DISEÑO E IMPLEMENTACION DE UNA BIBLIOTECA DE CLASES EN C++ PARA LA SIMULACION DE AMBIENTES VIRTUALES EN TIEMPO REAL

MAURICIO ANTONIO FRANCO MARTÍNEZ

HENRY AUGUSTO VILLAMIZAR RUEDA

CORPORACIÓN UNIVERSITARIA TECNOLÓGICA DE BOLÍVAR

FACULTAD DE INGENIERÍA DE SISTEMAS

CARTAGENA DE INDIAS

2002

DISEÑO E IMPLEMENTACION DE UNA BIBLIOTECA DE CLASES EN C++ PARA LA SIMULACION DE AMBIENTES VIRTUALES EN TIEMPO REAL

MAURICIO ANTONIO FRANCO MARTÍNEZ

HENRY AUGUSTO VILLAMIZAR RUEDA

Tesis para optar al título de

Ingenieros de Sistemas.

Director

MOISÉS QUINTANA ALVAREZ

MSC en Informática

CORPORACIÓN UNIVERSITARIA TECNOLÓGICA DE BOLÍVAR

FACULTAD DE INGENIERÍA DE SISTEMAS

CARTAGENA DE INDIAS

2002

Artículo 107.

La Universidad Tecnológica de Bolívar, se reserva el derecho de propiedad intelectual de todos los trabajos de grado aprobados y no pueden ser explotados comercialmente sin su autorización.

Nota de aceptación

____

____

____

____

Presidente del Jurado

____

Jurado

____

Jurado

Cartagena, 22 de mayo de 2002

Cartagena de Indias, 20 de Mayo de 2002

Señores

CORPORACIÓN UNIVERSITARIA TECNOLÓGICA DE BOLÍVAR

COMITÉ DE EVALUACIÓN

L.C.

Respetados Señores:

Cordialmente me dirijo a ustedes, para informarles que el trabajo de grado titulado **DISEÑO E IMPLEMENTACIÓN DE UNA BIBLIOTECA PARA LA SIMULACIÓN DE AMBIENTES VIRTUALES EN TIEMPO REAL** ha sido desarrollado de acuerdo a los objetivos establecidos.

Como director del proyecto considero que el trabajo es satisfactorio y amerita ser presentado por sus autores.

Atentamente,

MOISÉS QUINTANA

LICENCIADO EN MATEMÁTICAS

MASTER EN INFORMÁTICA APLICADA

FACULTAD DE INGENIERÍA DE SISTEMAS

DIRECTOR DEL PROYECTO

Cartagena de Indias, 20 de Mayo de 2002

Señores

CORPORACIÓN UNIVERSITARIA TECNOLÓGICA DE BOLÍVAR

COMITÉ DE EVALUACIÓN

L.C.

Respetados Señores:

Cordialmente me dirijo a ustedes, para informarles que el trabajo de grado titulado **DISEÑO E IMPLEMENTACIÓN DE UNA BIBLIOTECA PARA LA SIMULACIÓN DE AMBIENTES VIRTUALES EN TIEMPO REAL** ha sido desarrollado de acuerdo a los objetivos establecidos.

Como autores del proyecto consideramos que el trabajo es satisfactorio y amerita ser presentado ante ustedes.

Agradeciendo su atención a la presente,

_____________________________	_____________________________
MAURICIO FRANCO MARTÍNEZ	HENRY VILLAMIZAR RUEDA

AGRADECIMIENTOS

Los autores expresan sus agradecimientos a:

A Moisés Quintana, su director.

A Eduardo Gómez, Gonzalo Garzón, Juan Carlos Mantilla, Margarita Upegui, Luis Eduardo Rueda, Juan Martínez, Jaime Arcila, Giovanni Vásquez y a los demás profesores de la facultad por sus ideas, conocimientos e inspiración.

A sus familias, por el apoyo, la fe y la motivación.

CONTENIDO

	pág.
INTRODUCCIÓN	10
1. CONCEPTOS PRELIMINARES	15
1.1 INTRODUCCIÓN A LA COMPUTACIÓN GRÁFICA	15
1.1.1 Imagen	15
1.1.2 Animación	16
1.1.3 Primitivas	19
1.2 GRÁFICOS TRIDIMENSIONALES	20
1.2.1 Proyecciones	20
1.2.2 Transformaciones	21
1.2.3 Iluminación y materiales	23
1.2.4 Mapeado de texturas	28
1.2.5 Eliminación de superficies ocultas	29
1.2.6 APIs de bajo nivel y alto rendimiento	29
2. FUNDAMENTACIÓN TEÓRICA	31
2.1 EL MODELO TRIDIMENSIONAL	31
2.1.1 Estructuras básicas	31
2.1.1.1 El vector	32

2.1.1.2 El color 32

2.1.1.3 El material 32

2.1.1.4 La cara 33

2.1.2 Optimización 33

2.2 LA CÁMARA 34

2.3 ROTACIONES EN EL ESPACIO 37

2.3.1 Cuaternios 38

2.3.2 Interpolación entre rotaciones 40

2.4 DETERMINACIÓN DE LA VISIBILIDAD 41

2.4.1 Herramientas 41

2.4.1.1 Subdivisión jerárquica 41

2.4.1.2 Conjuntos potencialmente visibles 42

2.4.1.3 Coherencia espacial y temporal 42

2.4.2 Algoritmos 43

2.4.2.1 Descarte de caras traseras (Back face culling) 43

2.4.2.2 Subdivisión espacial mediante árboles de ocho hijos (Octrees) 43

2.4.2.3 Descarte de polígonos ocultos (Occlusion culling) 44

2.4.2.4 Niveles de detalle (LOD) 44

2.5 TERRENOS 45

2.5.1 Representación y renderización 45

2.5.2 Generación de terrenos aleatorios 46

2.5.3 Recorrido de grandes terrenos en tiempo real 47
2.6 SIMULACIÓN FÍSICA 48
2.6.1 Solución numérica de ecuaciones diferenciales 49
2.6.1.1 El método de Runge-Kutta 49
2.6.2 Dinámica de partículas 51
2.6.3 Dinámica de cuerpos rígidos 52
2.6.3.1 Resolución de colisiones 55
2.6.3.1.1 Contacto de colisión 56
2.6.3.1.2 Contacto de reposo 58
2.6.3.2 Detección de colisiones 59
2.6.3.2.1 Determinación de intersecciones 59
2.6.3.2.2 Determinación para n objetos 61
2.6.3.2.3 Determinación del tiempo exacto de colisión 62
3. DISEÑO DE LA BIBLIOTECA DE CLASES 64
3.1 OBJETIVOS 64
3.1.1 Objetivo general 64
3.1.2 Objetivos específicos 64
3.1.3 Análisis de los objetivos 65
3.2 DISEÑO DEL SISTEMA 66
3.2.1 Diseño general del sistema 66
3.2.2 Diseño de clases 68

3.2.2.1 Sistema de transformación 68

3.2.2.2 Modelos 69

3.2.2.3 Simulación física 70

3.3. IMPLEMENTACIÓN 70

3.3.1 Detalles técnicos 70

3.3.2 Módulos 71

3.3.3 Estilo del código 72

4. CONCLUSIONES 74

4.1 RESULTADOS 74

4.1.1 Resultados obtenidos usando el descarte de geometría 74

4.1.2 Resultados obtenidos en la simulación física 75

4.1.3 Otros resultados 76

4.2 CONCLUSIÓN 78

4.3 RECOMENDACIONES 80

4.3.1 Aplicabilidad 80

4.3.1.1 Arquitectura 80

4.3.1.2 Geografía 80

4.3.1.3 Medicina 80

4.3.1.4 Educación e investigación 80

4.3.1.5 Ingeniería 81

4.3.1.6 Otras 81

4.3.2 Proyección 81
4.3.2.1 Descarte y optimización de geometría 82
4.3.2.2 Simulación física 82
4.3.2.3 Sistemas distribuidos 83
4.3.2.4 Interfaces electrónicas 83
4.3.2.5 Animación 83
4.3.2.6 Simulación biológica 84
4.3.2.7 Mejoras en el código 84
4.3.2.8 Audio 84
BIBLIOGRAFÍA 85

LISTA DE FIGURAS

pág.

Figura 1. Flujo de transformación. 34

Figura 2. Flujo de transformación con cámara. 35

Figura 3. Flujo general del sistema. 67

RESUMEN

La realidad virtual es una herramienta poderosa que nos permite visualizar, recorrer y simular aspectos de nuestro entorno de forma interactiva. Un sistema mínimo, capaz de simular ambientes virtuales en tiempo real debe proveer las siguientes funcionalidades:

Cargar y manipular modelos tridimensionales en el espacio.
Permitir el recorrido de ambientes virtuales complejos en tiempo real.
Simular el comportamiento de los objetos basándose en las leyes físicas del entorno.

Buscando alcanzar tales funcionalidades, este trabajo presenta un conjunto de técnicas y algoritmos, implementados en una biblioteca de clases, mediante la cual programadores e ingenieros puedan construir aplicaciones para realidad virtual de forma rápida y fácil.

Para aprovechar la aceleración de video ofrecida por el hardware actual, el despliegue de gráficos tridimensionales en la pantalla es realizado a través de la biblioteca de renderización OpenGL.

La manipulación de modelos tridimensionales en el espacio se alcanza mediante la implementación de un sistema de transformación espacial basado en el álgebra de vectores, matrices y cuaternios.

Para manejar ambientes virtuales complejos cuya geometría excede la capacidad de procesamiento de la máquina, se emplean técnicas para descartar los polígonos innecesarios de una escena y permitir su recorrido en tiempo real. Las técnicas implementadas son: descarte del campo de visión y subdivisión espacial mediante octrees (árboles de ocho hijos). Para el recorrido de terrenos grandes en tiempo real, se implementó una versión del algoritmo propuesto por [Rot98].

Para establecer las leyes físicas del entorno, se resolvieron dos problemas principales: la detección de colisiones entre los modelos y la simulación de la dinámica de las entidades físicas. El primero mediante el uso de volúmenes envolventes más simples y el segundo mediante la resolución progresiva de las ecuaciones diferenciales de la dinámica.

INTRODUCCIÓN

La realidad virtual es una herramienta poderosa que nos permite visualizar, recorrer y simular aspectos de nuestro entorno de forma interactiva y en tiempo real. Sus beneficios son apreciables en todas las áreas del conocimiento humano.

Para cumplir con sus propósitos, un sistema utópico de realidad virtual debe proveer varias características: una simulación física y biológica del universo en tiempo real, y una interfaz capaz de engañar a los sentidos humanos.

Actualmente no existen tales sistemas de inmersión, pero en el transcurso de los últimos años la tecnología ha avanzado a pasos agigantados permitiendo el desarrollo de sistemas rudimentarios de realidad virtual. Con el poder computacional de las máquinas actuales se han logrado implementar simulaciones físico-mecánicas en tiempo real; la ciencia naciente de la inteligencia artificial está comenzando a simular comportamientos inherentemente biológicos y las compañías de hardware están desarrollando interfaces cada vez más acoplables al ser humano.

Aún dentro de sus actuales limitaciones, la realidad virtual resuelve muchas necesidades de diversas áreas como: la ingeniería, la milicia, la docencia, las ciencias físicas y naturales, el entretenimiento, la manufactura y la medicina, entre otras.

Según Donald Gillies (gillies@ee.ubc.ca), profesor de ingeniería eléctrica de la Universidad de British Columbia, un sistema en tiempo real es aquel en el que la validez de los cómputos no solo depende de la validez lógica de los cómputos, sino también del tiempo en el que el resultado es producido. Si las condiciones de tiempo del sistema no son alcanzadas, el sistema falla.

Para poder engañar a los sentidos humanos, la realidad virtual debe actuar como un sistema en tiempo real. Mediante una tarea periódica, el sistema debe recibir, procesar y enviar información de regreso a la interfaz humana, si el sistema presenta una latencia percibible, este falla.

El código de una aplicación típica de realidad virtual es el siguiente:

```
Iniciar_aplicación();
Preparar_el_sistema();

Repetir indefinidamente
{
 Leer_datos_de_la_interfaz();

 Actualizar_sistema_físico();
 Actualizar_sistema_biológico();
```

```
 Enviar_datos_a_la_interfaz();
}

Terminar_aplicación();
```

En la mayoría de aplicaciones, el periodo del sistema (tiempo en el que realiza una tarea periódica) debe ser impercibible para los sentidos. Por ejemplo, para crear el efecto visual del movimiento, el sistema debe ser capaz de producir animaciones de por lo menos 30 cuadros por segundo. Si un periodo no logra completarse en un tiempo límite de 1/30 de segundo, la percepción visual se degrada.

Para el presente trabajo se asume un sistema mínimo de realidad virtual, cuya interfaz hombre-computador está compuesta por el monitor, el ratón y el teclado. En este caso, la única salida del sistema es a través de imágenes tridimensionales, las cuales cambian dependiendo de una simulación física básica y de la interacción del usuario.

La tecnología actual no permite desplegar gráficos tridimensionales en el espacio real (hologramas), por lo tanto, estos son proyectados en pantallas bidimensionales, a partir de las cuales se visualiza el mundo virtual. Proyectando un modelo tridimensional en dos imágenes bidimensionales, una para cada ojo, se puede crear el efecto de profundidad necesario para engañar a la mente humana.

Todos los sistemas actuales de realidad virtual se valen de una técnica llamada "renderización" para generar imágenes fotorealísticas bidimensionales a partir de modelos tridimensionales.

Un sistema de renderización recibe la especificación de un modelo tridimensional compuesto por: polígonos, materiales, luces, texturas y su transformación deseada (rotación, traslación, escalamiento) y produce una imagen bidimensional del modelo proyectado.

Para implementar la renderización como sistema de visualización de la realidad virtual, el sistema debe generar varias imágenes por segundo para producir una animación convincente. Debido a que el proceso de renderización es computacionalmente costoso, para ciertos modelos tridimensionales las condiciones de tiempo real del sistema pueden verse amenazadas. Un sistema apropiado de realidad virtual debe proveer esquemas de optimización para el proceso de renderización.

El principal aporte de este trabajo se resume en varios puntos:

1. Un sistema mínimo de realidad virtual implementado en una biblioteca de clases en C++ que contiene las siguientes características:
    - Carga y manipulación de modelos tridimensionales en el espacio.

- Simulación del comportamiento de una cámara mediante la cual se permite el recorrido del mundo virtual.
- Optimización del proceso de renderización basada en métodos de descarte de geometría para permitir el recorrido de ambientes virtuales complejos en tiempo real.
- Implementación de un modulo para la visualización y recorrido de mapas de elevaciones terrestres.
- Simulación física de la dinámica de partículas y cuerpos rígidos.

2. Un documento teórico que reúne los métodos más efectivos y novedosos en cada campo del trabajo.

3. La recopilación de una bibliografía anexa en el CDROM, que queda como material de estudio para los profesores y estudiantes.

4. El diseño de un sistema de realidad virtual y la implementación de su parte básica, que quedan como un proyecto abierto a la comunidad universitaria.

1. CONCEPTOS PRELIMINARES

1.1 INTRODUCCIÓN A LA COMPUTACIÓN GRÁFICA

1.1.1 Imagen. Las imágenes digitales están compuestas por píxeles. Un píxel es el punto de color más pequeño que se puede apreciar en un monitor. Para construir un color se usan tres canales: rojo, verde y azul. Cada canal se puede ajustar a 256 intensidades diferentes, al combinar los tres canales y variar la intensidad de cada uno, se obtiene un color. La cantidad de colores que se pueden construir con tres canales de 256 intensidades cada uno, está dada por 2563 = 16'777.216 colores. Al número de bits necesarios para representar un píxel de una imagen en memoria se le conoce como profundidad de la imagen. Una imagen en blanco y negro solo necesita un bit para representar un píxel, mientras que una imagen de 16 millones de colores necesita 24 bits (ocho bits por canal). Adicionalmente, se puede usar un canal adicional llamado canal alpha para representar la opacidad o transparencia de un píxel. Las funciones encargadas de manipular y escribir los píxeles en la pantalla las provee el sistema operativo, el cual, a través de el adaptador gráfico accede a las funciones del monitor.

1.1.2 Animación. Por animación se entiende movimiento. En el computador, al igual que en el cine, el movimiento se logra mediante la proyección de una secuencia de imágenes estáticas a gran velocidad (24 veces por segundo). Dos problemas se deben evitar a la hora de desplegar una animación: el parpadeo y los saltos en la imagen. Obsérvese el siguiente ciclo de animación:

```
for (i=0; i < NumeroDeCuadros; i++)
{
 Limpia_la_pantalla();
 Dibuja_el_cuadro(i);
}
```

El parpadeo ocurre porque delante del espectador se está borrando, actualizando y dibujando nuevamente la imagen. Esto se corrige usando una memoria auxiliar (buffer) para construir la imagen y otra para desplegarla, de esta forma, mientras se construye la nueva imagen, el espectador esta viendo la imagen anterior, cuando la imagen está lista se transfiere a la pantalla. El ciclo de animación usando doble “buffer” queda de la siguiente forma:

```
for (i=0; i < NumeroDeCuadros; i++)
{
 // Prepara la imagen
 Limpia_el_buffer();
```

```
 Dibuja_el_cuadro(i);
 // Transfiere la imagen a la pantalla
 Intercambia_los_buffers();
}
```

Los saltos en la imagen ocurren porque los monitores de tubo de rayos catódicos (CRT) se deben sincronizar con la animación. Cuando los monitores escriben una imagen no lo hacen instantáneamente, se toman aproximadamente 1/60 segundos para hacerlo. Al número de veces por segundo que un monitor refresca la imagen se le llama tasa de refresco.

Dentro del monitor existe un tubo al vacío llamado tubo de rayos catódicos. El cátodo del monitor se encuentra en la parte trasera, en donde existe una pistola que dispara electrones directo al vidrio del monitor. Esta pistola contiene tres cañones, uno para cada canal de colores. Los electrones que disparan son desviados mediante manipulación magnética y repartidos en toda la pantalla. Estos, al chocar con la pantalla golpean una celda de fósforo que se ilumina dependiendo de la intensidad con la que sean disparados los electrones. Debido a que el monitor solo puede disparar un electrón a la vez, los cañones tienen que recorrer toda la pantalla para actualizarla completamente.

El patrón que siguen los cañones para actualizar la imagen es el siguiente: primero se ubican en la esquina superior izquierda, actualizan los píxeles de izquierda a derecha,

luego se devuelven hasta la segunda línea a la izquierda. Durante este tiempo los cañones no escriben nada en la pantalla, a este instante en el que se devuelven de derecha a izquierda se le llama retrazo horizontal. Una vez posicionados en la segunda línea a la izquierda, comienzan nuevamente el recorrido y continúan hasta que llegan a la esquina inferior derecha del monitor, entonces se devuelven de la esquina inferior derecha hasta la esquina superior izquierda para empezar nuevamente la actualización. A este momento en el que los cañones se regresan al principio del ciclo se le llama retrazo vertical.

Cuando se está desplegando una animación no se puede enviar la imagen al monitor en medio de una actualización, porque de esta forma puede combinarla con la imagen que está actualizando y producir un salto en la animación. Por esto se debe sincronizar la animación con el retrazo vertical. Se espera el momento en que el monitor esté regresando sus pistolas al inicio del ciclo y se le envía la nueva imagen, de esta forma cuando el monitor empiece a actualizar la imagen, lo hará con una imagen totalmente nueva. El monitor envía una señal al puerto 0x3da en el momento en que realiza el retrazo vertical. El ciclo correcto de animación queda de la siguiente forma:

```
for (i=0; i < NumeroDeCuadros; i++)
{
 // Prepara la imagen
 Limpia_el_buffer();
 Dibuja_el_cuadro(i);
```

```
 // Transfiere la imagen a la pantalla
 Espera_el_retraso_vertical();
 Intercambia_los_buffers();
}
```

Debido a que el sistema de doble "buffer" y el retrazo vertical son características del hardware, el sistema operativo es el encargado de administrar estas funciones.

1.1.3 Primitivas. Usando la función para dibujar píxeles que provee el sistema operativo, se pueden construir funciones para dibujar primitivas, tales como círculos, líneas, rectángulos, etc. El algoritmo más usado para dibujar líneas es el algoritmo de Bresenham. Primero, el algoritmo debe determinar la orientación de la línea recta y luego debe escoger los píxeles apropiados para ser rellenados, los cuales deben representar el menor error visual. A partir de la función que dibuje una línea recta, resulta muy fácil construir funciones para dibujar polígonos. Sin embargo, dibujar polígonos rellenos de forma rápida puede ser bastante complicado. Por eso, el algoritmo de relleno se dirige únicamente a polígonos convexos. Para rellenar los polígonos se trazan líneas horizontales (debido a que son más rápidas de trazar) desde un borde del polígono hasta el otro. Primero se determinan los bordes del polígono, luego se almacenan en una lista los píxeles que los conforman y por último se trazan las rectas horizontales entre los píxeles de la lista. Este proceso se llama recorrido de conversión (scan conversion) y el proceso de rellenar completamente un polígono se

llama rasterización. El algoritmo de rasterización se limita solo a triángulos ya que los demás polígonos se construyen a partir de estos. Mas adelante se explicará como proyectando triángulos en el espacio se pueden obtener modelos tridimensionales. El proceso consiste en proyectar primero cada uno de sus vértices y luego construir el polígono sobre los vértices proyectados como se explicó en este capítulo.

1.2. GRÁFICOS TRIDIMENSIONALES

1.2.1 Proyecciones. Debido a que actualmente no existen dispositivos capaces de desplegar gráficos tridimensionales reales (hologramas), estos se deben simular en una pantalla bidimensional. El efecto se logra mediante la proyección de un modelo tridimensional en un plano. Existen dos tipos de proyecciones: proyección paralela y proyección en perspectiva. Ambas se encarga de convertir un punto en el espacio a un punto en un plano, cada una lo hace de forma diferente. La proyección paralela es usada en aplicaciones CAD, en donde los objetos no se deforman a medida que se alejan, mantienen sus mismas dimensiones. Esta se obtiene de la siguiente forma:

$$f(x,y,z) \rightarrow f(x+z, y+z)$$

La proyección en perspectiva es usada para simular nuestro campo de visión, en donde los objetos se hacen más pequeños a medida que se alejan. Esta se obtiene de la siguiente forma:

$$f(x,y,z) \rightarrow f(x/z, y/z)$$

1.2.2 Transformaciones. A un modelo en el espacio se le puede modificar su posición, tamaño, orientación e incluso su propia forma. Luego de transformar cada uno de los vértices de un modelo se proyectan para mostrar su nueva posición en la pantalla. Para cambiar la posición de un objeto simplemente se suma el vector traslación a cada uno de sus vértices:

Nuevo Vértice = Vértice (X+Tx, Y+Ty, Z+Tz)

Para cambiar el tamaño de un objeto se escalan sus vértices por un vector de escalamiento:

Nuevo Vértice = Vértice (X*Sx, Y*Sy, Z*Sz)

Igualmente, para rotar un modelo, se rotan cada uno de sus vértices usando las fórmulas trigonométricas.

Rotación respecto al eje X:

Nuevo Vértice = Vértice (X, Y*cos(α) - Z*sen(α), Z*cos(α) + Y*sen(α))

Rotación respecto al eje Y:

Nuevo Vértice = Vértice (X*cos(α) + Z*sen(α), Y, Z*cos(α) - X*sen(α))

Rotación respecto al eje Z:

Nuevo Vértice = Vértice ($X*\cos(\alpha)$ - $Y*\text{sen}(\alpha)$, $Y*\cos(\alpha) + X*\text{sen}(\alpha)$, Z)

En realidad, lo que se hace no es modificar los vértices del modelo, porque de hacerlo, el error introducido que se acumularía con cada transformación terminaría deformándolo. El método correcto consiste en almacenar las transformaciones en variables independientes al modelo. De esta forma, las transformaciones se van acumulando y cuando se necesite transformar el modelo, el error introducido será mínimo porque el modelo original se transformaría una sola vez. Una manera óptima de almacenar transformaciones es usando matrices de transformación.

Las matrices de transformación son matrices de cuarto orden que encapsulan todas las transformaciones que puede sufrir un modelo en el espacio. Para mostrar el estado actual del modelo, cada uno de sus vértices es multiplicado por su matriz de transformación. La ventaja de usar matrices radica en que se pueden concatenar transformaciones mediante la multiplicación de estas. La submatriz formada por las tres primeras columnas y las tres primeras filas representa los vectores que definen la base canónica del modelo. La matriz identidad indica que no se ha efectuado ninguna transformación.

$$M = \begin{bmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ X_{41} & X_{42} & X_{43} & X_{44} \end{bmatrix}$$

El vector (X11, X21, X31, X41) representa el eje X.

El vector (X12, X22, X32, X42) representa el eje Y.

El vector (X13, X23, X33, X43) representa el eje Z.

El vector (X14, X24, X34, X44) representa la traslación. Como se puede notar, los vectores pertenecen a un sistema coordenado homogéneo de cuatro dimensiones (x, y, z, w), que luego son proyectados al espacio tridimensional (x/w, y/w, z/w) con el fin de mantener la matriz cuadrada e invertible. Para construir una matriz de transformación que traslade un modelo una posición (x,y,z), se reemplazan los valores en la cuarta columna. Al multiplicar la matriz T por la matriz actual del modelo se halla una nueva matriz, al multiplicar la nueva matriz por cada uno de los vértices del modelo, este es transformado a la posición deseada. De forma similar, para construir una matriz de transformación que escale un modelo se reemplazan los valores de la diagonal. Las matrices de rotación se construyen dependiendo del eje respecto al cual se quiera rotar.

1.2.2 Iluminación y materiales. Hasta ahora se ha explicado como dibujar figuras geométricas en el espacio, pero estas carecen de realismo y profundidad, las proyecciones aparecen como manchas sobre la pantalla. Así como los dibujantes sombrean sus dibujos para darles volumen, igualmente se deben sombrear los polígonos para imprimirles realismo. Para esto se tiene que simular el comportamiento de la luz. Simular todas las propiedades de la luz es bastante complicado debido a que la luz no viaja en línea recta y se esparce en todas las direcciones, pero para efectos de simplificación se puede asumir que viaja en forma rectilínea, por lo que se ahorran muchos cálculos. Para simular la iluminación, se ha establecido que los objetos reflejan la luz que no pueden absorber. Un objeto negro ha absorbido toda la luz, mientras que

uno blanco la ha rechazado por completo. Se asume que la luz tiene tres componentes: Componente ambiental, difusa y especular. Cada componente es representada por un color cuyas intensidades varían de 0 a 1. Los colores que emite la luz son rechazados o absorbidos por cada objeto de la escena dependiendo de las propiedades del material del objeto. La componente ambiental hace referencia a la luz del ambiente, representa aquella luz que se ha esparcido demasiado y de la cual es muy difícil determinar su fuente. La componente difusa representa la luz que viene de una sola dirección, por lo tanto, entre más perpendicular caiga sobre la superficie, más brillante será. La luz es reflejada de igual forma en todas las direcciones. La componente especular representa la luz que viene de una dirección particular y tiende a rebotar en otra dirección específica. Se puede notar con mayor intensidad en los materiales brillantes, por ejemplo en el vidrio. Pero no todos los objetos reaccionan de la misma forma ante una misma fuente de luz. Los materiales de los objetos reflejan las luces dependiendo de sus propiedades, algunos son opacos como la tierra, otros son brillantes como el aluminio. Para definirle las propiedades a los materiales, estos mantienen cinco componentes: ambiental, difusa, especular, brillante y emisiva. Además de las tres componentes que tienen las luces, se puede simular el hecho de que un objeto emita luz mediante la componente emisiva. Es decir, un material con una componente emisiva alta parecerá una fuente de luz. La componente brillante indicará que tan brillantes serán los reflejos causados por la luz especular. Las componentes de los materiales se combinan con las componentes de las luces que influyen sobre estos para definir el color final de los píxeles que representan al objeto. Así por ejemplo, si un material tiene una componente difusa roja, significa que el material absorberá toda la

luz verde y azul. Por lo tanto si se le aplica una luz verde o azul el objeto se verá negro. La luz no se esparce uniforme por toda una superficie, es más fuerte donde el ángulo de inclinación de la luz sea más recto. Igualmente, dependiendo de la superficie la luz rebotará en una dirección específica. Para poder simular este efecto es necesario conocer el vector normal a la superficie que represente la orientación del polígono. La normal de un triángulo se puede hallar fácilmente mediante el producto cruz de dos vectores formados por los vértices del triángulo. Para conocer el ángulo entre la normal y el vector de la luz se usa el producto punto entre dos vectores. Si el ángulo es 0 grados, se usa la máxima intensidad de la luz en ese punto. Un ángulo de 90 o más grados indica que el polígono está totalmente de espalda a la luz. Por lo que no refleja nada. El valor de las intensidades de cada vértice es calculado mediante la combinación de las propiedades del material y las propiedades de la luz. Se asume que el valor de una intensidad varía de 0 a 1. Las fórmulas para calcular la intensidad final de un vértice están dadas de la siguiente forma:

Componente Ambiental Final = Componente Ambiental de la Luz * Componente Ambiental del Material

La componente difusa depende del ángulo de inclinación del vector luz sobre la normal, existirá componente difusa siempre y cuando el ángulo sea mayor que cero:

Componente Difusa Final = Max (Vector Luz . Vector Normal , 0) * Componente Difusa de la Luz * Componente Difusa del Material.

La componente especular depende de la posición del espectador para poder simular el efecto del brillo, por lo tanto se necesita conocer el vector que sale del ojo del espectador hasta el vértice del polígono (V).

Componente Especular Final = Max (H . N, 0) * Componente Brillo del Material * Componente Especular de la Luz * Componente Especular del Material

El color final del vértice estará dado por:

Color Final = Componente Emisiva del Material + Componente Ambiental Final * Componente Difusa Final * Componente Especular Final

Luego, para calcular el color final de la luz a lo largo del polígono se interpola linealmente entre los valores de las intensidades calculadas en las normales de cada vértice. De esta forma se construye un potente sistema de iluminación, sin embargo, se puede mejorar partiendo de las siguientes consideraciones: Sobre un objeto no actúa una sola luz, actúan muchas, cada una de las cuales contribuye al color final de cada vértice. Muchas luces no tienen un ángulo de 360 grados de esparcimiento, por ejemplo, las linternas, los reflectores, los láser.

Para cubrir el primero de estos casos se establece que la contribución de cada luz depende de la distancia de la misma al objeto, para esto se define un factor de atenuación para cada luz, dada por:

$$Factor\ de\ Atenuación = 1 / (Kc + Kl * d + Kq * d^2)$$

En donde, Kc representa la constante de atenuación, Kl la atenuación lineal, Kq la atenuación cuadrática y d la distancia de la luz al vértice. Para luces con una distancia infinita el factor de atenuación es 1. Por ejemplo, la luz solar.

El efecto del reflector se logra asumiendo que este consta de una dirección, un ángulo de esparcimiento y una distribución de intensidad de la luz a lo largo de su cono, representada por un exponente de distribución.

El factor del reflector será igual a:

1 – Si el ángulo de esparcimiento es de 360 grados, es decir, no es un reflector.

0 – Si el vértice no está dentro del cono de iluminación.

Max (Vértice . Vector Dirección, 0) * Exponente de Distribución de Intensidad – Para cualquier otro caso.

Para determinar si un vértice está dentro del cono de iluminación, se evalúa:

Si (Max (Vértice . Vector Dirección, 0)) < Cos (Angulo de Esparcimiento / 2)) el vértice se encuentra fuera

Al final, la contribución de cada luz al color de un vértice estará dada por:

Contribución Final = (Factor de Atenuación * Factor del Reflector * Componente Ambiental Final * Componente Difusa Final * Componente Especular Final)

Entonces, el color final de un vértice estará dado:

Color Final = Componente Emisiva + Contribución Final de Cada Luz

1.2.3 Mapeado de texturas. Usando solamente luces y geometría sólida es muy difícil generar imágenes virtuales fotorealísticas, capaces de engañar por sí solas al ojo humano. Sin embargo, mediante un proceso sencillo llamado mapeado de texturas es posible envolver la geometría con imágenes bidimensionales permitiendo alcanzar gran realismo. Como se explicó anteriormente, una imagen es un arreglo de píxeles, cada píxel representa un color, lo que se hace es asignar píxeles a vértices, esto se hace mediante coordenadas bidimensionales, llamadas coordenadas de textura. Casi nunca se tendrá el mismo número de vértices que de píxeles, por lo que se interpola entre los valores de las coordenadas de textura. De esta forma se calcula el color que debe tener un vértice y se mezcla con el color calculado por el sistema de iluminación visto en el capítulo anterior. Las formas como se asignan las texturas a un polígono pueden ser muchas, se puede hacer que la imagen se estire para envolver a todo el polígono, o

por el contrario que se repita, que se mezcle con el color del vértice o que lo reemplace por completo. Cuando las imágenes se estiran o se encogen sufren una deformación, para prevenir esta anomalía se usan filtros para la magnificación (cuando se estiran) y minificación (cuando se encogen). El objetivo de los filtros es escoger el píxel apropiado para cada vértice asociado, de esta forma la imagen mantiene su consistencia.

1.2.4 Eliminación de superficies ocultas. Un reto común para todos los renderizadores era el de dibujar correctamente una imagen sin importar el orden en el que se especificaran los polígonos. Suponiendo que se quería dibujar una imagen comprendida por dos planos intersectados, si se especificaba primero el plano A y luego el B, cuando el plano B era renderizado terminaba reemplazando los píxeles de la renderización del plano A. Para corregir el problema se propuso la creación de un "buffer" adicional que mantuviera la información de profundidad asociada a cada pixel, a este se le llamó el "buffer" Z. Cuando se proyectan nuevos polígonos, se evalúa si los nuevos píxeles tienen un valor de profundidad mayor o menor al que se encuentra actualmente en el "buffer" Z. Solamente aquellos píxeles que se encuentren más cerca del espectador, sobrescriben a los que se están proyectados en la pantalla.

1.2.5 APIs de bajo nivel y alto rendimiento. Hasta hace algunos años para crear animaciones tridimensionales en tiempo real se necesitaban estaciones de trabajo

especiales basadas en Unix, que resultaban extremadamente costosas. La compañía líder en este área era Silicon Graphics (SGI), la cual había desarrollado una biblioteca gráfica llamada OpenGL, que permitía desplegar renderizaciones en tiempo real. Con el avance del hardware gráfico en los últimos años y la estandarización de OpenGL por parte de SGI, muchos fabricantes de tarjetas de video implementaron versiones por hardware de la biblioteca, lo cual hacía posible desarrollar aplicaciones tridimensionales en computadores personales de bajo costo. Para los desarrolladores de aplicaciones tridimensionales, OpenGL provee 250 llamadas, que son usadas para renderizar escenas, crear luces, aplicar texturas, dibujar líneas, puntos, polígonos, etc. Actualmente existen muchas implementaciones de código abierto de OpenGL (por ejemplo, Mesa3D para Linux), lo que ha hecho que el estándar sea adoptado por un mayor número de usuarios y que ahora el código sea independiente de la plataforma. Además de OpenGL existen otras APIs que aunque son ampliamente usadas, no la superan en popularidad, por ejemplo: DirectX de Microsoft, Glide de 3dfx o Quick3D de Apple, entre otras. El uso de OpenGL como interfaz de programación (API) trae principalmente tres beneficios para el programador: Lo libra de la tarea de implementar algoritmos para dibujar líneas, polígonos o para renderizar correctamente la imagen, tienen un rendimiento más óptimo debido a que el código es implementado en el hardware, es un estándar en la industria, por lo tanto el código de la aplicación es independiente de la plataforma y portable.

2. FUNDAMENTACIÓN TEÓRICA

2.1 EL MODELO TRIDIMENSIONAL

Un modelo está formado por una colección de objetos, cada objeto está formado por una colección de triángulos llamados caras. Cada cara está formada por tres vértices, un vector normal y un material. Cada vértice de la cara está formado por un color, una coordenada de textura y una posición relativa al origen del objeto. A los modelos que están formados a base de polígonos se les llama apropiadamente modelos poligonales, existen otros tipos de modelos, pero cualquier modelo no poligonal se puede convertir en un modelo poligonal. A continuación se explicará con más detalle cada una de las componentes del modelo poligonal, empezando desde las estructuras de datos más básicas.

2.1.1 Estructuras básicas. Las estructuras básicas son el vector y el color, a partir de estos se construyen las demás entidades que forman al modelo.

2.1.1.1 El Vector. Un vector representa una posición en el espacio, está compuesto por tres coordenadas. El valor de cada coordenada es un escalar o número de punto flotante. El tamaño en memoria de un vector es de 12 bytes.

2.1.1.2 El Color. Un color almacena las intensidades de los canales que lo componen. Como se explicó en el numeral 1.1.1, las intensidades varían de 0 a 255, por lo que sería suficiente un byte para almacenar cada una. Sin embargo, se usan valores escalares debido a que el sistema de iluminación necesita los colores en un rango de 0.00 a 1.00. Para evitar que el sistema de iluminación tenga que convertir los colores a este rango en tiempo de ejecución, se almacenan como escalares.

2.1.1.3 El material. El material de la cara está formado por una textura y sus componentes: ambiental, difusa, especular, brillante y emisiva. La textura es un mapa de bits que se aplica sobre la superficie del material. Cada componente es representada por un color, excepto la componente brillante que es representada por un valor escalar. La cantidad de bytes que se necesitan para almacenar un material en memoria está dada por: cuatro colores, un valor escalar y un imagen de tamaño variable, para un total mínimo de 68 bytes. La información del material es usada por el renderizador para simular la reacción de una cara ante una fuente de luz.

2.1.1.4 La cara. Cada cara está formada por: un material, tres vértices y un vector normal calculado a partir de la posición de sus vértices. Para ahorrar cálculos en tiempo de ejecución, el vector normal es precalculado y almacenado en la cara. Cada uno de los vértices contiene: el vector posición, el color y las coordenadas de textura en ese punto. Con esta información el renderizador realiza las siguientes operaciones:
Los tres vectores de posición son transformados y luego proyectados en la pantalla. Con los tres puntos proyectados se construye un triángulo. El color a lo largo del triángulo es calculado mediante una interpolación lineal entre los colores de sus vértices. Usando las coordenadas de textura se selecciona la región de la textura correspondiente a la cara. Usando la normal se calcula la intensidad de la luz en la cara. Usando el material, la intensidad de la luz, los colores y la textura se calcula el color final de los píxeles que forman la cara.

2.1.2 Optimización. Bajo el esquema anterior, un objeto de 10000 caras necesitaría al menos 1.79 Mb para almacenarse en memoria. Por lo general, un mundo virtual está compuesto de cientos de objetos aún más complejos. Por lo tanto, una optimización de memoria se hace necesaria. Debido a que un vértice es compartido por varias caras, no es necesario almacenar varias veces la misma información, además porque esto puede crear inconsistencias. La solución es manejar un esquema de índices. En un contenedor se almacenan los valores de cada vértice y se les asigna un índice, en las caras se almacena el valor de los índices en lugar del valor de los vértices. El siguiente ejemplo muestra como se almacenan los vectores posición de una pirámide bajo los

dos esquemas. Bajo el primer esquema, cada una de las caras almacena nueve escalares que corresponden a sus tres vectores posición, usando así 648 bytes.

Bajo el segundo esquema, los 6 vértices de la pirámide son almacenados en un contenedor, las caras almacenan únicamente los índices de los vectores posición en el contenedor. El contenedor con 36 valores escalares y las caras con 24 valores enteros requieren 240 bytes de memoria. Para efectos de sencillez, en el ejemplo se tomaron únicamente los valores de los vectores posición, pero en realidad existe un contenedor para cada elemento de la cara: normales, materiales, texturas, colores, coordenadas de textura y vectores posición.

2.2. LA CÁMARA

Con el fin de facilitar la navegación a través del mundo virtual, se crean cámaras. Las cámaras, al igual que las demás entidades del espacio, mantienen una posición y una orientación relativas a un sistema de coordenadas. Además, estas poseen lentes que permiten modificar el campo de visión. De esta forma, cuando el usuario quiera recorrer el espacio, simplemente modifica el estado de una cámara y a través de esta percibe el mundo virtual. Hasta ahora se ha expuesto que el proceso que se sigue para renderizar un modelo es el siguiente:

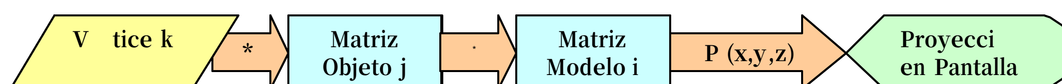

Figura 1. Flujo de transformación

Los vértices de cada objeto del modelo son transformados en el siguiente orden: Primero, son multiplicados por la matriz de transformación del objeto al que pertenecen. Luego, son multiplicados por la matriz de transformación del modelo. El vector resultante es proyectado para revelar la posición de un píxel en la pantalla. Para simular una cámara se debe tener en cuenta que los objetos son transformados de forma inversa a la cámara. Si la cámara se mueve hacia delante, en la pantalla los objetos se ven movidos hacia atrás; si la cámara se mueve hacia arriba, los objetos se ven movidos hacia abajo. Por lo que se deduce que los objetos, antes de ser proyectados, son transformados por la inversa de la matriz de transformación de la cámara. Quedando el proceso de renderización de la siguiente forma:

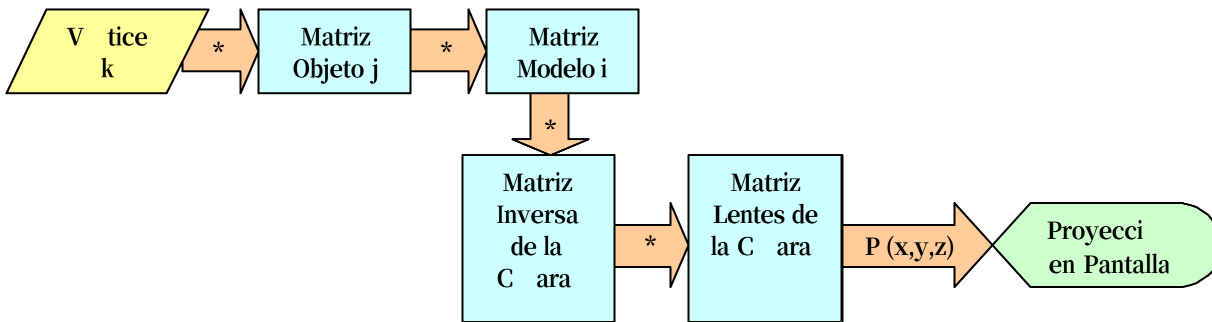

Figura 2. Flujo de transformación con cámara

La matriz que representa los lentes de la cámara se usa para simular el campo de visión, el cual se puede construir a partir de los planos que lo limitan. Para convertir seis planos que definen un volumen de visión en una matriz de transformación o viceversa se tiene en cuenta lo siguiente: En las columnas de una matriz de transformación se encuentra definida la base canónica o ejes coordenados de un espacio. En las filas de la matriz se encuentran definidos los estados de sus

coordenadas, a partir de los cuales se pueden hallar los vectores normales de los planos que definen el volumen del campo de visión. La primera fila de la matriz de la cámara representa el estado de las X, y por lo tanto los planos laterales. La segunda fila representa el estado de las Y, y por lo tanto los planos superior e inferior, y la tercera fila representa el estado de las Z, y por lo tanto los planos cercano y lejano. Un plano se puede representar mediante un vector normal y su distancia al origen (Nx, Ny, Nz, d). Los planos se pueden extraer de la matriz de transformación de la siguiente forma:

Plano derecho = Fila4 - Fila1

Plano izquierdo = Fila4 + Fila1

Plano superior = Fila4 - Fila2

Plano inferior = Fila4 + Fila2

Plano lejano = Fila4 - Fila3

Plano cercano = Fila4 + Fila3

Usando los planos se puede saber fácilmente si un punto se encuentra dentro del campo de visión, esto se logra reemplazando el punto en la fórmula del plano:

$$ax + by + cz + d = 0$$

Si el resultado es igual a cero, el punto se encuentra sobre el plano.

Si el resultado es mayor que cero, se encuentra delante del plano.

Si el resultado es menor que cero el punto se encuentra detrás del plano.

2.3. ROTACIONES EN EL ESPACIO

Para representar la orientación de un objeto en un plano se usa simplemente un ángulo, sin embargo, manipular las rotaciones en el espacio resulta un poco más complicado. A continuación se exponen varias formas de representar dichas orientaciones:

Mediante los ángulos de Euler: Consiste en usar tres ángulos para representar la orientación respecto a cada uno de los ejes. Los ángulos de Euler son susceptibles a un problema llamado “Cerradura del Cardán”, el cual sucede porque la orientación final del objeto cambia dependiendo del orden de las rotaciones.

Mediante un ángulo y un eje de rotación: Se usa un vector que represente un eje y se mantiene un ángulo de rotación respecto a ese eje. De esta forma se representa consistentemente una orientación, pero su manipulación es complicada. La orientación se puede convertir en una matriz de rotación si se tiene en cuenta que el vector que representa el eje de rotación es el vector propio de la matriz.

Mediante una matriz: Se usa una matriz 3x3, en donde cada columna representa un eje del objeto. Se tiene que mantener la matriz de forma ortogonal para asegurar que los vectores que representan los ejes del objeto sean perpendiculares. Su manipulación también es complicada.

Aunque todos los métodos anteriores sirven para representar correctamente orientaciones, estos carecen de flexibilidad y no permiten una fácil manipulación. Por ejemplo, interpolar entre dos orientaciones puede ser una tarea complicada bajo cualquier método. Una forma compacta de manejar y representar orientaciones es mediante cuaternios.

2.3.1 Cuaternios. A mediados del siglo XIX, el matemático irlandés W.R. Hamilton, desarrolló el álgebra de cuaternios, mediante la cual se podían representar rotaciones y orientaciones en tres dimensiones. Hamilton expuso a los cuaternios como una extensión de los números complejos en cuatro dimensiones. Un número complejo tiene la forma $z = a+bi$, donde $i^2=-1$. También se pueden representar de forma polar $z = (r,\theta)$ o de forma exponencial $z = r \times e^{i\theta}$. Cuando se multiplican dos números complejos, los módulos se multiplican y los ángulos se suman, por lo tanto mediante estos se pueden representar rotaciones en el plano. Para manejar rotaciones en el espacio se usan cuaternios de la forma:

$$Q = W + Xi + Yj + Zk$$

donde i2=j2=k2=-1, ij=-ji=k, jk=-kj=i, ki=-ik=j

Los cuaternios también se representan de la forma: $Q = < W, V >$ donde W es un número real y V es un vector tridimensional.

Las operaciones básicas entre cuaternios son las siguientes:

Adición: Q1+Q2 = < W1 + W2, V1 + V2 >

Multiplicación: Q1Q2 = < W1W2 – V1.V2, V1xV2 + W1V2 + W2V1 >

Conjugada: Qc = < W, -V >

Norma: N(Q) = W2 + X2 + Y2 + Z2

Inversa: Q-1 = Qc / N(Q)

Se dice que un cuaternio es unitario si N(Q) = 1, por lo tanto Q-1 = Qc.

Las orientaciones tienen que ser representadas por cuaternios unitarios.

Una orientación representada mediante un eje de rotación y un ángulo se puede convertir en un cuaternio de la siguiente forma:

$$Q = < \cos(\theta/2), \sin(\theta/2)V >$$

donde θ es el ángulo y V es el eje de rotación.

Un cuaternio se puede convertir en una matriz de rotación de la siguiente forma:

$$R_m = \begin{bmatrix} 1-2y^2-2x^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}$$

2.3.2. Interpolación entre rotaciones. Una forma de almacenar la animación de un modelo es guardando sus diferentes posiciones y orientaciones, la animación es reconstruida mediante interpolaciones entre estos valores. A este tipo de animación se le llama interpolación entre cuadros de animación (key frame interpolation). Una extensión de este tipo de animación es la cinemática inversa (inverse kinematics), la cual calcula cuadros de animación apropiados mediante restricciones en sus rotaciones. Por ejemplo, con solo especificar la posición de una extremidad del objeto, el sistema deduce la posición del resto del objeto debido a sus movimientos permitidos. Ambos métodos de animación necesitan aplicar métodos de interpolación apropiados. El método más simple consiste en usar la interpolación lineal con cada valor de posición y orientación del modelo:

Valor Interpolado = Valor[i] * (1.0-factor) + Valor[i+1] * factor

donde Valor[i] es el valor actual, Valor[i+1] es el valor siguiente y factor es la fracción de tiempo entre los dos valores.

Se pueden lograr caminos de animación más complejos si se interpola entre los cuaternios. La forma más apropiada es la interpolación lineal esférica (SLERP):

$$\mathrm{SLERP}(p, q, t) = \frac{p \sin((1-t)\theta) + q \sin(t\theta)}{\sin(\theta)}$$

2.4 DETERMINACIÓN DE LA VISIBILIDAD

La potencia requerida para procesar algunos mundos virtuales en tiempo real puede llegar exceder la capacidad de los computadores modernos, incluso la de aquellos con aceleración gráfica. Para mantener una animación fluida, independiente de la complejidad del mundo virtual, es necesario implementar algoritmos para la determinación de la visibilidad. Estos se encargan de seleccionar rápidamente los polígonos que son visibles por el espectador y renderizar únicamente aquellos que contribuyen con la imagen final. El método más simple consiste en descartar aquellos polígonos que se encuentran fuera del campo de visión. Esto se logra evaluando los polígonos respecto a los planos que forman el volumen de visión. Sin embargo, aún dentro del campo de visión se encuentran muchos polígonos que no contribuyen con la imagen final. Estos polígonos, aunque sus píxeles antes de ser proyectados son descartados por el algoritmo del “buffer” Z, para este entonces ya han sido transformados, y por lo tanto han consumido gran tiempo del procesador.

2.4.1 Herramientas

2.4.1.1 Subdivisión Jerárquica. Un esquema implementado por la mayoría de los algoritmos de determinación de visibilidad es la subdivisión jerárquica del espacio, debido a que permite descartar rápidamente grandes partes de la escena. Si una parte alta de la jerarquía se clasifica como invisible, su descendencia también lo es y por lo

tanto no será procesada. Esta técnica incluye métodos como: BSP ([Bsp98]), octrees, quadtrees, etc.

2.4.1.2 Conjuntos potencialmente visibles. Este término denota un conjunto de polígonos o celdas de una escena que son potencialmente visibles para un espectador. El método consiste en dividir un mundo virtual en celdas, la información de visibilidad de celda a celda es precalculada y adjuntada a la celda. En tiempo de ejecución, esta información puede ser consultada para saber que conjunto de polígonos es potencialmente visible desde la celda actual. Este método se expuso por primera vez en [Air90].

2.4.1.3 Coherencia espacial y temporal. Los algoritmos para la determinación de la visibilidad toman ventaja de estas propiedades para ser más efectivos. El término coherencia espacial se refiere tanto a coherencia espacial de objetos como a coherencia espacial de imágenes. La coherencia espacial de objetos se refiere al hecho de que polígonos cercanos mantienen una relación de visibilidad. Los algoritmos que más explotan esta propiedad son los de subdivisión jerárquica, ya que se aseguran de que los objetos cercanos sean evaluados primeros a niveles inferiores. De forma similar, la coherencia espacial de imágenes se refiere a la relación existente a nivel de píxeles. La coherencia temporal puede ser explotada, por ejemplo, cuando se

almacena en memoria temporal información de visibilidad referente a un cuadro de animación y se usa para predecir la visibilidad del próximo cuadro.

2.4.2 Algoritmos

2.4.2.1 Descarte de caras traseras (Back face culling). El descarte de caras traseras sirve para liberarse de mucha de la geometría innecesaria que se encuentra dentro del campo de visión. Cuando los objetos del mundo virtual son convexos, se pueden descartar los polígonos traseros, ya que estos no son visibles ante el espectador.
Si la normal de un polígono mantiene un ángulo menor de 90° respecto a la normal del plano trasero del volumen de visión, entonces ese polígono no es visible y por lo tanto puede ser descartado. Esto se puede determinar mediante el producto punto entre los dos vectores normales.

2.4.2.2 Subdivisión espacial mediante árboles de ocho hijos. Un árbol de ocho hijos (octree) es una estructura de datos jerárquica que representa el conjunto de polígonos de un volumen. El volumen es dividido en ocho partes iguales y repartido entre los subnodos del árbol. El árbol es construido subdividiendo recursivamente el volumen y almacenando los polígonos en sus nodos terminales. Un nodo es redefinido mientras se cumplan dos condiciones: Primero, que el número de polígonos almacenados en el nodo sea mayor a un número máximo establecido. Segundo, que el volumen que

representa cada nodo no sea menor a un tamaño mínimo establecido. Una vez construido el árbol resulta fácil determinar los polígonos que se deben renderizar en un instante dado. Para esto se evalúa si el nodo que representa el volumen está dentro del campo de visión. Si lo está, se evalúan recursivamente sus subnodos, en caso contrario se cancela el recorrido del árbol por esa rama. Cuando se alcance un nodo terminal se renderizan los polígonos almacenados en el nodo. De esta forma, solo se renderizan aquellos polígonos dentro del campo de visión.

2.4.2.3 Descarte de polígonos ocultos. Mediante este método se pretende encontrar aquellos polígonos que aún estando dentro del campo de visión y orientados hacia el espectador son invisibles debido a que están tapados por otros polígonos más cercanos. La idea es identificar polígonos candidatos para ocultar otros polígonos y comprobar en tiempo de ejecución si los demás polígonos se encuentran ocultos detrás de estos. Por lo general, los candidatos son polígonos grandes. Los métodos más populares son los propuestos en [Gre93] y [Zha97].

2.4.2.4 Niveles de detalle. Cuando un modelo se encuentra visible frente al espectador, pero a la vez se está muy lejos, su proyección en la pantalla no es muy detallada debido a que todo el modelo debe ser proyectado en pocos píxeles.
Esta característica se puede aprovechar para aligerar la renderización de los modelos distantes al espectador. El método de niveles de detalle consiste en calcular varias

resoluciones de un modelo e ir renderizando las resoluciones más bajas a medida que éste se aleje del espectador. El cálculo puede ser en tiempo real o mediante un preprocesamiento. El trabajo "Mallas progresivas" presentado por [Hop96] presenta un método eficiente para lograr diferentes niveles de detalle en tiempo real. Para evitar el salto visual que se produce cuando se intercambian dos resoluciones de un modelo se aplica un método llamado "geomorphing", el cual interpola entre las posiciones de los vértices de ambas resoluciones o entre los valores alpha de ambos modelos para ocultar el intercambio.

2.5 TERRENOS

Los sistemas geográficos de información (GIS) proveen datos sobre la superficie terrestre que pueden ser visualizados en tres dimensiones. La exploración de estos mapas en un ambiente virtual ayuda a muchas personas a estudiar aspectos geográficos del planeta, sus aplicaciones varían en áreas que van desde la ecología hasta la milicia. A continuación se explican los métodos usados para representar, renderizar y recorrer terrenos en tiempo real.

2.5.1 Representación y renderización. Los valores de las elevaciones son almacenados en un arreglo bidimensional llamado mapa de alturas. Usualmente se usan imágenes (bitmaps) en escala de gris para representarlos. Cada elemento del

arreglo representa la elevación de un punto tomada a un intervalo fijo en el plano XZ. Usando estos valores se construye una malla de triángulos, en donde el valor XZ de los vértices es fijo y el valor Y es la altura en ese punto.

2.5.2 Generación de terrenos aleatorios. En ciertas aplicaciones, pruebas o simulaciones puede ser conveniente usar terrenos artificiales. Para esto es necesario generar valores aleatorios para cada una de las elevaciones y a la vez mantener una relación entre estas para poder formar valles, montañas y otros accidentes naturales. La matemática que puede modelar este tipo de comportamientos desordenados, pero a la vez relacionados entre sí, es la matemática del caos. Los modelos geométricos que representan fenómenos caóticos son llamados fractales. Una forma de generar fractales que describan terrenos convincentes es mediante el método del desplazamiento del punto medio ([Mar96]). La versión bidimensional del método divide iterativamente una línea en segmentos hasta obtener la silueta de una montaña. Para cada segmento de línea se calcula una altura en su centro. Para hallar esta altura, se promedia la altura de los extremos del segmento y se le suma un valor aleatorio. El rango del valor aleatorio se va reduciendo con cada iteración. Si el rango se reduce lentamente, el terreno será más suave, si se reduce bruscamente, será más rugoso. Para extender el algoritmo a tres dimensiones se deben calcular los valores aleatorios y los promedios de forma alterna entre los valores de las esquinas y los centros de los bordes de la malla. Para que este método funcione, la malla debe tener un tamaño de $(2n+1) \times (2n+1)$.

2.5.3. Recorrido de grandes terrenos en tiempo real. Para los terrenos cuyos tamaños exceden la capacidad de procesamiento de la máquina, se debe aplicar un algoritmo de determinación de visibilidad y de esta forma permitir su recorrido en tiempo real.

Las mallas poseen características especiales que se pueden aprovechar para reducir el número de triángulos de un terreno mientras se maximiza su calidad visual. Los trabajos más influyentes sobre multiresolución de terrenos son los presentados en [Lin96], [Duc97] y [Rot98]. Cuando un espectador recorre el terreno, en un momento dado ciertas regiones contribuyen más que otras con la imagen final, y por lo tanto deben ser renderizadas con mayor detalle que el resto. Para controlar dinámicamente los niveles de detalle del terreno, se almacenan los datos de las elevaciones en un árbol de cuatro hijos (quadtree). Para dar más detalle a una región se desciende por la rama correspondiente para acceder a datos de mayor resolución, con los cuales se construyen triángulos más pequeños. Las regiones son redefinidas dependiendo de la evaluación de una medida de error, la cual representa el error visual del terreno una vez proyectado en la pantalla. Por lo general, las regiones que obtienen menor error visual son aquellas que: Se encuentran más cerca del espectador y se encuentran lejos, pero debido a su altura son claramente visibles. El primer criterio hace referencia a que la resolución de una región decrece a medida que su distancia al espectador se incrementa. El segundo criterio buscar conservar las regiones más altas y rugosas que se observan en la distancia. La medida de error se halla de la siguiente forma:

$$\text{Medida de error} = \text{rugosidad} / \text{distancia}$$

La rugosidad de una región es una medida para que representa los cambios en su altura, se calcula:

$$\text{Rugosidad} = \max(|dh_i|)$$

donde los dh_i son las diferencias de altura calculadas entre los vértices de la región.
Se debe tener especial cuidado cuando se subdividen las regiones para evitar “uniones T”. Las “uniones T” ocurren cuando dos triángulos adjuntos dejan de compartir un vértice, el efecto que causan son huecos en la malla.

2.6 SIMULACIÓN FÍSICA

Las leyes físicas que gobiernan el universo se pueden simular para crear una atmósfera de realismo en el mundo virtual, para esto se deben conocer las ecuaciones que describen estos fenómenos. En la mayoría de los fenómenos físicos, la tasa de cambio de una cantidad depende del valor de la cantidad misma, por ejemplo: La tasa de cambio de la temperatura de un objeto depende de la temperatura actual del objeto. Es por eso que estos fenómenos se describen mediante ecuaciones diferenciales. Una ecuación diferencial es aquella en donde las derivadas de la variable dependiente, aparecen en la ecuación junto a la variable dependiente e independiente.

2.6.1 Solución numérica de ecuaciones diferenciales. La solución simbólica de ecuaciones diferenciales es muy compleja para ser resuelta mediante un algoritmo, sin embargo, usando iteraciones, la solución numérica obtiene un valor aproximado de forma sencilla. Lo que se busca con la solución numérica es hallar el valor de una variable en un estado n+1 cuando se tiene el valor en el estado n y la función que describe el comportamiento de la variable.

$$\text{Hallar } \chi(n+1) \text{ a partir de } \chi(n) \text{ y } f(x,n)$$

Los métodos más usados son los de Runge-Kutta y la extrapolación de Richardson o Bulirsch-Stoer. El método de Runge-Kutta propaga la solución sobre un intervalo basándose en las series de Taylor. Debido a su solidez siempre tiene éxito, y cuando el cálculo de la función es rápido, este método es el más eficiente. Para problemas cuya eficiencia computacional es importante, se usa la rutina de Bulirsch-Stoer (ver [Pre88]).

2.6.1.1 Método de Runge-Kutta. La forma pagana de resolver ecuaciones diferenciales es mediante el método de Euler, el cual es una forma primitiva del Runge-Kutta. El problema con usar este método es que introduce mucho error y causa inestabilidad numérica.

$$\chi(t0 + \Delta t) = \chi(t0) + \Delta t . f(x,t0) \quad \text{(Euler)}$$

Tanto el método de Euler como el Runge-Kutta se basan en las series de Taylor para hallar la aproximación numérica:

$$\chi(t0 + \Delta t) = \chi(t0) + \Delta t.f(x,t0) + ((\Delta t)2/2!).f'(x,t0) + ((\Delta t)3/3!).f''(x,t0) + \ldots$$

$$\ldots + ((\Delta t)n/n!).(\partial nx/\partial tn) \quad \text{(Taylor)}$$

Como se puede notar, el método de Euler trunca las series llevando a un error de O(h2) en la aproximación. El Runge-Kutta extiende un poco más las series hasta conseguir un error de O(h5):

$$\chi(t0 + \Delta t) = \chi(t0) + \Delta t.f(x,t0) + ((\Delta t)2/2!).f'(x,t0) + ((\Delta t)3/3!).f''(x,t0) + ((\Delta t)4/4!).f'''(x,t0) +$$

$$O(h5) \quad \text{(Runge-Kutta)}$$

El algoritmo del método de Runge-Kutta de orden 4 es el siguiente:

```
k1 = dt * f(y, t)
k2 = dt * f(y + (k1/2.0), t + (dt/2.0))
k3 = dt * f(y + (k2/2.0), t + (dt/2.0))
k4 = dt * f(y + k3, t + dt)
y = y + (k1/6) + (k2/3) + (k3/3) + (k4/6)
```

Una versión mejorada del algoritmo predice dinámicamente un dt óptimo. La idea es hallar un dt que sea lo más grande posible para ahorrar cálculos, pero no suficientemente grande como para producir inestabilidad numérica debido al error.

El método adaptivo del Runge-Kutta calcula $X(t0 + \Delta t)$ de dos formas: La primera mediante un paso desde t0 hasta $t0+\Delta t$, y la segunda usando dos pasos desde t0 hasta $t0+(\Delta t/2)$ y desde $t0+(\Delta t/2)$ hasta Δt. Por último se establece una medida de error entre las dos soluciones, dada por:

$$e = | Xa - Xb |$$

El dt óptimo se halla mediante: dt = dt * (Máximo error permitido / e)

2.6.2 Dinámica de partículas. Las partículas son objetos que poseen masa, posición y velocidad, pero carecen de extensión espacial. Debido a su sencillez se pueden adaptar para exhibir comportamientos interesantes, desde gases y fenómenos naturales como la lluvia, hasta gelatinas y telas cuando se interconectan mediante resortes. El campo de la física que estudia el movimiento de los cuerpos se llama cinética y el que estudia la relación entre las fuerzas y el movimiento se llama dinámica. A partir de estos, se deducen las tres ecuaciones que describen el movimiento de las partículas:

(1) La velocidad es el cambio de la posición respecto al tiempo. $v=dx/dt$

(2) La aceleración es el cambio de velocidad respecto al tiempo. $a=dv/dt$

(3) La aceleración es igual a la sumatoria de fuerzas que actúan sobre la partícula entre su masa. $a = \sum Fi / m$ (solo cuando la masa es constante)

Por lo tanto, para hallar la posición de una partícula en un instante dado, se debe resolver:

$$a = \sum Fi / m$$

$$v(t) = \int a.dt, \quad C = v(0)$$

$$x(t) = \int v(t).dt, \quad C = x(0)$$

La constante C resultante de la integración representa el estado inicial de las variables. Durante cada ciclo de simulación se determinan las fuerzas que actúan sobre las partículas, a partir de las cuales se hallan las aceleraciones y mediante la integración numérica se obtienen las nuevas posiciones. Entre las fuerzas que alteran el movimiento de las partículas están los resortes, la gravedad, la colisión con otros objetos, la resistencia viscosa y cualquier otra restricción que se quiera aplicar. La resistencia viscosa es una fuerza mínima, que tiende a reducir la velocidad, se usa para corregir problemas de inestabilidad numérica.

2.6.3 Dinámica de cuerpos rígidos. Los cuerpos rígidos son objetos que poseen extensión espacial, tienen una masa distribuida por todo su volumen y son

indeformables. A continuación se explican los conceptos necesarios para simular la dinámica de estos. Se puede ver un cuerpo rígido como una colección infinita de partículas fuertemente interconectadas. Se le llama centro de masa de un cuerpo, al punto de equilibrio o centro geométrico definido por el vector:

$$CM = (\sum mi . ri) / M$$

donde M es la masa total, mi es la masa
de la partícula i y ri su vector posición.

Se puede demostrar que el momento lineal (masa*velocidad) total de un cuerpo es igual a la masa total por la velocidad en su centro de masa. Por lo tanto, para simular el comportamiento de todas las partículas de un cuerpo, basta con simular el comportamiento de su centro de masa. Debido a su extensión espacial, los cuerpos rígidos poseen una orientación, que al igual que la posición, cambia con respecto al tiempo. Por lo que también mantienen una velocidad y una aceleración angular.
De forma similar:

$$d2\Omega/dt2 = d\omega/dt = \alpha$$

La orientación de un cuerpo rígido, se puede calcular a partir su aceleración angular.
La velocidad lineal de una partícula B que se encuentra en un cuerpo en movimiento es igual a:

$$V B = V O + \omega . r\perp OB$$

donde V_O es la velocidad lineal del cuerpo, ω su velocidad angular, y $r\perp_{OB}$ es el vector perpendicular al vector posición de B respecto al centro de masa del cuerpo.

La fuerza que modifica la aceleración angular se llama torque y se genera cuando se aplica una fuerza en cualquier punto diferente al centro de masa. El torque ocasionado en un punto A por aplicar la fuerza en un punto B está dado por:

$$\tau_{AB} = r\perp_{AB} \cdot F_B$$

donde $r\perp_{AB}$ es el vector perpendicular al vector AB y F_B es la fuerza aplicada en B.

Así como poseen un momento lineal, las partículas también poseen un momento angular respecto a otra partícula dado por:

$$L_{AB} = r\perp_{AB} \cdot P_B$$

donde P_B es el momento lineal de la partícula B. El momento de inercia de una partícula es una medida que indica que tan díficil es rotar el cuerpo alrededor de la partícula, se calcula como:

$$I_A = \sum m_i \cdot (r_{Ai})^2$$

donde I_A es el momento de inercia de todo el cuerpo respecto a la partícula A, m_i es la masa de la partícula i y r_{Ai} es el vector desde A hasta la partícula i. Para hallar la

aceleración angular se divide el torque total entre el momento de inercia en el centro de masa. El algoritmo para simular la dinámica de un cuerpo rígido realiza los siguientes pasos:

1. Calcula el centro de masa y el momento de inercia en el centro de masa.
2. Establece los valores iniciales para la posición, orientación, velocidad lineal y velocidad angular del cuerpo.
3. Halla todas las fuerzas que actúan sobre el cuerpo, incluyendo sus puntos de aplicación.
4. Suma todas las fuerzas y las divide entre la masa total para hallar la aceleración lineal del centro de masa.
5. Con cada fuerza calcula el torque total en el centro de masa.
6. Divide el torque total entre el momento de inercia en el centro de masa para hallar la aceleración angular.
7. Usando la rutina de Runge-Kutta integra numéricamente la aceleración lineal y la aceleración angular para hallar la velocidad lineal y la velocidad angular.
8. Integra nuevamente para hallar la posición y la orientación del cuerpo, a partir de sus velocidades lineal y angular.
9. Dibuja los objetos en la nueva posición y orientación, y regresa al paso 3.

2.6.3.1. Resolución de colisiones. Cuando dos cuerpos rígidos entran en contacto, pueden pasar dos cosas: Si los cuerpos se acercan entre sí con cierta velocidad,

colisionan y se repelen alejándose uno del otro; Si los cuerpos se acercan con una velocidad insignificante, se dice que entran en contacto de reposo. Para el primer caso, se debe calcular una fuerza de repulsión llamada impulso, que cambia la dirección y magnitud de las velocidades cuando ocurre el contacto. Para el segundo caso, se debe calcular una fuerza que restrinja la interpenetración de los cuerpos.

2.6.3.1.1 Contacto de colisión. Un sistema de detección de colisiones le provee al sistema de resolución de colisiones la siguiente información sobre un contacto: El tiempo exacto de la colisión, los objetos que participan en la colisión, el punto de colisión, la normal de colisión. Geométricamente pueden ocurrir varios tipos de contacto, los únicos casos que se consideran para lograr una respuesta adecuada a la colisión son los del tipo filo-filo y vértice-cara. En los demás casos es muy difícil determinar el vector normal de colisión. Cuando el contacto es de tipo vértice-cara, la normal de colisión es la normal de la cara. Cuando el contacto es de tipo filo-filo la normal de colisión es el vector perpendicular a los vectores que representan el filo de cada cara. Una vez se obtienen los datos de la colisión, se aplica un impulso a cada cuerpo. Existen muchas formas de calcular la dirección y la magnitud del impulso dependiendo del modelo de colisión que se use. A continuación se expone el impulso propuesto por Newton llamado “Ley de la restitución para colisiones instantáneas sin fricción”, en este modelo se introduce una nueva cantidad llamada coeficiente de restitución simbolizado por la letra e.

$$j = \frac{-(1+e)\mathrm{v}_1^{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}\left(\frac{1}{M_A} + \frac{1}{M_B}\right) + \left[\left(\mathbf{I}_A^{-1}(\mathbf{r}_{AP} \times \mathbf{n})\right) \times \mathbf{r}_{AP} + \left(\mathbf{I}_B^{-1}(\mathbf{r}_{BP} \times \mathbf{n})\right) \times \mathbf{r}_{BP}\right] \cdot \mathbf{n}}$$

donde: e es el coeficiente de restitución, n es la normal de colisión, V^{AB} es la velocidad del cuerpo A respecto al cuerpo B, M es la masa de los cuerpos, I es el momento de inercia, r^{AP} es la distancia del centro de masa de A hasta el punto de colisión.

El coeficiente de restitución representa cuanta energía se disipa durante la colisión y puede variar desde e=0 (colisión plástica) hasta e=1 (colisión elástica). Para comportamientos más realistas, se puede implementar la fuerza de fricción, la cual no está presente en este modelo. El impulso es un cambio de momento, y se aplica a la velocidad de los cuerpos de la siguiente forma:

$$\mathbf{v}_2^{\mathrm{A}} = \mathbf{v}_1^{\mathrm{A}} + \frac{j}{M^{\mathrm{A}}} \mathbf{n}$$

$$\mathbf{v}_2^{\mathrm{B}} = \mathbf{v}_1^{\mathrm{B}} - \frac{j}{M^{\mathrm{B}}} \mathbf{n}$$

donde V1 y V2 son las velocidades antes y después de la colisión respectivamente, j es la magnitud del impulso, M es la masa de los cuerpos y n la normal de colisión.

De forma similar, el cambio de momento también se aplica a las velocidades angulares:

$$\omega_2^A = \omega_1^A + \frac{\mathbf{r}_\perp^{AP} \cdot j\mathbf{n}}{I^A}$$

donde W1 y W2 son las velocidades angulares antes y después del impulso respectivamente, r⊥AP es el vector perpendicular al vector formado entre A y el punto de colisión, I A es el momento de inercia, j es la magnitud del impulso y n es la normal de colisión.

2.6.3.1.2 Contacto de reposo. Cuando ocurre un contacto de reposo, se debe calcular una fuerza por cada punto de contacto que cumpla con tres condiciones: La fuerza debe ser suficientemente grande para prevenir la interpenetración, la fuerza debe ser repulsiva, nunca puede tender a pegar dos cuerpos, la fuerza debe hacerse cero cuando los cuerpos se separen. Para establecer la primera restricción se parte de que la distancia de cada punto de contacto entre dos cuerpos debe ser mayor o igual a cero para un tiempo t:

$$d(t) = n(t) . (PA(t) – PB(t))$$

donde n es la normal de colisión, PA y PB son los puntos de contacto en los cuerpos A y B respectivamente. Para hallar la aceleración que deben mantener los cuerpos se deriva dos veces la ecuación y se obtiene:

$$d''(t) = n(t) \cdot (PA''(t) - PB''(t)) + 2n'(t) \cdot (PA'(t) - PB'(t))$$

La primera restricción que se debe mantener es: $d''(t) >= 0$. La segunda restricción hace referencia a que la fuerza debe ser positiva: $f >= 0$. La tercera restricción dice que la fuerza debe hacerse cero una vez se rompa el contacto, quedando de la forma: $f \cdot d''(t) = 0$. Usando las tres restricciones en un algoritmo de programación cuadrática, se halla la fuerza en cada uno de los puntos de contacto. Este método basado en restricciones fue propuesto por [Bar98].

2.6.3.2 Detección de colisiones. El sistema de detección de colisiones es el encargado de determinar la información de contacto de un mundo virtual en un tiempo determinado. Estas determinaciones presentan problemas geométricos que deben ser resueltos en un tiempo mínimo, dependiendo del tipo de aplicación se debe escoger entre resultados más precisos o más veloces.

2.6.3.2.1 Determinación de intersecciones. El primer problema es hallar la distancia mínima entre dos modelos tridimensionales para determinar si se encuentran separados ($d>0$), ínter penetrados ($d<0$) o en contacto ($d=0$). El acercamiento más fácil consiste en chequear un modelo polígono a polígono para ver si alguno intersecta un polígono de otro modelo, pero para la mayoría de los casos esta operación es extremadamente lenta e ineficiente. La propuesta más aceptada es envolver a los modelos en

volúmenes geométricos más simples, y chequear la intersección entre los volúmenes en lugar de los modelos, de esta forma se agiliza el proceso sacrificando un poco de precisión. Los volúmenes más usados son: esferas, elipsoides, cajas alineadas a los ejes, cajas orientables y armazones convexos. Las esferas son las menos precisas, pero las más rápidas cuando se quiere determinar la intersección. Dos esferas se intersectan cuando la distancia entre sus centros es menor a la suma de sus radios.

Si (distancia (c1, c2) < r1 + r2) entonces hay intersección. Las cajas alineadas a los ejes son un poco más precisas y menos rápidas que las esferas, el problema es que estas deben recalcularse cada vez que se rota el modelo. Para calcular la intersección se examinan de forma independiente las proyecciones de los modelos en cada uno de los ejes. Si en algún eje las proyecciones se intersectan, significa que entre las cajas existe intersección. Si ((Ax1 < Bx2) y (Bx1 < Ax2)) o ((Ay1 < By2) y (By1 < Ay2)) o

((Az1 < Bz2) y (Bz1 < Az2)) entonces hay intersección. Las cajas orientables son más precisas que las cajas alineadas a los ejes, pero más complicadas de evaluar. A diferencia de las anteriores, estas no se recalculan sino que se transforman junto con el modelo. El método para determinar si dos cajas orientables se intersectan en el espacio se llama “Teorema de los ejes separantes”. El teorema dice que existen 15 ejes en los cuales se deben chequear las proyecciones de las cajas para determinar si se intersectan o no. Si en alguno de los ejes se encuentra que las proyecciones no se intersectan, entonces las cajas no se intersectan. Los 15 ejes a los cuales se refiere el teorema son los 3 ejes locales de cada caja y los productos cruz resultantes de las combinaciones entre los ejes locales: Ax, Ay, Az, Bx, By, Bz, (Ax × Bx), (Ax × By), (Ax × Bz), (Ay × Bx), (Ay × By), (Ay × Bz), (Az × Bx), (Az × By), (Az × Bz), y (Az × Bx).

El radio de la proyección de una caja sobre un eje está definido por: ra = a1 .|Ax . n| + a2 .|Ay . n| + a3 .|Az . n| donde a1, a2, y a3 son las dimensiones de la caja,

A es el eje local, y n es el eje al cual se quiere proyectar. Para determinar si las proyecciones se separan en el eje n, se evalúa: Si (s > ra + rb) entonces las cajas no se intersectan. La distancia s es la proyección del vector T sobre el eje n : s = |T.n|

T es un vector construido entre los centros de las cajas. Si se evalúan los 15 ejes y en ninguno se encuentra que las proyecciones se separan, entonces se puede afirmar que las cajas se intersectan. Para lograr mayor precisión, los volúmenes envolventes pueden ser subdivididos jerárquicamente. El armazón convexo es un volumen envolvente definido por una versión a menor resolución del modelo. Se ajusta mejor a la geometría pero solo en pocas situaciones puede cumplir con los requerimientos de velocidad. Existen muchos otros algoritmos para hallar la distancia entre dos modelos tridimensionales, incluso para diferentes tipos de modelos. Los métodos explicados anteriormente solo funcionan con modelos convexos y poligonales.

2.6.3.2.2 Determinación para n objetos. Para una escena que contenga n objetos, se deben realizar O(n2) evaluaciones de intersección, lo cual representa un costo computacional muy alto cuando n es grande. Muchas técnicas de optimización se han propuesto para reducir el número de chequeos necesarios. Uno de los algoritmos asume una velocidad y aceleración máxima para los objetos, a partir de la cual calcula un límite mínimo de tiempo de colisión para programar las posibles colisiones. Una cola de posibles colisiones se mantiene y se actualiza cada vez que ocurre una colisión. Los elementos de la cola son ordenados por tiempo mínimo de colisión. Este tipo de algoritmos son llamados algoritmos de programación de tareas (Scheduling algorithms). Otros algoritmos explotan la coherencia espacial y temporal mediante el uso de árboles de ocho hijos (octrees).

2.6.3.2.3 Determinación del tiempo exacto de colisión. Un último problema consiste en determinar el tiempo exacto de la colisión. Debido a que la simulación física se mueve por intervalos de tiempo, una colisión puede ocurrir en medio de un intervalo. Supóngase que la simulación física se actualiza cada Δt, en un tiempo t0 se encuentra que dos objetos no están intersectados, pero en el tiempo $t0 + \Delta t$ resulta que los objetos están ínter penetrados. En el peor de los casos, un objeto puede pasar a través de otro sin ser notado por el sistema de detección de colisiones, creando un efecto de tunel. Escogiendo un Δt más pequeño para la simulación puede reducir el problema, pero no aliviarlo por completo. Existen varios esquemas para hallar el tiempo exacto de colisión. El método más sencillo es retroceder la simulación mediante la bisección del intervalo. Si en un instante $t0 + \Delta t$ se descubre que los objetos se ínter penetran, se retrocede la simulación al tiempo $t0 + (\Delta t)/2$ y se evalúa de forma sucesiva hasta encontrar el tiempo exacto en el que los objetos colisionan. Debido a que numéricamente es muy difícil determinar el tiempo exacto, se debe manejar un rango de error en la distancia de colisión.

3. DISEÑO DE LA BIBLIOTECA DE CLASES

3.1 OBJETIVOS

3.1.1 Objetivo General. Diseñar y construir una biblioteca de clases que le sirva a los ingenieros de sistemas y programadores como una herramienta de alto nivel para desarrollar aplicaciones de realidad virtual.

3.1.2 Objetivos Específicos. Proveer las herramientas matemáticas necesarias para trabajar en el espacio. Cargar modelos tridimensionales de varios formatos reconocidos para garantizar la compatibilidad de la información. Permitir el recorrido en tiempo real de ambientes virtuales complejos y modelos arquitectónicos mediante técnicas de optimización. Importar terrenos GIS y permitir su recorrido en tiempo real. Simular fenómenos naturales mediante física de partículas. Simular las leyes físicas y la interacción entre los modelos. Detectar intersecciones entre los modelos. Aprovechar la aceleración 3D de las tarjetas de video para el proceso de renderización.

3.1.3 Análisis de los objetivos. Debido a que los ambientes virtuales se simulan en un espacio tridimensional es indispensable contar con un arsenal de funciones matemáticas que faciliten la manipulación vectorial. La forma de cumplir con este objetivo es mediante la elaboración de las clases: vector tridimensional, matriz de transformación (4x4), cuaternio y plano. La ventaja de usar clases para este tipo de problemas radica en que se pueden sobrecargar los operadores y encapsular las operaciones matemáticas de modo que puedan ser aplicadas de forma intuitiva. La carga y manipulación de modelos tridimensionales en el espacio se logra mediante la implementación de un sistema de transformaciones y una estructura de modelo apropiadas, en las secciones 2.1, 2.2 y 2.3 de este trabajo se dan pautas para lograr este objetivo. Para permitir el intercambio de información con otros programas de modelamiento tridimensional, la biblioteca debe ser capaz de interpretar varios formatos de archivos tridimensionales. Los formatos de archivo escogidos fueron los de los programas 3D Studio Max y Alias Wavefront. Ambos formatos tienen estructuras de modelos diferentes, por lo tanto, para cada tipo de formato se debe construir una función de importación apropiada. La biblioteca debe ser capaz de determinar la geometría innecesaria de una escena y descartarla antes de su procesamiento para acelerar la renderización y de esta forma aumentar el número de cuadros por segundo de la animación. La meta del descarte de geometría es optimizar la simulación para mantener la aplicación como un sistema en tiempo real. Esto se logra mediante la implementación de algoritmos para la determinación de la visibilidad basados en la teoría expuesta en la sección 2.4. La estrategia escogida para este trabajo es usar octrees para subdividir el espacio y realizar de forma más rápida el descarte del campo

de visión. Para extender el uso y la aplicabilidad de la biblioteca se decidió incluir un módulo para el procesamiento de terrenos y mapas de elevaciones. Los terrenos y las mallas tridimensionales poseen propiedades diferentes a los modelos, por lo tanto su tratamiento también debe ser diferente. A estos también se les debe aplicar un método apropiado para la determinación de la visibilidad, en este caso el método escogido fue el algoritmo propuesto por [Rot98]. Para exhibir comportamientos reales, el sistema debe ser capaz de simular las leyes físicas básicas de nuestro entorno en tiempo real. Para esto es necesario implementar un método numérico que permita resolver sistemas de ecuaciones diferenciales. El sistema debe ser capaz de detectar las intersecciones entre las entidades espaciales y aplicar las fuerzas necesarias. Las entidades a simular son: partículas y cuerpos rígidos. Debido a que los sistemas de aceleración 3D son un estándar en la industria, se hace necesario construir la biblioteca sobre algún sistema de renderización existente. Estos sistemas aprovechan las funciones de las aceleradoras gráficas y proveen un mayor rendimiento. Para este trabajo se escogió OpenGL por su independencia de la plataforma y su esquema de código abierto.

3.2 DISEÑO DEL SISTEMA

3.2.1 Diseño general del sistema. El flujo del sistema mostrado es una expansión del esquema de transformación expuesto en la sección 2.2, este se ha ampliado para incluir la simulación física, la optimización de geometría y un mecanismo de control por parte del usuario.

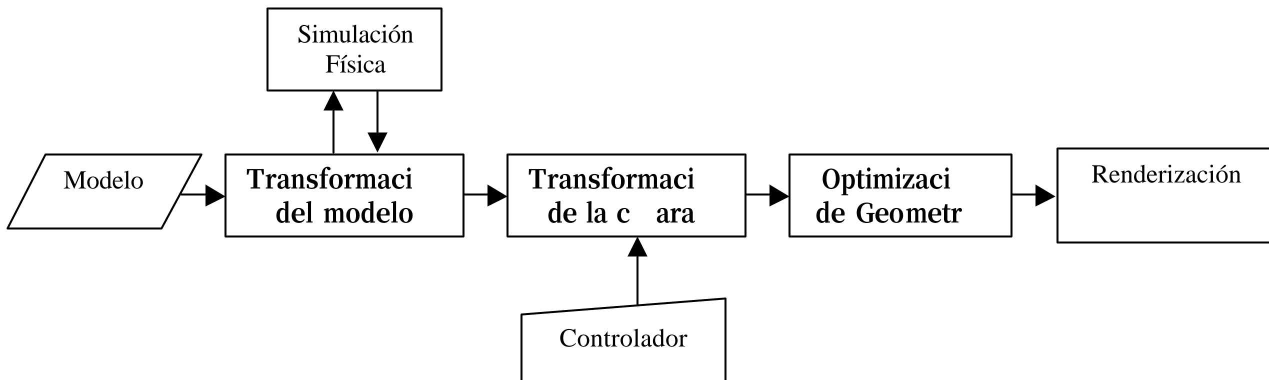

Figura 3. Flujo general del sistema.

La simulación física realiza dos grandes tareas: la detección de colisiones y la simulación de las leyes dinámicas. El detector de colisiones usa la geometría y el estado de los modelos para determinar la información de contacto. Para encontrar la información de contacto de forma rápida y precisa, el sistema de detección de colisiones debe resolver tres problemas: determinar exactamente los puntos y la normal de contacto, encontrar el tiempo exacto de colisión y usar algún método para reducir el número de chequeos necesarios entre las parejas de objetos. Usando la información de contacto, el sistema de respuesta a las colisiones aplica las fuerzas, los impulsos y las restricciones correspondientes, transformando de esta forma el estado de los modelos. El módulo de optimización recibe la geometría transformada y determina rápidamente la visibilidad de la escena. El descarte de geometría a nivel de objetos debe ser realizado por la biblioteca, el descarte a nivel de triángulos es hecho por OpenGL. El orden en el que se realice el descarte influye en el rendimiento de la aplicación, por lo tanto se propone seguir el siguiente esquema: descartar la geometría

fuera del campo de visión usando octrees, generar el nivel de detalle apropiado para los modelos restantes y por último descartar los polígonos ocultos.

3.2.2 Diseño de clases. A continuación se expone el diseño de las clases más relevantes del sistema. La descripción detallada de estas clases se encuentra en el Manual de Referencia adjunto a este trabajo.

3.2.3 Sistema de transformación. El sistema de transformación debe entregarle al renderizador la matriz de transformación a través de la cual la geometría será proyectada en la pantalla. Cada objeto en el espacio posee una transformación independiente, por lo tanto, cada vez que se envíe un objeto al renderizador se debe hacer con la matriz correspondiente. El estado de los objetos se almacena en la clase Transformable. A través de sus métodos se modifican la posición, orientación y escalamiento de estos. Cuando se va a renderizar el objeto, la función obtenerMatriz() provee la matriz de transformación apropiada para entregarle al renderizador. Para modificar el estado de los objetos a través de una entrada del usuario, se usa la clase Controlador. Mediante el método actualizar() se cambia el estado de un transformable dependiendo del estado de un dispositivo de entrada. La clase Cámara hereda de Transformable porque como cualquier otro objeto, tiene una posición y una orientación en el espacio. Además, la cámara mantiene un volumen de visión definido por seis planos. Cuando se especifica el método usar(), las matrices de transformación de todos

los “transformables” que luego se especifiquen son multiplicadas por la inversa de la matriz de transformación de la cámara, de esta forma el renderizador muestra a los objetos vistos a través de la cámara. Usando los planos se construyen métodos para determinar la visibilidad de ciertos objetos.

3.2.4 Modelos. En la sección 2.1 se expuso la estructura de los modelos, su representación se maneja mediante cuatro clases: Contenedor, Cara, Objeto y Modelo. En la clase Contenedor se almacenan arreglos para cada tipo de componente del modelo: vértices, colores, normales, coordenadas de textura, texturas y materiales. En la clase Cara se guardan índices a los arreglos de datos de un contenedor. El método renderizar() envía una cara al renderizador, para poder realizar el descarte de geometría a nivel de caras este método recibe la referencia a una cámara. La clase Objeto contiene un arreglo de caras, un contenedor y un volúmen envolvente. Por lo general, las caras apuntan al contenedor del objeto. Con el volúmen envolvente la clase puede determinar si el objeto intersecta a cualquier otro objeto, para esto se usa el método intersecta(). La función cargar() permite importar un objeto desde disco. La clase Modelo hereda de Transformable, contiene un arreglo de clases Objeto y un volumen envolvente que cubre todo el modelo. De esta forma los modelos están compuestos por objetos que pueden mantener transformaciones independientes. Los métodos son similares a los de la clase Objeto.

3.2.5 Simulación física. Las dos entidades físicas a simular son: partículas y cuerpos rígidos. Las partículas heredan de Transformable debido a que poseen una posición, sin embargo no tienen orientación ni escalamiento. Además, las partículas poseen masa, carga, velocidad y aceleración. En lugar de almacenar la aceleración en la clase, se almacena la fuerza total que actúa sobre la partícula en un instante dado. Para hallar la aceleración se divide la fuerza entre la masa. De esta forma se pueden ir acumulando fuerzas a medida que se determinan. El método integrar() halla la nueva velocidad a partir de la aceleración, y luego halla la posición a partir de la velocidad. El método renderizar() dibuja la partícula en la posición actual. La clase CuerpoRígido define propiedades físicas a los objetos. Además de poseer las propiedades que tienen las partículas, incluye las cantidades para el movimiento angular y la conservación del momento. El cuerpo rígido almacena la posición del centro de masa porque puede ser diferente a la posición o al centro del objeto. El simulador físico es el encargado de determinar las fuerzas que actúan sobre las entidades físicas y llenar apropiadamente los acumuladores de fuerzas.

3.3. IMPLEMENTACIÓN

3.3.1 Detalles técnicos. La biblioteca se bautizó con el nombre de Tridium. Se construyó con el compilador Borland C++ 5.5.1 y el enlazador Turbo Link versión 2.0.68.0 para Win32, ambos obtenidos de forma gratuita de http://www.borland.com/freecompiler. Se trabajó sobre el sistema operativo Windows

98 segunda edición, sin embargo, el código se puede portar a otras plataformas sin realizar mayores cambios. Para la renderización se usó la implementación de OpenGL versión 1.1.0 de Microsoft. Bajo estas características, la ultima compilación de la biblioteca no generó ni errores ni advertencias. Se crearon aproximadamente 60 clases, para un total de 9.838 líneas de código, 58 archivos de declaración (.h), 57 archivos de implementación (.cpp), un tamaño de código de 188.174 bytes y un tamaño de datos de 32.427 bytes. La construcción tardó año y medio de trabajo no continuo. Los requerimientos del hardware dependen de la implementación de OpenGL que se use y del sistema operativo al que se porte. Actualmente la biblioteca corre sobre Windows 9x/2000/NT y versiones de OpenGL iguales o superiores a la 1.1. Se recomienda usar algún tipo de tarjeta aceleradora de video para obtener tiempos de respuesta razonables. Gracias a un diseño basado en capas, la biblioteca no realiza llamadas directas al procesador y por lo tanto es independiente de este. Las llamadas al sistema operativo las hace a través de unas pocas clases (manejo de ventanas, cronómetro y entrada) que pueden ser reemplazadas fácilmente. Si el código de la aplicación se limita a usar las funciones de la biblioteca, se mantiene independiente del sistema operativo y del hardware.

3.3.2 Módulos. La biblioteca está dividida en siete módulos: sistema, utilidades, geometría, núcleo 2D, núcleo 3D, terrenos y física. El módulo más básico es el del sistema, el cual se encuentra en el nivel más bajo del esquema de dependencias. Los módulos de los niveles más altos se construyen a partir de las funciones que proveen

los módulos debajo de ellos. El módulo del sistema es el encargado de interactuar con el sistema operativo, realiza operaciones como: abrir ventanas, establecer el modo de video, intercambiar mensajes, leer la entrada del usuario, etc. El módulo de utilidades básicas contiene diversas clases y funciones para mejorar las que provee el lenguaje de programación. En este se crearon clases para manejar cadenas, archivos y listas genéricas entre otras. En el módulo de geometría se definen las funciones para trabajar en el espacio, contiene las clases: vector, matriz, cuaternio, plano, rayo y polígono. El manejo de imágenes bidimensionales y la carga de archivos gráficos se realiza en el núcleo 2D. El módulo núcleo 3D maneja el sistema de transformación, los modelos y el descarte de geometría. El módulo de terrenos está compuesto por: el importador de mapas de elevaciones, el generador de terrenos aleatorios y el algoritmo para recorrer terrenos grandes (quadtree). Por último, en el módulo de la física se maneja la detección de colisiones y la simulación de la leyes dinámicas de los mundos virtuales. En el archivo de cabecera tridium.h se incluyen los demás archivos de la biblioteca, en este se puede apreciar el contenido de cada módulo y el orden de las dependencias.

3.3.3 Estilo del código. Con el fin de conservar la uniformidad, el orden y la claridad en el código se emplearon algunas convenciones estéticas y funcionales. Estas se exponen a continuación: El nombre de los tipos de datos definidos termina en “_t”, cuando el nombre tiene más de una palabra se usa el caracter “_” para separarlas, por ejemplo: vector_t, matriz_t, rigid_body_t, etc. El nombre de las variables, clases, estructuras y funciones están dados en inglés para mantener la uniformidad con el

lenguaje de programación y compartir el código internacionalmente. Todas las clases de la biblioteca heredan de una clase genérica llamada class_t con el fin de mantener una base común. La clases se declaran en un archivo de cabecera (.h) y se implementan en un archivo de código (.cpp). El archivo de cabecera de la clase se añade al archivo de cabecera principal (tridium.h) y el archivo de código de la clase incluye a tridium.h. De esta forma se mantiene enlazada toda la biblioteca, si ocurre una actualización en un archivo, los archivos que dependan de él son automáticamente recompilados. En los archivos de cabecera no se incluye ni datos ni código. Los archivos de cabecera se definen dentro de una directiva del preprocesador para evitar que sean incluidos más de una vez (#ifndef). La separación de bloques se hace mediante tabulaciones de un espacio. Para separar la implementación de las funciones y la declaración de las clases se usan líneas de guiones ("//------------------"). En las clases, el nombre de las propiedades es declarado primero que el de los métodos. Los tipos de datos estándar del lenguaje son redefinidos para garantizar la portabilidad del código.

4. CONCLUSIONES

4.1. RESULTADOS

4.1.1 Resultados obtenidos usando el descarte de geometría.

Las técnicas de descarte de geometría empleadas fueron: octrees para recorrer modelos tridimensionales y quadtrees para recorrer terrenos y mallas. El descarte de geometría a nivel de triángulos (descarte de caras traseras y campo de visión) es aplicado por OpenGL, por lo tanto no fue implementado en la biblioteca.

4.1.1.1 Octrees.

Para la prueba se usó el modelo de un parque con 29.174 caras y 16.586 vértices. El modelo del parque es cortesía de Elmar Moelzer (moelzere@aon.at) y Ben Humphrey (digiben@GameTutorials.com). Las demostraciones se corrieron sobre una máquina Pentium III-550Mhz, con 256MB de RAM y una aceleradora gráfica Trident Blade 3D con 8 MB de memoria. El modelo del parque fue almacenado en un octree. Las dimensiones del modelo eran de 125 unidades cúbicas, para lo cual se usó un número máximo de 50 caras por nodo y un tamaño de nodo mínimo de 10 unidades. El tiempo de construcción del octree fue de 2.6 segundos (3'118.716 ciclos de CPU). La prueba consistió en medir el rendimiento del sistema mientras un espectador recorría el parque sin usar descarte de geometría y

usando octrees. Los resultados obtenidos se expresan a continuación: Cuando se usa el octree, el número de triángulos renderizados cambia drásticamente dependiento de la posición del espectador. En el peor de los casos el octree obtiene el mismo rendimiento que cuando no se usa.

4.1.1.2 Quadtrees. Para la prueba se usó un mapa de 512 x 512 datos de elevaciones tomadas por un satélite sobre una región al sur del cruce de Hainnes en el territorio Yukón, Canadá. El modelo es el mismo empleado en [Rot98]. El mapa produjo una malla de 131.072 triángulos (5122 / 2). La máquina usada y el tipo de prueba aplicada fueron similares a las del octree. Los resultados se expresan a continuación: Si se compara con el octree, el quadtree es más eficiente y presenta menos variaciones en el número de triángulos renderizados por segundo. Esto se debe a que las mallas presentan diferentes características y son más controlables.

4.1.2 Resultados obtenidos en la simulación física. Interconectando tres partículas mediante dos resortes se simuló el comportamiento de un péndulo en tiempo real. La partícula azul se mantuvo fija, se aplicó la fuerza de gravedad sobre el sistema y mediante el teclado se aplicaron fuerzas sobre la partícula roja. De forma similar, se interconectó un arreglo bidimensional de partículas mediante resortes para simular el comportamiento de un pedazo de tela. Se aplicaron fuerzas sobre los extremos inferiores de la tela mientras la fila superior de partículas se mantuvo fija. Las fuerzas

se propagaron por el sistema creando una animación real de la dinámica de la tela. En ninguno de los ejemplos anteriores se restringió el movimiento de las partículas debido a las propiedades de los resortes o por colisiones con otros objetos. Los resortes, capaces de deformarse indiscriminadamente producían algunos comportamientos indeseables, por ejemplo, dos partículas llegaban a acercarse una distancia menor a la longitud de reposo de su resorte. Mediante el emisor de partículas se pudieron generar comportamientos estocásticos de sistemas de partículas. Por lo general, los fenómenos naturales como la lluvia, el fuego y el humo entre otros, siguen este tipo de comportamientos. En el ejemplo se muestra la animación de un destello de partículas creado con el emisor. Aunque la simulación de la dinámica de los cuerpos rígidos está completamente implementada, dos cosas no funcionaron como se esperaba. La primera es que debido al uso de esferas envolventes sin subdivisión jerárquica, la determinación del contacto no es precisa, la normal de colisión obtenida es el vector entre los centros de las esferas, por lo tanto, como la fuerzas de colisión actúan directamente sobre los centros de masa no se pudo apreciar el movimiento angular ocasionado por la interacción de dos o más cuerpos. El segundo problema es que debido a que no se implementó un mecanismo de restricción para el contacto de reposo, en ocasiones los cuerpos llegan a ínter penetrarse.

4.1.3 Otros resultados. Otros logros obtenidos en la persecución de los objetivos del trabajo se expresan a continuación: Con el estudio de la especificación de los formatos Autodesk 3DS y Wavefront OBJ se implementaron clases para cargar

satisfactoriamente estos tipos de modelos. De esta forma el usuario puede hacer sus diseños en Autocad, 3D Studio o Lightwave e importarlos directamente dentro de la biblioteca. Las datos importados corresponden únicamente a las entidades de los modelos poligonales: vértices, caras y materiales. Los formatos de imágenes importados son los de Windows BMP y Truevision TGA. No se implementó la importación de imágenes comprimidas.

El diseño orientado a objetos permitió encapsular todas las transformaciones del espacio en una clase de la cual heredaron las demás entidades espaciales como luces, cámaras, modelos, etc.

El sistema de transformaciones basado en cuaternios permitió movimientos con seis grados de libertad. Usando solamente el esquema de matrices que provee OpenGL resulta complicado lograr la misma flexibilidad en las transformaciones.

Con el sistema de terrenos se logró la importación de datos de elevaciones a partir de imágenes satelitales mientras que con el generador de terrenos se crearon mapas de elevaciones artificiales convincentes basados en modelos fractales.

Usando el módulo de terrenos se pudieron construir otro tipo de aplicaciones, por ejemplo, a partir de una imagen MRI de una cavidad cerebral el sistema construyó una malla tridimensional del modelo permitiendo una mejor visualización de la cavidad.

4.2 CONCLUSIÓN

Con este trabajo se presenta una biblioteca de clases en C++ para el desarrollo de aplicaciones de realidad virtual. Para alcanzar este objetivo se solucionaron dos grandes problemas: permitir el recorrido del mundo virtual en tiempo real y simular las leyes físicas del entorno. Para facilitar el recorrido y la manipulación de los objetos en el espacio se desarrolló un sistema de transformaciones basado en cuaternios. Los cuaternios presentan varias ventajas sobre los otros métodos para representar orientaciones porque permiten concatenar rotaciones de forma correcta, fácil y utilizando menos espacio en la memoria. Gracias al uso de cuaternios, las entidades espaciales (luces, cámaras, etc..) pueden alcanzar movimientos con seis grados de libertad. Mediante técnicas para la determinación de la visibilidad el sistema es capaz de descartar rápidamente los polígonos innecesarios de una escena y mantener un tiempo de respuesta más pequeño. Las técnicas implementadas fueron octrees para modelos poligonales y quadtrees para mallas. Cuando las escenas son menos densas el octree obtiene mayor rendimiento, para mejorar el rendimiento del sistema en las escenas densas se deben implementar algoritmos de niveles de detalle y descarte de polígonos ocultos. El comportamiento físico se simuló mediante la resolución progresiva de las ecuaciones de la dinámica. Para resolver las ecuaciones diferenciales se empleó la rutina de Runge-Kutta debido a que los métodos como Euler y del Punto Medio acumulaban un error numérico muy grande y causaban la inestabilidad del sistema. Sin embargo, cuando se empleaban resortes muy rígidos el

sistema todavía se volvía inestable. Uno de los principales problemas en la simulación física fue el de la detección de colisiones y sobre todo, el de la determinación exacta del contacto. En este trabajo se implementaron las esferas envolventes sin subdivisión jerárquica, las cuales no pueden determinar de forma precisa la información de contacto, por esta razón el movimiento angular de los cuerpos rígidos, aunque está implementado, no es percibible. Otros problemas que no quedaron resueltos en la simulación física fueron: la respuesta al contacto de reposo y la restricción de movimientos. El código de la biblioteca fue escrito pensando en la portabilidad, por lo que ahora resulta muy fácil trasladarlo a otras plataformas mediante el reemplazo de unas pocas clases. El diseño orientado a objetos facilitó su crecimiento mientras se construía una base tecnológica firme. Las herramientas de depuración desarrolladas fueron indispensables para detectar anomalías y cuellos de botella en el rendimiento del programa. No se usó ningún sistema de código concurrente ni control de versiones, mas que un backup periódico, lo que provocó retrasos debido a varios inconvenientes. La mayor fuente de información del trabajo provino de Internet, especialmente de grupos de discusión y reportes de conferencias de SIGGRAPH. Sin embargo, el entendimiento de estos temas no hubiera sido posible sin las bases necesarias en las siguientes materias: cálculo vectorial, ecuaciones diferenciales, álgebra lineal, métodos numéricos, física, programación y sistemas dinámicos. El mayor aporte de este trabajo a la comunidad universitaria radica en que es un proyecto abierto dispuesto a ser apropiado, mantenido y mejorado por estudiantes y profesores.

4.3 RECOMENDACIONES

4.3.1 Aplicabilidad. A continuación se proponen algunas aplicaciones y usos prácticos de esta tesis.

4.3.1.1 Arquitectura. Usando la biblioteca es posible importar modelos arquitectónicos desde Autocad y permitir su exploración en tiempo real mediante cámaras. Para modelos muy grandes se usan las técnicas de descarte de geometría. De esta forma se pueden recorrer proyectos de construcción antes de ser implementados, vender bienes raíces desde lugares remotos o crear aplicaciones de turismo virtual.

4.3.1.2 Geografía. El módulo de terrenos permite recorrer y analizar características geográficas de una región en tres dimensiones y en tiempo real. Esto constituye una herramienta poderosa en aplicaciones militares. También permite explorar datos obtenidos a través de sistemas geográficos de información (GIS - IGAC) o del sistema ambiental de información (SIAC).

4.3.1.3 Medicina. Con el módulo de terrenos se pueden construir mallas a partir de radiografías. La exploración y manipulación de modelos y mallas tridimensionales

permite analizar y estudiar datos sobre el cuerpo humano evitando el uso de cadáveres o esqueletos reales.

4.3.1.4 Educación e investigación. Con la construcción de modelos tridimensionales se facilita la enseñanza en diversas materias de ingeniería como mecánica, electromagnetismo, física ondulatoria, cálculo vectorial y álgebra lineal entre otras. La simulación física permite la investigación y exploración de las leyes del universo en tiempo real.

4.3.1.5 Ingeniería. Aplicaciones de ingeniería mecánica, ingeniería civil y robótica pueden ser simuladas en tres dimensiones y con física real mediante la biblioteca.

4.3.1.6 Otras. La construcción de videojuegos, efectos especiales para cine y herramientas de superación para niños discapacitados (con problemas mentales) son otras posibles aplicaciones.

4.3.2 Proyección. Algunas ideas para mejorar, ampliar y desarrollar este trabajo se exponen a continuación.

4.3.2.1 Descarte y optimización de geometría. El sistema para la determinación de la visibilidad y descarte de geometría se puede mejorar considerablemente mediante la implementación de técnicas de niveles de detalle (ver [Hop96]) y descarte de polígonos ocultos (ver [Zha97]). De igual forma, se pueden evaluar otros métodos para el recorrido en tiempo real de terrenos grandes (ver [Duc97] y [Lin01]). El método de octrees se puede rediseñar, optimizar y extender para manejar geometría dinámica. Por último, la representación del modelo tridimensional se puede optimizar para aprovechar los mecanismos de listas de ejecución y arreglos de vértices que provee OpenGL.

4.3.2.2 Simulación Física. El módulo de detección de colisiones debe ser ampliado para determinar de forma más rápida y más precisa la información del contacto, varias opciones se pueden encontrar en la página del grupo de investigación GAMMA (ver Enlaces). La fuerza debido al contacto de reposo y las restricciones también deben ser calculadas usando alguno de los métodos propuestos por [Mir95] o [Bar98]. También se pueden implementar otro tipo de fuerzas, por ejemplo la fricción. Para poder ser usada en robótica y mecánica, a la biblioteca se le deben añadir clases capaces de manejar uniones (joints) y restringir el grado de libertad de sus movimientos. Otros trabajos de grado que se pueden realizar con base en este trabajo son: simulación de la dinámica de fluidos para realidad virtual y simulación de las leyes de la termodinámica.

4.3.2.3 Sistemas distribuidos. La biblioteca se puede extender para funcionar sobre un sistema distribuido, de tal manera que se pueda acceder progresivamente a la geometría de un mundo virtual muy grande o que se puedan compartir los estados de los "transformables" y permitir la interacción con un grupo de navegantes virtuales provenientes de varios puntos de una red.

4.3.2.4 Interfaces electrónicas. Las ingenierías electrónica, mecánica y mecatrónica pueden aportar y beneficiarse de la biblioteca en varios proyectos: la construcción de mejores interfaces de entrada y salida, el desarrollo de un sistema de captura de información geométrica con láser, sistemas de sensado remoto e interfaces de comunicación con robots entre otras.

4.3.2.5 Animación. Se puede implementar un sistema para el manejo de la animación mediante cinemática inversa que permita capturar y estudiar los movimientos humanos, para medicina, deporte o cine (ver Enlaces - Jeff Lander).

4.3.2.6 Simulación biológica. La simulación de ecosistemas, procesos evolutivos y de otras características del medio ambiente también pueden ser implementadas para permitir el estudio de este tipo de comportamientos (ver Enlaces – Karl Sims).

4.3.2.7 Mejoras en el código. El código se puede extender mediante la portabilidad a otros sistemas operativos y plataformas (ej: Linux y Java). Usando los conocimientos vistos en compiladores se puede construir un lenguaje propio de la biblioteca que permita realizar operaciones sin necesidad de recompilar el código (similar a SQL o Matlab). Otro proyecto consistiría en integrar la biblioteca a Matlab. Nunca está de más la construcción de herramientas de depuración más eficaces, por ejemplo, un manejador propio de memoria o un medidor de rendimiento a varios niveles (el que está actualmente implementado bajo el nombre de profiler solo mide el rendimiento en un nivel).

4.3.2.8 Audio. La implementación de un sistema de sonido ambiental mediante OpenAL abre otro campo gigante de investigación y desarrollo.

BIBLIOGRAFÍA

ABRASH, Michael. Graphics Programming Black Book. Coriolis Group. 1998.

AIREY, John. ROHLF, John. BROOKS, Frederick. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In Proceedings 1990 Symposium on Interactive 3D Graphics. 1990.

BARAFF , David. WITKIN, Andrew. “Physically based modeling”. Siggraph ’98 course notes. 1998.

BEER. JOHNSTON. Mecánica vectorial para ingenieros, sexta edición. WCB/McGraw Hill. 1997.

OPENGL ARCHITECTURE REVIEW BOARD. OpenGL Reference Manual. Addison-Wesley. 1992.

BSP FAQ. http://reality.sgi.com/bspfaq. 1998.

DUCHAINEAU, Mark. WOLINSKY, Murray. SIGETI, David. MILLER, Mark. ALDRICH, Charles. MINEEV-WEINSTEIN, Mark. ROAMing Terrain: Real-Time Optimal Adapting Meshes. IEEE Visualization. 1997.

FOLEY, James. VAN DAM, Andries. FEINER, Steven. HUGUES, John. Computer Graphics: Principles and Practice, Addison-Wesley. 1990.

GOMEZ, Miguel. Simple Intersection Tests for Games. http://www.gamasutra.com/features/. 1999.

GOTTSCHALK, Stephan. Collision Queries Using Oriented Bounding Boxes. University of North Carolina at Chapel Hill. 2000.

GREENE, N. Hierarchical Z-Buffer Visibility. SIGGRAPH ’93 Conference Proceedings. 1993.

HECKER, Chris. “Physics”. Game Developer Magazine (Octubre 96). 1996.

HOPPE, Hugues. Progressive Meshes. Computer Graphics Proceedings of Siggraph '96. 1996.

LANDER, Jeff. "Collision Response: Bouncy, Trouncy, Fun". http://www.gamasutra.com/features/20000208/lander_01.html. 2000.

LINDSTROM, Peter, PASCUCCI, Valerio. Visualization of Large Terrains Made Easy. Proceedings of IEEE Visualization. San Diego, California. 2001.

LIN, Ming Chieh. Efficient Collision Detection for Animation and Robotics. University of North Carolina at Chapel Hill. 1993.

LINDSTROM, Peter. KOLLER, David. RIBARSKY, William. HODGES, Larry. FAUST, Nick. TURNER, Gregory. Real-Time, Continuous Level of Detail Rendering of Height Fields. Computer Graphics Proceedings (Siggraph '96).

LOISLEL, Sébastien, Zed3D – A Compact Reference for 3D Computer Graphics Programming, version 0.95b. 1996.

MARTZ, Paul. Generating Random Fractal Terrain. http://www.gameprogrammer.com/fractal.html. 1997.

The Matrix and Quaternions FAQ. Version 1.4. 1998.

MICROSOFT CORPORATION. Microsoft Win32 Programmer's Reference. Win32.hlp. 1996.

MIRTICH, Brian. CANNY, John. Impulse Based Simulation of Rigid Bodies. University of California at Berkeley. 1995.

NEIDER, Jackie. DAVIS, Tom. WOO, Mason. OpenGL Programming Guide. Addison-Wesley. 1993.

PRESS. FLANNERY. TEUKOLSKY. VETTERLING. Numerical Recipes in C. Cambridge University Press, Cambridge, England, 1988.

RÖTTGER, Steffan. HEIDRICH, Wolfgang. SLUSALLEK, Philipp. SEIDEL, Hans-Peter. Real-Time Generation of Continuous Levels of Detail for Height Fields. Proceedings of the 6th international conference in central Europe on Computer Graphics and Visualization. 1998.

SCHILDT, Herbert. Borland C++, Manual de referencia. Osborne/McGraw Hill.ISBN: 0-07-882230-0. 1997.

Coordinate Systems, Vectors, Planes, FAQ. Version 1.3. 1998.

ZHANG, Hansong. MANOCHA, Dinesh. HUDSON, Tom. HOFF, Kenneth. Visibility Culling using Hierarchical Occlusion Maps. SIGGRAPH '97 Conference Proceedings. 1997.