encarga de realizar todos los caculos de movimiento de partículas, la razón para iniciar por un motor de partículas es sencilla, no hay que tener en cuenta la geometría de las mismas, ya que son cuerpos puntuales, así como tampoco tienen orientación, es decir, no se puede decir hacia adonde apunta una partícula.

Antes de empezar con la teoría y física de partículas, se requirió implementar una serie de estructuras típicas de todo motor, esto resulto muy sencillo debido a que ya estaban implementadas en el motor de reenderizado, están estructuras son vectores, matrices 3x3, matrices 4x4 y Quaterniones.

Para desarrollar un motor de partículas, se deben tener en cuenta las leyes bajo las cuales estas se rigen, en este caso las leyes de mecánica clásica de Newton; es importante recordar que por partícula se entiende por un cuerpo puntual y no una partícula como un electrón, que si tienen volumen.

Se utilizaron las dos primeras leyes de Newton para el motor de partículas, la primera ley de Newton es la ley de inercia que dice que un cuerpo que viaja a una velocidad constante, lo seguirá haciendo hasta que una fuerza actué sobre el, la segunda ley dice que una fuerza que actúa sobre un cuerpo, produce en ese cuerpo una aceleración proporcional a su masa. De esta segunda ley se deriva la formula básica de fuerza:

$$F = m*a$$

Donde *F* es la fuerza ejercida sobre el objeto, *m* es la masa del objeto y *a* la aceleración causada por la fuerza que actúa sobre el objeto.

Para aplicar esta teoría a una partícula se creó la clase *Particle*, que contiene la información que se necesita conocer sobre una partícula, esta información se conforma de su posición, velocidad, aceleración y por ultimo su masa; en el mundo real no existen masas puntuales, sin geometría y volumen, pero para realizar los cálculos sobre las partículas y poder crear buenos efectos, es necesario asignarles una masa. Usando estos datos y la ecuación X.1, se puede calcular la aceleración causada por una fuerza actuando sobre una partícula, para posteriormente calcular su velocidad integrando la aceleración, ya que esta es la primera derivada de la velocidad, igualmente la velocidad es la primera derivada de la posición, por lo que podemos calcular la posición en función de la velocidad y aceleración usando la siguiente ecuación:

$$X = v*t + ((a\text{^}2)*t)/2$$

Donde *X* es la posición de la partícula, *v* es su velocidad, *a* es su aceleración y *t* es el intervalo de tiempo transcurrido. Ahora solo falta aplicarle fuerzas a las partículas para poder tenerlas en movimiento, para esto se usa un generador de fuerzas.

Un generador de fuerzas es un conjunto de reglas y condiciones que se usan para calcular la dirección y magnitud de una fuerza sobre una partícula, el generador de fuerzas mas simple es el que genera la gravedad, su conjunto de reglas dice que a cualquier partícula a la que se le asigne, se le aplica una fuerza vertical negativa con magnitud *g*, *g* es el valor de la gravedad que en la tierra es en promedio 9,8m/s^2.

En la práctica el generador de fuerzas es una interfaz que un programador de físicas en un videojuego debe implementar para definir sus propias reglas de generación de fuerzas, así como asignarle ese generador de todas las partículas que quiere que sean afectadas por ese tipo de fuerza. Esto genera un problema en el momento de asignarle más de un generador de fuerzas a una partícula, ya que no se sabe cuantos generadores de fuerza se asignaran a cada partícula, y no se pueden calcular los efectos de cada fuerza independientemente y luego sumarlos, sin embargo usando el principio de D'Alemberts, se pueden sumar estas fuerzas para obtener una fuerza neta y solo un efecto total sobre la partícula, con lo cual el problema queda resuelto.

Finalmente la estructura del motor de partículas queda de la siguiente forma:

- Una clase *Particle*, que contiene toda la información necesaria para realizar los cálculos matemáticos sobre ellas, así como los métodos para manipular esta información. Esta clase se encuentra definida en el archivo *Particle.h*.

- Una clase que sirve de interfaz para que el usuario implemente y defina sus propios generadores de fuerza, esta interfaz se llama *ParticleForceGenerator* y se encuentra definida en el archivo *pfgen.h*.

- Una clase llamada *ParticleForceRegistry* que se encarga de mantener las relaciones entre los generadores de fuerza y las partículas en el mundo virtual, esta clase se encuentra definida en el archivo *pfgen.h*.

- Una clase llamada *ParticleWorld*, cuya función es la de mantener referencia a todos los objetos del motor, esta realiza el ciclo de físicas del motor, donde se aplican todas las ecuaciones y actualiza la información de todas las partículas.

Adicionalmente el motor de partículas cuenta con varios generadores de fuerzas por defecto ya implementados, que corresponden a los más usados en el desarrollo de videojuegos. A continuación se enumeran y explican brevemente.

- **Generador de gravedad.** Este generador de gravedad funciona como el ejemplo mencionado previamente, le asigna una fuerza constante vertical negativa de valor *g* a todas las partículas a las cuales se les aplique este generador.

- **Generador de fricción.** Genera una fuerza constante de fricción a las partículas causada por el aire, esta fuerza es proporcional a la velocidad de la partícula y es aplicada en dirección opuesta al movimiento de la misma, esta fuerza y la velocidad están relacionadas por dos coeficientes *k1* y *k2*, *k2* esta relacionado a la velocidad como tal y *k2* al cuadrado de la misma. Estos coeficientes deben ser asignados al generador de fuerzas por el programador.

- **Generador tipo resorte.** Este generador usa la ley de Hook para calcular la fuerza que debe asignarle a la(s) partículas; hay dos versiones de este generador, la primera relaciona dos partículas con un resorte, esto significa que cada una de las dos partículas se encuentra en un extremo del resorte. La

segunda reemplaza una de las dos partículas por un punto fijo en el espacio tridimensional.

- **Generador tipo Bungee.** Como su nombre lo sugiere, este generador funciona como una cuerda de Bungee, funciona igual que un resorte, pero a medias, o sea que solo asigna fuerzas si se estira más allá de su longitud natural, más no si se comprime. Este generador también puede relacionar dos partículas o una sola y un punto estático.

- **Generador tipo Buoyancy.** Este generador sirve para simular el efecto de flotación de una partícula en un líquido, esta basado en un generador de tipo resorte y funciona para superficies paralelas al plano XZ.

Este es el motor de partículas en su totalidad, con un motor de estas características se pueden lograr diferentes efectos usados en los videojuegos, como lo son disparo de municiones, fuegos artificiales, explosiones, fuego, objetos livianos como hojas flotando sobre el agua y con un poco de trabajo efectos de movimiento de elementos como capas y banderas.

El siguiente paso en la construcción del motor fue expandir el motor de partículas a un motor de masas agregadas, esto fue relativamente sencillo, ya que el motor de masas agregadas se basa en el motor de partículas y lo usa en su totalidad. Sin embargo para logara esto se tuvo que agregar nuevas funcionalidades y aplicar nuevos conceptos al motor de partículas.

El primer concepto hacia un motor de masas agregas son las limitaciones fuertes, con un resorte podemos evitar que dos particular se alejen demasiado, pero se alejaran y acercaran eventualmente, si se construye un cubo con 8 partículas

unidas por resortes se obtendría una especia de figura suave, sin forma, en movimiento constante, que se aleja mucho de una caja típica en un fps. Para poder lograr mejores efectos, se introducen limitaciones fuertes, las cuales sirven para mantener las partículas a distancias constantes, pero para lograr este objetivo, no se pueden usar fuerzas, ya que estas cambian la posición de la partícula gradualmente por medio de su aceleración.

Para poder soportar limitaciones fuertes en el motor de físicas, lo primero que se requirió fue implementar un sistema de colisiones, este sistema se divide en dos pares, detección de colisiones y resolución de las mismas, para esto se uso la teoría de conservación del momentum o impulso. El momentum de un objeto esta descrito por su velocidad y masa, de la siguiente forma:

$$P = m*v$$

Donde *P* es el momentum de un objeto, *m* es la masa del mismo y *v* es su velocidad. La teoría de conservación del momentum dice que si dos objetos chocan el momentum neto, o sea la suma de sus dos momentum se mantiene constante después de la colisión, mas no su velocidad, además para las simulaciones suponemos que los objetos no se rompen y mantiene también su masa constante después del choque. Esta ecuación representa la conservación de momentum.

$$m1*v1 + m2*v2 = m1*v'1 + m2*v'2$$

Donde *m1* y *m2* son las masas de las partículas, *v1* y *v2* son sus velocidades antes del choque y *v'1* y *v'2* sus velocidades después del choque. Sin embargo con esta ecuación no se puede calcular las velocidades de las partículas después del choque, por lo que es necesario usar una constante llamada constante de restitución para determinar las velocidades después del choque, esta constante en

el mundo real depende de la composición de los objetos, en el mundo virtual, depende del programador del videojuegos y los efectos que quiera lograr. Esta constante relaciona la velocidad antes y la velocidad después del choque así:

$$v' = -c*v$$

Donde *v'* es la velocidad después del choque, *v* la velocidad antes del choque y *c* es la constante de restauración del objeto. Con estas dos formulas, se puede entonces calcular la velocidad de los objetos después el choque, solo falta resolver situaciones en las que dos choques se presenten al tiempo, sin embargo por el mismo diseño del motor esto ya esta implícitamente solucionado, como ya se menciono anteriormente, se decidió tratar los contactos de forma iterativa, de modo que se resuelve cada colisión una por una en orden de importancia, entre mas rápido se acerquen los objetos entre si, mayor es la importancia de resolver esa colisión, para no tomar el riesgo de que los objetos se traspasen.

Teniendo el motor la posibilidad de resolver colisiones entre objetos, ahora es necesario detectar estas colisiones para poder resolverlas, el sistema de detección de colisiones se encarga de ejecutar los algoritmos de colisión y de especificar en que puntos se dio esta colisión, para posteriormente resolver cada contacto por separado. Para la colisión entre partículas, es necesario dejar de tratarlas como partículas y hacerlo como micro esferas, de lo contrario la única manera de que colisionara seria que las dos partículas estuvieran en el mismo punto del espacio, y en un espacio tridimensional constituido por los números reales, la probabilidad de que esto ocurra es demasiado baja.

Se encontraron dos problemas serios al tratar con la detección de colisiones, el primero es la interpenetración, esto sucede cuando dos objetos no solo hacen contacto superficialmente sino que se traspasan antes de poder detectarse una colisión debido a su velocidad, si se trata de forma normal estas colisiones, las

velocidades después de la colisión pueden no ser suficientemente grandes y en el jugador va a ver entonces que los objetos se traspasan antes de separarse en las colisiones, para resolver este problema, el sistema de detección y el de resolución de colisiones deben trabajar juntos, el primero debe detectar de la interpenetración y el segundo debe calcular la velocidad extra para evitar el efecto de la interpenetración.

El segundo problema son los contactos de descanso, estos se dan cuando un objeto descansa sobre otro de forma estática, el problema surge cuando se detecta cierto nivel de interpenetración entre los dos objetos y el sistema de resolución de colisiones los separa, inmediatamente después la gravedad los vuelve a unir y en ese momento se forma un ciclo que hace que los objetos salten de la tierra y en el mejor de los casos vibren cuando están sobre otro. Para solucionar este problema hay que identificar cuando una colisión es un contacto de descanso y tratarlo de manera especial para evitar los efectos anteriormente mencionados.

Con el sistema de colisiones listo, el motor de partículas ahora tiene funcionalidades mucho más realistas y complejas, además esta listo para soportar limitaciones fuertes y poder simular masas agregadas. Estas limitaciones fuertes pueden mantener partículas a distancias fijas por medio de la generación de contactos ficticios entre ellas, ya que estos contactos se traducen en impulsos y alteran directamente la velocidad de las partículas, en contraste con las fuerzas y la aceleración que lo hacen gradualmente.

De igual forma que las fuerzas son creadas con los generadores de fuerza, los contactos son creados con generadores de contactos y un programador puede implementar con sus propias reglas de generación, se decidió agregar al motor dos tipos de limitaciones fuertes, con el fin de poder crear objetos tridimensionales a partir de partículas, estos son varillas y cables, ambos pueden funcionar

uniendo dos partículas o una partícula y un punto fijo en el espacio. Las varillas tienen una longitud fija y si las partículas tratan de alejarse o acercarse más allá de esta longitud, se genera un contacto para evitarlo y mantenerlas a distancia, de forma similar, los cables tiene una longitud fija, pero solo generan contactos si las partículas se alejan mas de esta longitud.

Ahora la estructura del motor de físicas puede simular partículas y objetos de masas agregadas, y tiene la siguiente estructura:

- Una clase *Particle*, que contiene toda la información necesaria para realizar los cálculos matemáticos sobre ellas, así como los métodos para manipular esta información. Esta clase se encuentra definida en el archivo *Particle.h*.

- Una clase que sirve de interfaz para que el usuario implemente y defina sus propios generadores de fuerza, esta interfaz se llama *ParticleForceGenerator* y se encuentra definida en el archivo *pfgen.h*.

- Una clase llamada *ParticleForceRegistry* que se encarga de mantener las relaciones entre los generadores de fuerza y las partículas en el mundo virtual, esta clase se encuentra definida en el archivo *pfgen.h*.

- Una clase llamada *ParticleContact* que se encarga de contener toda la información necesaria sobre un contacto entre dos partículas, esta clase se define en el archivo *pcontacts.h*.

- Una clase llamada *ParticleContactResolver* que se encarga de resolver uno a uno los contactos generados en el mundo virtual, esta clase se define en el archivo *pcontacts.h*.

- Una interfaz llamada *ParticleContactGenerator*, que se encuentra también en el archivo *pcontacts.h* y sirve para que el usuario cree sus propios generadores de contactos.

- Una clase llamada *ParticleWorld*, cuya función es la de mantener referencia a todos los objetos del motor, esta realiza el ciclo de físicas del motor, donde se aplican todas las ecuaciones y actualiza la información de todas las partículas, un objeto de mundo maneja todos los generadores de fuerzas y de contactos, así como el sistema de resolución de colisiones, esta clase se encuentra definida en el archivo *pworld.h*.

Con el motor en este punto, ahora es posible simular también objetos tridimensionales, como cajas, muebles, paredes, pirámides y en general cualquier poliedro, por medio de uniones de partículas, y adicionalmente darle efectos motrices a estos objetos por medio de fuerzas e impulsos. Los objetos creados a partir de partículas pueden rotar aun cuando no se ha implementado nada específico para conseguirlo, ya que el movimiento independiente de cada partícula logra que el objeto como un todo rote en el espacio. A pesar de la versatilidad del motor de masas agregadas, es bastante complejo y dispendioso el proceso de simular todos los objetos de un videojuego a partir de partículas conectadas, además los objetos pueden presentar comportamientos extraños de vibración en situaciones de muy altas velocidades y por ultimo el motor carece de fricción, el cual es un elemento muy importante para la sensación de realismo de un videojuego. Un motor de masas agregadas generalmente no llena todas las expectativas del desarrollo de videojuegos profesionales, sin embargo muchos videojuegos los usan en conjunto con motores de cuerpos rígidos para mejorar los efectos físicos del mismo.

5. ESTADO DEL ARTE DE LAS METODOLOGÍAS DE INGENIERÍA DE SOFTWARE PARA DESARROLLO DE VIDEOJUEGOS

Se ha hecho una investigación exhaustiva respecto a las metodologías de desarrollo alrededor de los videojuegos con el propósito de estructurar un artículo que las resuma, y permitirle a los desarrolladores de juegos alrededor del mundo (y en especial en Latinoamérica) una guía adecuada para que el desarrollo de videojuegos se realice a través de un modelo estructurado que permita cumplir con plazos y con los objetivos en su totalidad.

Entre los resultados de la investigación se ha encontrado que se utilizan diversos estándares ya utilizados por la ingeniería de software, tales como los casos de uso, el UML (Unified Modelling Language), el desarrollo orientado a objetos, entre otros.

Debido a que los videojuegos son creados por una gran cantidad de personas con distintos niveles de talento y creatividad, no se puede hablar de un proceso estrictamente de ingeniería, por lo que no puede exigírsele a las personas un conocimiento en este tipo de desarrollo. El software debe estar listo en determinada fecha, por tanto es responsabilidad del ingeniero encargado de cumplir con las pautas requeridas en el tiempo planteado, y comunicarse con personas fuera del contexto de la programación de manera que entiendan la metodología de trabajo y puedan aplicarla en conjunto con su creatividad.

También se han encontrado casos en los que el hecho de que la visión de un producto artístico interfiere con la rigurosidad del paquete final. Casos como la división de grupos de desarrollo de un mismo proyecto debido a la divergencia en

la visión de los líderes del proyecto, tiempo que luego sería perdido debido a la necesidad intrínseca de tener un solo equipo de desarrollo y una sola

visión[25]. De esta forma se hace evidente que la implementación de una metodología de desarrollo rigurosa es indispensable en proyectos de esta índole.

La ingeniería de software dentro de los videojuegos mantiene diferencias, tanto por la conformación del equipo como por el objetivo del mismo, pero los parámetros generales bajo los que deben regirse ambos desarrollos es la misma. Entre estos parámetros podemos encontrar:

Alcance: Abarca los requerimientos que determinan que debería poder hacer el producto final. El grupo de requerimientos permite entender el alcance del mismo y los objetivos que se desean alcanzar.

- **Arquitectura**. Expresa el orden lógico y físico del diseño.

- **Planeación**. Designa la secuencia de actividades que deben realizarse de manera que se pueda elaborar el producto deseado.

- **Implementación**. es el desarrollo del producto dentro del alcance planteado por los requerimientos, de acuerdo al diseño planteado por la arquitectura y en concordancia con el plan de desarrollo.

- **Pruebas**. verifica la validez del producto tal y como se entrega posterior al proceso de desarrollo.

[25] ZERBST, Stefan; DUVEL, Oliver. 3D Game Engine Programming, pag 16.

6. CONCLUSIONES

La investigación alrededor de las APIs gráficas, su utilización en un motor de videojuegos y su creación, la investigación alrededor de la ingeniería de software en videojuegos y la creación de un videojuego nos ha llevado a las siguientes conclusiones:

Los motores de videojuegos deben ser completamente autónomos de la aplicación que se desarrolle con el mismo (el desarrollador no tiene que escribir código que modifique motor, solo el videojuego), debido a que el motor debe poder aplicarse a diferentes productos con distintos enfoques.

Assembler es el lenguaje de programación con mayor capacidad de optimizar (ejecutar la mayor cantidad de operaciones en menor tiempo) las operaciones del CPU.

La programación de bajo nivel es una constante el desarrollo de videojuegos, ya sea a través de código Assembler o código de programación de shaders.

La utilización de C++ es una opción necesaria para la optimización del código y del adecuado manejo de memoria VRAM (memoria RAM del GPU), siendo, para la creación de motores, una opción superior a lenguajes como JAVA o C#.

Las aplicaciones multimedia pueden tener distintos enfoques y objetivos, por lo que los distintos módulos dentro de un motor permiten al desarrollador utilizar las herramientas (de cada módulo) que necesite y dejar a un lado las que no.

Los desarrollos de este tipo deben escoger entre la alta complejidad de un motor (módulo) o la creación integral de motores que permitan la creación de aplicaciones interactivas.

Las metodologías de desarrollo de videojuegos están supeditadas al desarrollo de software tradicional, pero ajustándose al concepto del videojuego, puesto que no se resuelve un problema sino que se busca comercializar un producto de entretenimiento.

Los motores de videojuegos de libre utilización disponibles se enfocan primordialmente en el motor gráfico, y en la mayoría de las ocasiones ignoran la implementación de los demás módulos.

Las clases de física, algebra y calculo cursadas durante el transcurso de la carrera, tiene una amplia aplicación directa en el desarrollo de motores para videojuegos, así como para desarrollar también videojuegos.

Las físicas en un videojuego están diseñadas para dar efectos que le parezcan reales y emocionantes a los jugadores, más no están diseñadas para simular lo que sucede en el mundo real.

Existen muchas metodologías y aproximaciones para desarrollar un motor de físicas, diseñadas para dar soluciones a diferentes requerimientos de parte de los desarrolladores de videojuegos, algunos videojuegos requieren físicas tan especializadas que un motor de físicas genérico no daría los resultados esperados y en cambio gastaría mas recursos de la maquina que ejecuta el juego.

Construir soluciones escalables y bien estructuradas en cuanto a orientación a objetos es fundamental en el desarrollo de motores para videojuegos, debido a la necesidad de los desarrolladores de un videojuego de adaptar el motor a sus necesidades, esta característica influye mucho al momento de decidir que motor usar para el desarrollo de un videojuego.

Es importante tener en cuenta bibliografía complementaria en el desarrollo de un proyecto, en este caso la falta de bibliografía especializada en el desarrollo de motores, fue suplida por bibliografía de calidad especializada en el desarrollo de videojuegos o en los API especializados en el desarrollo de los mismos.

En cuanto a simulación de comportamientos físicos, en los videojuegos es muy útil conocer y entender los fundamentos físicos, de modo que se puedan hacer alteraciones a los modelos reales en función de obtener simplicidad y desempeño.

La falta de conocimiento sobre el desarrollo de videojuegos a nivel regional, dificulto en gran medida el proceso de evaluación del motor por parte de expertos en el tema. Además del tiempo que se requiere para evaluar un software de este

tipo, ya que su desempeño se mide en la realización de videojuegos, lo cual es un proceso que puede tardar de meses a años.

7. RECOMENDACIONES

Para futuros proyectos un amplio conocimiento en C++ es una condición inevitable para el desarrollo ya sea de motores o videojuegos.

Conocimientos en programación sobre Win32 son una gran facilidad en el desarrollo de este tipo de aplicaciones.

Experiencia en la creación de librerías estáticas y dinámicas reducen el tiempo de aprendizaje de los conceptos por medio de la aplicación rápida y directa (para instrucciones sobre como crear una librería estática y una dinámica con Visual Studio 2005 ver anexo 1).

Una complementación idónea sería la creación de librerías dinámicas que implementen las clases y funciones por medio de OpenGL, OpenAL y otras librerías libres de manera que el desarrollador tuviera la opción de escoger el tipo de desarrollo con el que se sienta cómodo.

REFERENCIAS BIBLIOGRÁFICAS

ADAMS, Jim. Programming Role Playing Games with DirectX. Segunda edición. WordWare Publishing, 2004. 849 p. Game Development. ISBN: 1-59200-315-X.

-------- . -------- . Primera edición. WordWare Publishing, 2002. 1056 p. Game Development. ISBN: 1-931841-09-8.

BETHKE, Erik. Game Development and Production. Primera edición. WordWare Publishing, 2003. 412 p. ISBN 1-55622-951-8

EBERLY, David. 3D Game Engine Design: A practical approach to real time computer graphics. Morgan Kauffman Publishers, 2000. 561 p. The Morgan Kaufmann Series in Interactive 3D Technology. ISBN: 978-1-55860-593-0

ENGEL, Wolfgang F (editor). Vertex and Pixel Shader Tips and Tricks. Wordware Publishing, Inc. 494 p. 2002. ISBN 1-55622-041-3.

MCCUSKEY, Mason. Beginning Game Audio Programming. Premier Press. 352 p. 2003. ISBN:1592000290.

MILLINGTON, Ian. Game physics engine development. Morgan Kaufmann Publishers. 456 p. 2007. ISBN-13: 978-0-12-369471-3.

PEDERSEN, Roger E. Game design foundations, Wordware Publishing. 2003. ISBN: 1-55622-973-9.

PRESSMAN, Roger. Ingeniería de Software, Un Enfoque Práctico. McGraw Hill. Quinta Edición. 2001. ISBN: 84-481-3214-9

ROLLINGS, Andrew, ADAMS, Ernest. Andrew Rollings and Ernest Adams on Game Design. New Riders Publishing, 2003 648 p. ISBN : 1-592-73001-9.

RUCKER, Rudy. Software Engineering and Computer Games. Addison Wesley. Primera edición. 2002. 544 p. ISBN: 0-201-76791-0.

SÁNCHEZ-CRESPO, Daniel. Core Techniques and Algorithms in Game Programming. New Riders Publishing. 2003. 888 p. ISBN: 0-1310-2009-9.

WALSH, Peter. Advanced 3D Game Programming with DirectX. WordWare Publishing, 2003. 610 p. ISBN-10: 1556227213

ZERBST, Stefan; DUVEL, Oliver. 3D Game Engine Programming. Thomson Course Technology, 2004. Game Developers Series. 896 p. ISBN-10: 1592003516

ANEXOS

Anexo A. Creación del proyecto en Visual Studio 2005:

Para poder generar la librería dinámica se debe crear un proyecto con las siguientes especificaciones:

- Especificar que tipo de configuración debe crear una librería dinámica (**Dynamic Link Library**).
- El **set** de caracteres debe usar el **set** de **Multi-bytes**.
- La opción de optimización debe deshabilitarse.
- La opción de formato de **debug** debe ser "**Program database for edit and Continue" /Zl.**
- Debe permitir reconstrucción mínima.
- Debe permitir **debug** multihilos y ambos tipos de chequeo en tiempo de ejecución.