UNIVERSIDAD DE LAS CIENCIAS INFORMATICAS
UCI
FACULTAD 5 REALIDAD VIRTUAL

MÓDULO DE EFECTOS VISUALES PARA MOTORES DE REALIDAD VIRTUAL
Luces y Sombras Dinámicas

TRABAJO DE DIPLOMA EN OPCIÓN AL TÍTULO DE
INGENIERO EN CIENCIAS INFORMÀTICAS

Autores
Yaíma Nodarse Valdés
Lien Muguercia Torres

Tutores
Dr. José Ignacio Guzmán Montoto
Ing. Yanoski R. Camacho Román

Ciudad de la Habana
Mayo, 2007

DECLARACIÓN DE AUTORÍA

Declaramos que somos las únicas autoras de este trabajo, y autorizamos al Proyecto Herramientas de Desarrollo para Sistemas de Realidad Virtual de la Facultad 5 de la Universidad de las Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmamos la presente a los ____ días del mes de _____ del año _____.

Autores:

Yaíma Nodarse Valdés

_____

Lien Muguercia Torres

_____

Tutor:

Dr. José Ignacio Guzmán Montoto

_____

Ing. Yanoski Camacho Román

_____

Agradecimientos

Muchos han contribuido de una forma u otra con la realización de este proyecto, a todos ellos nuestro más sincero agradecimiento, en especial a Yanoski, Pepe y Fernando, por su paciencia y el día a día, por adentrarnos en el maravilloso mundo de los gráficos por computadoras y ser nuestra guía en todo momento. A la UCI por darnos la oportunidad de formar parte de esta tropa del futuro y formarnos como los profesionales que somos.

De Lien

A mi mamá, por su amor incondicional y el apoyo que me ha dado durante toda la vida, por estar siempre presente. A Armando, por ser más que un padre para mi, por ser mi guía, por enseñarme a pensar en el mañana, por exhortarme a ser mejor cada día y a superarme. A Plutu por ser como mi hermano y estar siempre disponible para cuando exploto. A mi tía Isabel por su cariño y preocupación, por sus consejos cuando más los necesitaba. A Bombi por su apoyo, fuerza y cariño durante toda la carrera. A los gordos, por ser los amigos sinceros y naturales que en cualquier momento están disponibles. A todos mis familiares, que de alguna manera siempre han estado atentos a mi carrera y me han apoyado, a todos, gracias.

De Yaima

A mi madre y mi padre, por su paciencia y amor todos estos años, por alentarme e inculcarme juntos el amor al estudio y las ansias de superación. A mis tres pequeños hermanos (aunque ya bastante grandecitos) que fueron y seguirán siendo el motor impulsor de mis ansias de superación, pues ellos son mi mayor inspiración. A todos mis abuelos por estar siempre tan pendientes y preocupados por mi y mis estudios. Para Tata que aunque ya no este conmigo, me colmó con su amor y su deseo de que fuera alguien en la vida. A Sachel quién siempre estuvo a mi lado alentándome y apoyándome en todos estos cinco años. A todos mis amigos por su apoyo y por estar siempre a mi lado cuando los necesite.

Dedicatoria

A mi mamá, Armando. A Plutu.

A mi familia.

Lien

A mis padres, hermanos y abuelos.

A Tata y a mi familia.

Yaíma

Resumen

Tradicionalmente, lograr la iluminación y el sombreado eficientes ha sido un tema de discusión en el área de gráficos por computadora. Lograr el nivel de realismo deseado requiere de un alto consumo de recursos y a veces no se cumple con las expectativas. Este trabajo propone una solución para incorporar mayor realismo a las escenas virtuales a través de luces y sombras dinámicas.

Partiendo del estudio de varios algoritmos que dan solución al problema de la iluminación y sombreado dinámico de modo eficiente, se plantean ventajas y desventajas según sus características y se propone un módulo para el trabajo con luces y sombras a través de los *shaders* (códigos que permiten introducir nuevos algoritmos en el hardware de video) en concordancia con las necesidades del cliente.

Como resultado, este módulo se acopló a la herramienta de desarrollo de aplicaciones de Realidad Virtual de la facultad 5, aportándole una tecnología de primera línea que sitúa a la herramienta al nivel de este tipo de sistemas a escala mundial.

Índice

Introducción

La tecnología ha progresado más rápido que nuestra habilidad para siquiera imaginar que vamos a hacer con ella. Hoy la Realidad Virtual (RV), ha trastocado la percepción y está revolucionando el mundo, no sólo de la informática sino también de diversidad de áreas como la medicina, la arquitectura, la educación y la ingeniería entre otros.

La RV entra en un exclusivo rango de herramientas para hacer, en el cual el usuario puede incursionar creativamente, hasta donde el límite de su imaginación se lo permita. Allí radica muy posiblemente el mayor atractivo, por cuanto la imaginación y la creatividad tienen la oportunidad de ejecutarse en un "mundo" artificial e ilimitado.

La Universidad de las Ciencias Informáticas (UCI) se ha incorporado a la investigación y desarrollo de este tema, para ello ha buscado la colaboración conjunta con otras empresas con algo más de experiencia siendo el “Centro de Investigación y Desarrollo #2” (CID2), conocido en el mercado como SIMPRO (Simuladores Profesionales) un ejemplo de lo antes expuesto.

Esta colaboración esta subdividida en proyectos productivos, cada uno con un campo distinto en el mercado: desarrollo de juegos, simuladores de conducción, simuladores para la defensa (tiro), simuladores quirúrgicos, etc.

Existe un proyecto encargado de suministrarle al resto las herramientas necesarias para la implementación y desarrollo del rol de cada uno. Dicho proyecto desarrolló una herramienta básica que reúne las funcionalidades comunes a todos estos proyectos, pero dicha herramienta se encuentra aun en un estado primario, necesitando de efectos visuales para el tratamiento de Luces y Sombras Dinámicas, es decir, aquellas que son producidas por y sobre objetos que están en movimiento, ya que solamente está implementado para objetos estáticos. Dicha cualidad es muy necesaria si se quiere dar el realismo adecuado a los entornos de realidad virtual, tanto en simuladores como en juegos, donde el realismo gráfico del entorno impulsa en

gran medida la inmersión del usuario. Para mantener la ilusión de credulidad en un mundo de RV se hace necesario un ambiente iluminado y sombreado, tal que su semejanza con el mundo real sea lo más certera posible.

¿Como implementar los efectos de luces y sombras para lograr un alto realismo en la escena sin consumir muchos recursos? Para ello se propone la creación de un módulo que reúna los algoritmos que den solución a este problema, de esta manera los efectos visuales constituyen el objeto de estudio y el trabajo con la iluminación y las sombras dinámicas el campo de acción.

El objetivo principal del trabajo es implementar un módulo para el tratamiento de luces dinámicas, para ello se debe lograr que dicho módulo sea completamente transparente al usuario, flexible a actualizaciones futuras, que utilice eficientemente las cualidades de las tarjetas gráficas y que soporte la biblioteca gráfica *OpenGl.* También se realizará el estudio de los algoritmos de sombras más usados actualmente y se propondrá el más factible según sus características y las necesidades del cliente, pero no se le dará implementación.

Se plantean entonces un grupo de tareas que permitirán satisfacer los objetivos, y que se pueden resumir en las siguientes:

- Estudio de las características básicas de los módulos de efectos visuales para sistemas de realidad virtual.
- Análisis de algoritmos, tecnologías y tendencias actuales utilizados en la creación y visualización de estos efectos visuales.
- Análisis de las potencialidades de los lenguajes utilizados en el mundo para el desarrollo de dichos efectos.
- Estudio de las potencialidades de las tarjetas gráficas a utilizar.
- Análisis y diseño de una biblioteca de clases que dé solución a los problemas planteados.

Como resultado de este trabajo se pretende obtener un módulo que contenga los efectos de iluminación y sombreados necesarios para el soporte de situaciones reales que aparecen en la vida diaria. Este módulo será incorporarlo a la herramienta desarrollada por la UCI conjuntamente con la empresa SIMPRO, herramienta actualizable que le permitirá al programador un fácil desarrollo y creación de aplicaciones de RV, de manera que se centre en qué hacer para resolver sus problemas específicos utilizando las soluciones y facilidades aportadas por este módulo, y no en cómo crear soluciones básicas.

Capítulo 1 Fundamentación Teórica

Introducción

En el mundo real, podemos ver las cosas porque reflejan la luz de una fuente de luz, o porque ellos mismos son una fuente de luz. En gráficos por computadora, como en la vida real, el ser humano no está capacitado para ver un objeto a menos que esté iluminado o emita luz.

Las sombras ayudan a definir las relaciones espaciales entre objetos en una escena. Ella dice cuando un pie hace contacto con el suelo si el sujeto corre; cuando una pelota rebota da la información acerca de un punto donde se encuentra en un momento de tiempo determinado, etc. La sombra en un objeto ayuda a saber cuál objeto es por su forma y comportamiento, en gráficos por computadoras ayudan a realizar el realismo en una escena.

Para generar imágenes más realistas, se necesita hacer modelos más realistas para iluminación, sombras y reflexión.

Gracias a la programación gráfica de las tarjetas se pueden obviar las ecuaciones tradicionales de iluminación y de efectos de sombras y se puede experimentar e implementar con nuevas y variadas técnicas. Algunas de estas técnicas son más rápidas y realistas que los métodos tradicionales.

Este capítulo introduce los conceptos básicos de la iluminación de escenas virtuales así como el empleo de *shader* para introducir nuevos algoritmos en la tarjeta de video. Además se presentan explicaciones de algunos de los modelos matemáticos más usados en la iluminación de mundos virtuales. Describiéndose las diferencias principales existentes en los focos de luz y su influencia en la escena.

1.1 Modelos de Iluminación

En la naturaleza, cuando la luz es emitida desde un foco luminoso, ésta es reflejada desde innumerables objetos antes de alcanzar los ojos del espectador. Cada vez que es reflejada, una parte de la energía es absorbida por la superficie, otra parte es dispersada en direcciones aleatorias y el resto se dirige a otra superficie o a los ojos del espectador. El proceso anterior se repite hasta que la energía se reduce a cero o el espectador percibe la luz.

Los cálculos necesarios para reproducir exactamente el comportamiento de la luz en la naturaleza consumirían demasiado tiempo y recursos como para permitir visualizar escenas en tiempo real. De esta forma, pensando en reducir el tiempo de procesamiento de la iluminación, se han desarrollado modelos matemáticos que aproximan el comportamiento físico de la luz en la naturaleza.

Estos modelos matemáticos han sido divido en dos grandes categorías: aquellos que representan luces ambientales y los que se refieren a luces directas.

Las luces ambientales son las que han sido dispersadas tantas veces producto de la reflexión con las superficies, que no se puede determinar la posición del foco desde el cual fueron emitidas. Las luces indirectas usadas por los fotógrafos es un buen ejemplo de luces ambientales. En los gráficos generados por computadoras, estas luces se representan sólo mediante su color y la intensidad, no aportando reflexiones especulares ni degradadas difusas.

Contrarias a estas, las luces directas inciden en la superficie sin ser dispersadas por otros objetos. Estas luces producen reflejos especulares en la superficie y su dirección de incidencia es usada para calcular factores de sombreado que modulan la intensidad y el color de la luz en la superficie.

Entre los modelos matemáticos más utilizados para representar contribuciones directas, se destaca el Phong. Este fue creado por Bui Tuong Pong y permite producir un alto grado de realismo en objetos tridimensionales combinando tres elementos básicos: el componente Ambiental, el Difuso y el Especular. [14]

El componente Ambiental produce una iluminación constante en la escena. Todos los píxeles que conforman el objeto reciben el mismo color pues este componente no depende de otros factores como la dirección de la luz, el vector normal a la superficie, etc. Este componente es calculado rápidamente, pero, por sí solo, produce resultados muy poco realistas, produciendo objetos con apariencia plana.

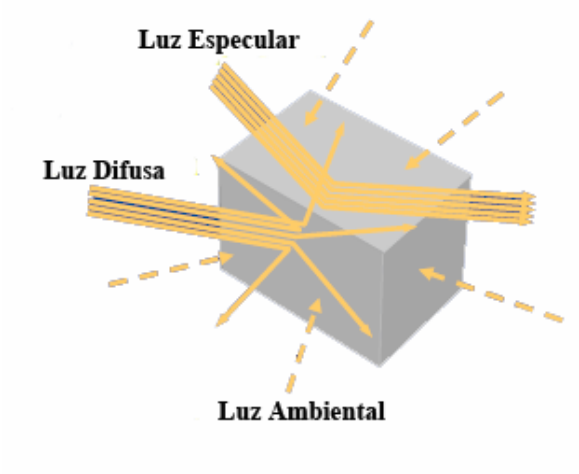

Figura 1: Modelo de Iluminación de Phong

Contrario al ambiental, el componente Difuso depende de la dirección de incidencia de la luz y de la normal a la superficie. Resultando en variaciones del color para cada píxel de la superficie, producto del cambio en la dirección de incidencia y de las distorsiones que se pueden presentar en la orientación del vector normal a la superficie.

El componente difuso usa el ángulo entre el vector normal y la dirección de la luz para calcular la intensidad en cada píxel. Siendo mayor mientras más pequeño sea este ángulo.

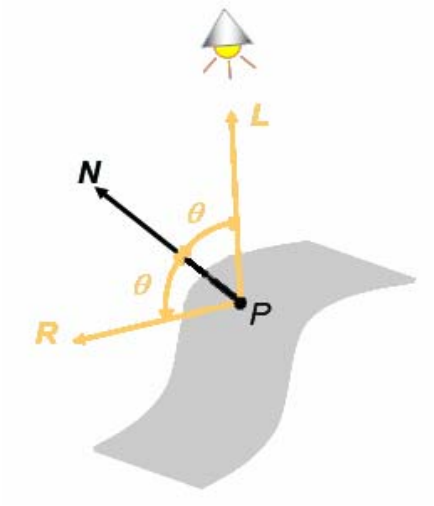

Figura 2: Dirección de incidencia de la luz.

El componente Especular representa los reflejos especulares que se producen en la superficie. Esta componente produce zonas de alta intensidad y brillo en contraste con otras porciones donde genera valores de reducida intensidad. De esta forma, el componente especular simula el efecto producido en la intensidad de la luz que se percibe cuando el rayo de luz incide en la superficie y es reflejado directamente hacia la cámara; produciéndose una mayor intensidad en el punto A de la figura 4 que en el punto B, pues el reflejo de la luz del punto A incide directamente en la cámara, mientras que el reflejo del punto B se aleja de esta. Por esta propiedad es que produce resultados más intensos que el componente difuso y varía con mayor rapidez en dependencia de la dirección de la luz, el vector normal a la superficie, la posición de la cámara, el factor que describe el poder especular, entre otros.

El proceso de calcular el componente especular es más costoso y lento que el cálculo de la intensidad difusa y ambiental, pero adiciona un alto realismo a la escena; brindando apariencias metálicas, rugosas, húmedas, etc.

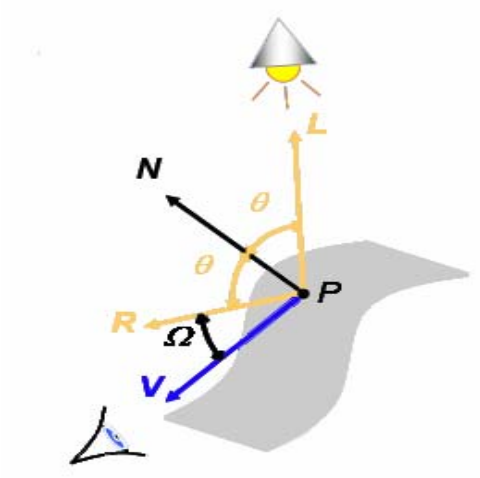

Figura 3: Reflexión especular de la luz.

1.2 Tipos de Luces

La iluminación de una escena, no sólo depende del modelo usado para calcular los componentes de la luz, sino también de las propiedades de los focos luminosos. En la práctica se destacan tres tipos de focos: los puntuales (*Point Lights*), los direccionales (*Directional Lights*) y los proyectores (*Spot Lights*). [14.1]

Focos Puntuales

Los focos puntuales son aquellos en los que la luz es emitida desde un sólo punto de la escena. Estos tienen color y posición, pero no una orientación específica; sino que la luz es emitida en todas las direcciones como se muestra en figura 5. Un ejemplo clásico de este tipo de luz son los bombillos, donde la luz parte del centro y es emitido en todas las direcciones.

Los focos puntuales presentan un rango de influencia. De forma que la intensidad de la luz es más débil a medida que la posición del foco se encuentre más lejana, siendo nula cuando esta sea mayor que el rango de influencia del foco.

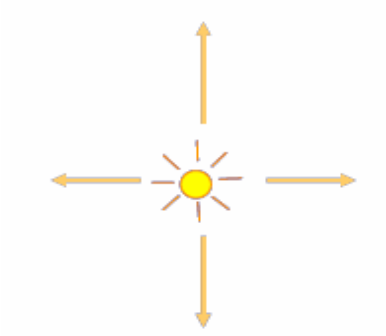

Figura 4: Fuente puntual de luz.

Focos Direccionales

A diferencia de los focos puntuales, los direccionales no presentan posición y emiten la luz de forma paralela. Esto significa que toda la luz emitida por estos focos se desplaza en la misma dirección y sentido como muestra la figura 6.

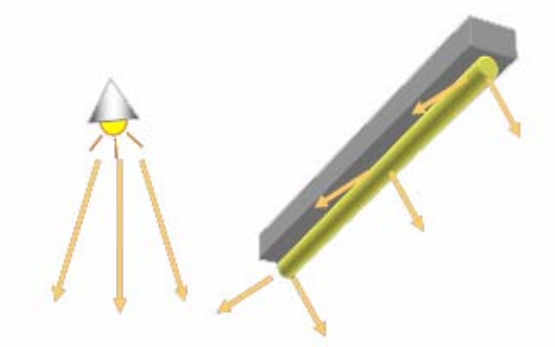

Figura 5: Focos direccionales

Los focos direccionales se pueden imaginar como emisores de luz situados a una distancia cercana al infinito o muy grande en comparación con la escena a representar. Un ejemplo de estas luces es el Sol, este se encuentra situado a una gran distancia en comparación con las dimensiones del planeta tierra y por tanto el planeta recibe la influencia de este en forma de rayos paralelos.

Debido al reducido número de factores que se tienen en cuenta para realizar el cálculo de la iluminación de los focos direccionales, se consideran como el tipo de luz menos costoso en cuanto a tiempo de procesamiento y recursos necesarios.

Proyectores

Los proyectores presentan color, posición y dirección en la que es emitida la luz. Siendo la región de influencia de estos focos en forma de cono según se muestra en la figura 7.

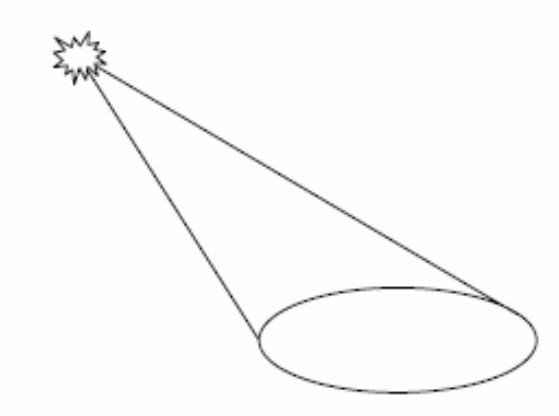

Figura 6: Proyector de luz

Estas luces son una de las más costosas en cuanto a la complejidad del cálculo, debido a la cantidad de factores que se han de tener en cuenta para representar estos focos. Entre estos factores se encuentran, el rango de influencia, el ángulo que forma el lateral del cono con su altura, la atenuación horizontal y la atenuación vertical, la distancia que viaja la luz antes de impactar la superficie, etc.

Un ejemplo de este tipo de luces son las linternas. Estas emiten la luz en forma de cono, siendo atenuada la intensidad de esta según los objetos iluminados se encuentren más alejados.

1.3 Shader

Gracias al avance en el hardware gráfico, el rendimiento de las aplicaciones de visualización se ha podido incrementar notablemente. Permitiendo representar escenas virtuales de mayor tamaño y complejidad. Sin embargo, aunque el aumento en el rendimiento ha posibilitado ejecutar las secuencias de visualización en un tiempo cada vez menor, se puede argumentar que al mismo tiempo estos sistemas no se han desarrollado en cuanto a tecnologías de visualización.

La limitante fundamental del hardware de aceleración gráfica es que su estructura no se puede cambiar. Cuando se crean estos hardwares, los ingenieros prefijan un conjunto de instrucciones y algoritmos en el chip de video. Cada uno de estos algoritmos es acelerado por el hardware gráfico. Pero no se brinda la posibilidad de ejecutar en estos chips otras instrucciones que no sean las codificadas en el momento de la creación del hardware. A esta estructura estática se le denomina sistema de funciones fijas (*Fixed-Function*). [13]

En los últimos 5 años se ha elaborado una alternativa al *Fixed-Function* para aumentar la flexibilidad gráfica del hardware de video. En dos momentos claves del *pipeline* gráfico se ha permitido introducir códigos que permitan ejecutar algoritmos no diseñados inicialmente en el hardware. A estos códigos se le llaman *shader*, y según su funcionamiento y lugar de ejecución se dividen en *Vertex Shader* y *Píxel Shader*.

Los ***Vertex Shader*** son los encargados de transformar todos los vértices de la escena. En ellos se ejecutan las transformaciones de espacio objeto a espacio de mundo, de cámara, y finalmente se obtiene la posición en la pantalla. Además, en ellos se incluyen todas las demás operaciones a nivel de vértices como son los cálculos procedurales de coordenadas de texturas, iluminación *per-vertex*, entre otras.

La figura 1 muestra como trabaja el pipeline gráfico donde la interface de *DirectX* u *OpenGL* pasa los *vertex* al driver, la GPU recibe estos datos y/o comandos y luego el *vertex shader* transforma los vértices recibidos y realiza otras operaciones a nivel de vértices, el interpolador (Rasterizador)

recibe la posición en la pantalla de cada uno de los vértices y genera los triángulos correspondientes. Al dibujar estos triángulos se calcula la posición de cada uno de los píxeles que conforman el triángulo.

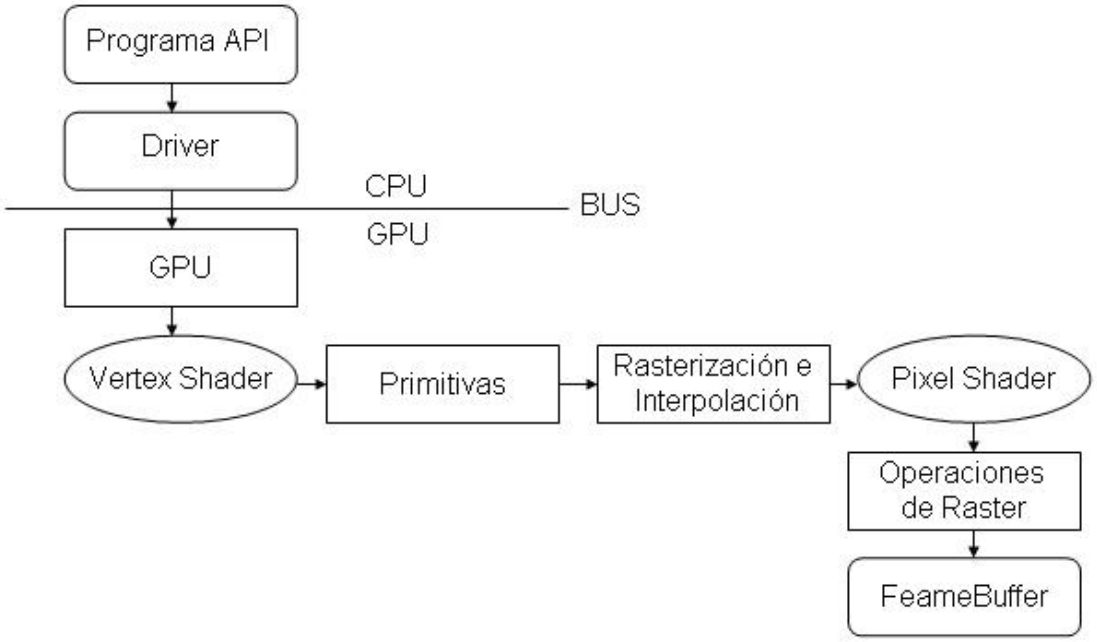

Figura 7: Pipeline Gráfico Conceptual usando Shaders

Cada uno de estos píxeles se introduce en el ***Píxel Shader*** para que este calcule el color del píxel en la pantalla. De esta forma en el *Píxel Shader* se realizan todas las operaciones a nivel de píxel como es el cálculo de la iluminación *per-píxel*, mapeo de texturas, uso de los mapas de normales para la determinación de la normal del píxel, etc.

De esta forma, el uso de los *shader* permite introducir nuevos algoritmos en el hardware de video. Dando la posibilidad de realizar avances en la visualización virtual sin la necesidad de esperar a que estos algoritmos sean codificados en el chip de video para obtener aceleración por hardware.

1.3.1 Unidad de Procesamiento Gráfico

GPU [8] es un acrónimo utilizado para abreviar *Graphics Processing Unit*, que quiere decir "Unidad de Procesado de Gráficos". Este se inventó como analogía a la sigla "CPU".

Básicamente, la GPU es una CPU dedicada exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo del procesador del ordenador en aplicaciones como los videojuegos, RV, etc. De esta forma, todo lo relacionado con los gráficos se procesa en la GPU y por tanto la CPU puede dedicarse a realizar otras operaciones.

El hardware gráfico actual es muy complejo; normalmente se utiliza la GPU para ofrecer múltiples canales de ejecución, con algo de Memoria de Acceso Aleatorio (VRAM) para los buffer y el rendereado de texturas y alguna instrucción en específico.

1.4 Soluciones de Iluminación

Contando con los modelos matemáticos para describir el comportamiento de la luz y el foco luminoso, es posible calcular la iluminación resultante en un objeto. Para esto se pueden tomar dos vías fundamentales: el cálculo de iluminación puede ser realizado para cada vértice del objeto o para cada píxel de este. [14.1]

1.4.1 Iluminación Per-Vertex

Cuando el cálculo se realiza en los vértices la iluminación se denomina "*Per Vertex Lighting*". Pues las ecuaciones que describen la contribución de la luz en el objeto son evaluadas en cada vértice de la geometría y luego para cada triángulo de ésta, se interpola el valor de la influencia en los vértices y se genera la contribución en cada píxel.

La figura 8 muestra un objeto al que se le ha iluminado usando esta técnica. Como se aprecia, la luz se nota un tanto distorsionada, pues la interpolación que ocurre para cada vértice produce gradientes de colores más oscuros o más claros de los que realmente describen la influencia de la luz.

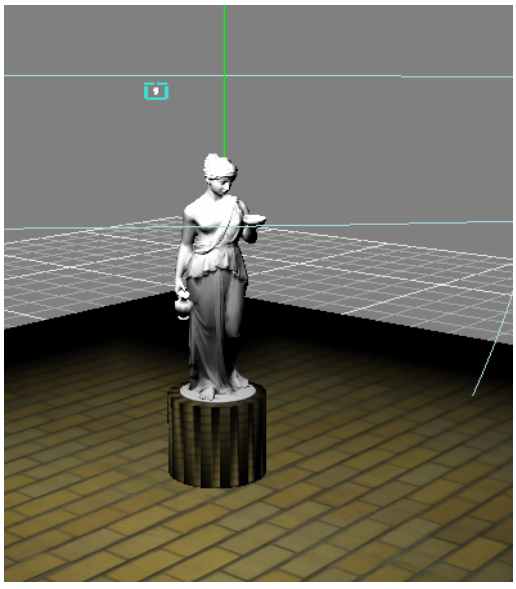

Figura 8: Iluminación "Per-Vertex"

Es de destacar que la solución *per vertex* tiende a mostrar resultados más precisos y con menos errores de aproximación a medida que los objetos iluminados son descritos con mayor cantidad de polígonos (presentan una mayor teselación).

Lo interesante de usar *per vertex lighting* es que el pipeline fijo contiene una implementación de esta, donde es posible calcular la contribución de hasta 8 luces en los vértices de un objeto. De ahí que las soluciones dirigidas al cálculo por vértices generalmente puedan ser ejecutadas en un mayor número de sistemas gráficos.

1.4.2 Iluminación Per-Píxel

Contrario a la iluminación "*per-vertex*", este método ejecuta las ecuaciones directamente en el píxel. De ahí que los resultados sean más precisos, pero el proceso total de iluminación tiende a ser más lento pues las ecuaciones son calculadas muchas veces por cada triángulo en lugar de tres veces (una por cada vértice) y luego interpolar el resultado para cada píxel.

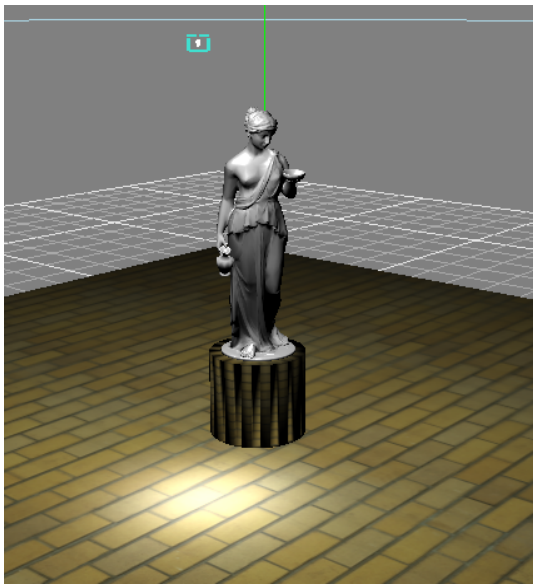

Figura 9: Iluminación "Per-Píxel"

La figura 9 muestra una escena iluminada *per-píxel*, como se aprecia en esta figura no aparecen errores de interpolación ni brillos en los bordes de los polígonos. Sin embargo, la iluminación "*per-píxel*" tiene el inconveniente que de manera estándar no se encuentra implementada en el *pipeline* fijo, de ahí que haya que usar los *shaders* para introducir esta funcionalidad en las tarjetas de video.

1.4.3 Mapas de Luces

De manera general la iluminación *per píxel* produce resultados de mayor calidad en comparación con la iluminación *per vertex*. Sin embargo, el cálculo de esta requiere mayores recursos y necesita evaluar la ecuación de iluminación muchas más veces que la en la iluminación *per vertex* de ahí que los tiempos de procesamiento de la luz sean mayores.

Cuando existen un número de luces y objetos estáticos, es posible precalcular la influencia de las luces en estos, siendo viable almacenar en una textura llamada mapa de luz (*Light MAP*) la contribución luminosa en la superficie y las sombras generadas por otros objetos en ésta.

El uso de mapa de luces reduce notablemente el número de cálculos a realizar para iluminar la escena pues ya se cuenta con el valor resultante de la iluminación en los píxeles de los objetos, por tanto no hace falta evaluar la ecuación de iluminación en cada píxel. Sin embargo se mantiene el carácter de iluminación per píxel.

1.5 Algoritmos de Iluminación y Sombreado

1.5.1 Algoritmos de Iluminación

A continuación se expone el estudio realizado que determina las técnicas y algoritmos que se utilizan actualmente, dadas las tendencias para el desarrollo a nivel mundial de módulos de luces similares al que se está concibiendo en este trabajo.

Iluminación de Hemisferios con Mapeado Bump

Este algoritmo logra simular una iluminación por irradiación del color. Consiste en iluminar las caras con una interpolación de dos colores, un color superior en la dirección de la luz y uno inferior contrario a la dirección de la luz, usando como interpolador el ángulo entre la normal del vértice y la dirección hasta la luz. El color superior debe representar el color aportado por la iluminación, y el color inferior debe representar el color reflejado del ambiente. Esto logra simular una iluminación por irradiación del color. (Figura 10).

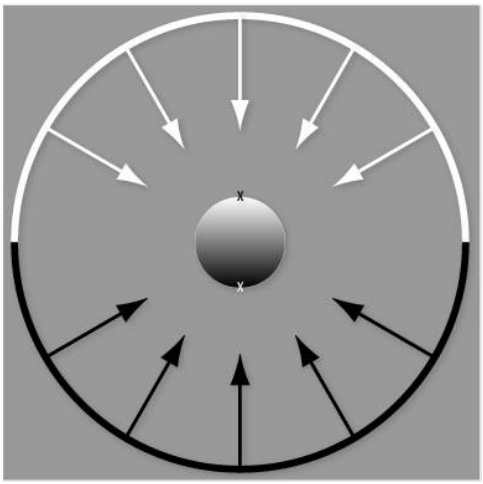

Figura 10: Iluminación de Hemisferios

Posteriormente aplica el mapeado *Bump*, el cual, utilizando una textura de normales, simula irregularidades en la superficie inexistentes en la geometría. Esto permite incrementar la resolución de detalles de un objeto sin añadir más vértices o caras a su estructura.

Finalmente aplica sombreado Phong para la reflexión especular. Esta representa el brillo de la luz que es reflejada por el objeto en zonas de máxima intensidad de iluminación.

El efecto requiere de 11 parámetros, las matrices de transformación a espacio de imagen y a espacio de mundo, la posición de la cámara y la dirección de la luz, ambas en coordenadas del mundo, los colores superior e inferior de la luz hemisférica, la potencia y la contribución especular al color final, las texturas de color, normales y detalles. Estas texturas se utilizan respectivamente para aportar los detalles de baja frecuencia, la perturbación de las normales y los detalles de alta frecuencia de la superficie del objeto. Además, es necesario que el objeto tenga calculadas por vértice la tangente y la binormal.

En el *vertex shader* se transforman los vértices –posición, tangente y binormal- a espacio de imagen, y se transforma la posición a espacio de mundo para enviarla al *píxel shader*. Se calcula la dirección del vértice hasta la cámara, las coordenadas de textura se propagan al *píxel shader* y se retorna.

En el *píxel shader* se leen los colores de las texturas en las posiciones correspondientes a las coordenadas especificadas, y se mezcla el color de baja frecuencia con el de alta, interpolados. La perturbación de las normales se decodifica del color de la textura de la siguiente forma: el color que se lee de la textura está en el rango [0, 1], como las normales tienen en las componentes tanto valores negativos como positivos, se tiene que la perturbación perpendicular o neutra, a la que le corresponde el vector {0, 0, 1} es el color {0.5, 0.5, 1}, así que se le aplica $C_N = 2 * C_T - 1$ para llevarlo al rango [-1, 1].

Con la normal, la tangente y la binormal calculadas por vértice, se construye una matriz que representa la transformación de espacio de textura a espacio tangente. Con esta matriz se transforma la perturbación en la normal decodificada de la textura.

Se calcula la iluminación hemisférica interpolando entre el color superior e inferior, utilizando como interpolador el producto escalar –normalizado en [0, 1]- entre la dirección de la luz inversa y la normal calculada por vértice. Este color de la luz se multiplica por el color de la superficie.

Si la normal en el píxel está de frente a la dirección de la luz, el píxel se ilumina especularmente. La intensidad especular proviene de la siguiente fórmula:

$$S = \max\left(reflect\left(V_D, N_P\right) \bullet -L_D, 0\right)^{S_P}$$

Donde:

VD: Dirección desde el píxel hasta la cámara.

NP: Normal perturbada.

LD: Dirección desde el píxel hasta la luz.

SP: Potencia especular.

max(a, b): Función intrínseca del GLSL. Retorna el mayor entre a y b.

reflect(r, n): Función intrínseca del GLSL. Retorna el vector resultante de reflejar el rayo de dirección r en una superficie de normal n.

El color especular se le suma al color de la superficie y se retorna.

Iluminación con Esferas Harmónicas

En 2001, *Ravi Ramamoorthi* y *Pat Hanrahan* presentan un método que usa esferas harmónicas para calcular la llamada iluminación difusa. Este método utiliza reflexión difusa, basada en el contenido de una imagen de luz sin acceder a ella en tiempo de procesamiento. La imagen de luz antes de ser procesada produce coeficientes que son usados en una representación matemática de la imagen en tiempo de procesamiento. El resultado es notablemente simple, certero y realista, y puede ser fácilmente codificado en un shader de OpenGL.

La iluminación con Esferas Harmónicas (*Spherical Harmonics Lighting*) proporcionan una representación de un espacio periódico de una imagen sobre una esfera. Esta representación es seguida a través de rotaciones invariantes. Usando esta representación por una imagen de luz, se puede reproducir exactamente la reflexión difusa desde una cara con solo nueve funciones básicas de esferas harmónicas. Éstas son obtenidas con constantes, lineales, y polinomios cuadráticos de la superficie normalizada.

Cada función básica de esferas harmónicas posee un coeficiente que depende de la imagen de luz que se esté usando. Los coeficientes son diferentes para cada color de canal, así se puede pensar en cada coeficiente con un valor RGB. Un paso del procesamiento por adelantado requiere calcular los nueves coeficientes RGB para que la imagen de luz sea usada.

$$\text{Difuso} = c_1 L_{22} (x^2 - y^2) + c_3 L_{20} z^2 + c_4 L_{20} - c_5 L_{20} + 2c_1 (L_{2-2} xy + L_{21} xz + L_{2-1} yz) + 2c_2 (L_{11} x + L_{1-1} y + L_{10} z$$

Ecuación 1: Reflexión difusa.

El *vertex shader* que codifica la fórmula para las nueve funciones básicas es actualmente bastante simple. Un compilador optimizado reduce todas las constantes involucradas en las operaciones. El resultado es bastante eficiente porque contiene un pequeño número de suma y operaciones de multiplicación que involucran los componentes de la normal del plano.

Así el *fragment shader* hace muy poco trabajo; porque típicamente la reflexión difusa cambia lentamente, para escenas de grande polígonos se calcula razonablemente el *vertex shader* e interpola durante el proceso de la imagen.

Iluminación Überlight

Hasta ahora en este epígrafe se han discutido algoritmos de iluminación que simulan el efecto global para efectos de iluminación más realistas. Puntos tradicionales, direccionales y reflectores pueden ser usados como punto de encuentro con este efecto de iluminación global. Sin embargo, la ausencia tradicional de fuente de luces parte en términos de su flexibilidad y facilidad de uso.

Controles Überlight

Para mejores resultados de iluminación de una escena, es crucial tomar la decisión correcta de formas y colocaciones de luces. Para el modelo de Iluminación Überlight (*ÜberLight Shader*), las luces son asignadas en una posición en las coordenadas del mundo. Este modelo usa un par de súper elipses para determinar la forma de la luz. Una súper elipse es una función que varía su forma desde una elipse a un rectángulo, basado en valores de parámetros redondeados.

La función de súper elipse es definida como:

$$\left(\frac{x}{a}\right)^{\frac{2}{d}} + \left(\frac{y}{b}\right)^{\frac{2}{d}} = 1$$

Ecuación 2: Super Elipse

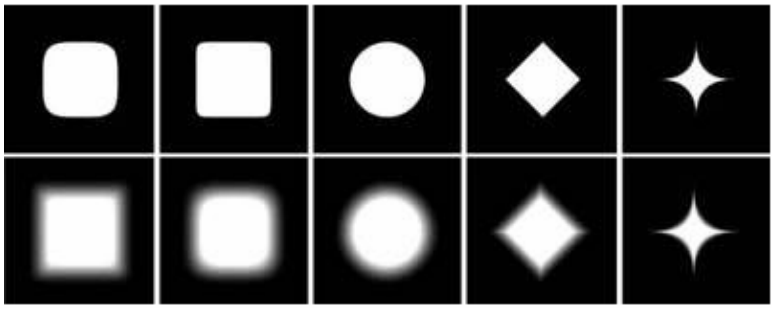

Figura 11: Überlight

Cuando el valor **d** se acerca a 0, esta ecuación se convierte en la ecuación del rectángulo, y cuando es iguala 1, la función se convierte en una elipse. Valores intermedios crean formas entre rectángulo y elipse y estas formas también son usadas para la iluminación.

Dos controles que adicionan la versatilidad de este modelo de iluminación son los parámetros de distancia cercanos y lejanos, conocidos también como valores *cuton* y *cutoff* (Figura 12). Este parámetro de distancia define la región de emisión que actualmente provee iluminación. Este control demuestra ser útil para suavizar la luz en la escena y previniendo la luz desde áreas llegadas donde la luz no es deseada.

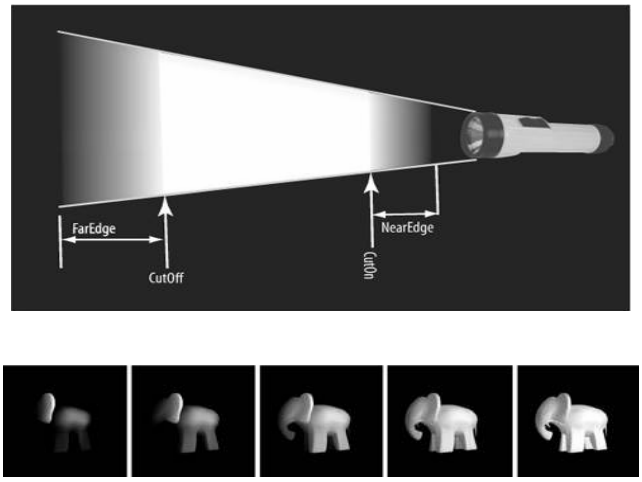

Figura 12: Überlight Sombras

El propósito principal del *vertex shader* es trasformar la posición del vértice, el plano normal y la posición de la cámara dentro del sistema de coordenadas de luces. En este sistema coordenado la luz están en el origen del sistema coordenado del mundo. Esto permite hacer los cálculos correspondientes más fácilmente, ejecutándose en el *fragment shader*. Los valores calculados son pasados al *fragment shader* a través de variables.

Al ejecutar estos cálculos, la aplicación debe suministrar un *ViewPosition*, la posición de la cámara en el mundo, y un *WCLightPos*, la posición de la fuente de luz en las coordenadas del mundo.

Para realizar las transformaciones necesarias, se necesitan matrices que transformen puntos desde las coordenadas modeladas a coordenadas del mundo y desde las coordenadas del mundo al sistema de coordenada de luz. Las matrices correspondientes para la transformación usual de algunos sistemas coordenados son matrices de transposición inversa. [9.1]

Una función en el *fragment shader* calcula el factor de atenuación a través de zonas de acción del emisor, otra calcula el factor de atenuación similar junto con la dirección del rayo de luz. Estos dos valores conjuntamente multiplicados dan el factor de iluminación para el *fragment*. [9.1]

El cálculo de reflexión de luz es similar a otros examinados anteriormente. Las normales calculadas pueden desnormalizarse mediante interpolación lineal: se pueden normalizar en los *fragment shader* y se obtienen resultados más exactos. Después que los factores de atenuación son calculados, se ejecuta un código de reflexión simple que da una apariencia plástica. Se pueden modificar estos cálculos para simular la reflexión desde algún otro tipo de material.

1.5.2 Algoritmos de sombreados

Oclusión Ambiental

La idea básica con este modelo es determinar, para cada punto en un objeto, cuanto de lo potencialmente visible en un hemisferio está visible y cuanto está oculto por algunas partes del objeto. El hemisferio considerado en cada punto de la superficie está en la dirección de la normal de la superficie en ese punto. Por ejemplo, considerando la clásica tetera de la figura 14, la parte superior de la agarradera de la tapa recibe iluminación desde un hemisferio completamente visible. Pero un punto un poco más abajo dentro del pico de la tetera recibe iluminación solo en un porcentaje pequeño del hemisferio visible, en la dirección de una pequeña abertura en el final del pico.

Figura 13: Oclusión Ambiental

Para un modelo específico se puede precalcular estos factores de oclusión y salvarlos como valores de atributos por vértice. Alternativamente se puede crear un mapa de textura que almacene valores para cada punto en el exterior. Un método para calcular factores de oclusión es unir la mayor cantidad de vértices posibles mediante líneas y darle un seguimiento a las intersecciones de otras partes del objeto y a las que no se intersectan. El por ciento de líneas similares que es desbloqueado es el factor de accesibilidad. La parte superior de la agarradera

de la tapa de la tetera tiene valor 1 desde entonces otra parte del modelo no es vista desde el hemisferio visible. El punto dentro del pico tiene un valor de accesibilidad de 0, porque su hemisferio visible está casi completamente en la oscuridad.

Entonces se multiplica el factor de accesibilidad por el valor de reflexión difuso. Esto tiene el efecto de oscurecer áreas que estaban tapadas por otras partes del modelo. Esto es lo bastante simple para usar estos valores en conjunto con otros modelos de iluminación. Por ejemplo, a la Iluminación de Hemisferios que se vio en la sección anterior se le puede incorporar oclusión ambiental con unos pocos cambios.

Al igual que a la Iluminación Basada en Imágenes y a la Iluminación con Esferas Harmónicas. En el caso anterior, la iluminación estaba hecha en el *fragment shader*, por lo tanto el factor de accesibilidad debe ser pasado al *fragment shader* como una variable de cambio. (Paralelamente, los valores de accesibilidad pueden ser almacenados en una textura que pueda ser accedida en el *fragment shader*).

Oclusión Ambiental (OA) es vista como una técnica independiente, pero los cálculos de los factores de oclusión que se le hace a un objeto son costosos. Si el objeto tiene partes moviéndose, los factores de oclusión deben ser recalculados para cada posición.

Mapeo de Sombras

La oclusión ambiental es muy eficiente para resaltar el realismo dirigido a objetos bajo condiciones de iluminación difusa, pero a menudo la escena necesita incorporar iluminación desde uno o más fuentes de luces bien definidos. En el mundo real, se sabe que una fuente de luz fuerte causa en los objetos una sombra a su semejanza. Produciendo similares sombras en la computadora, las escenas generadas tomarán mayor realismo.

Los lenguajes de alto nivel, HLSL y GLSL, incluyen facilidades para usar de manera general algoritmos de sombrado llamados Mapeo de Sombras (*Shadow Mapping*) (MS) [9.2], como se puede ver en el epígrafe “Lenguajes de programación”.

En este algoritmo, la escena es rendereada en múltiples ocasiones para cada luz que es capaz de dar sombra y una vez para generar la escena final, incluyendo la sombra. Cada paso de la luz es rendereado desde cada punto de luz. Los resultados son almacenados en una textura llamada Mapa de Sombra (*Shadow Map*) o Mapa de Profundidad (*Depth Map*). Esta textura esencialmente representa una superficie visible de un punto de luz. Estas superficies son completamente iluminadas por una fuente de luz, no son visibles desde un punto de luz que esté en la sombra. Cada textura generada es accedida en el último paso del proceso de render para crear la escena final con sombras desde más luces. [9.2]

Puesto que este algoritmo involucra un rendereado extra para cada fuente de luz, su rendimiento depende del número de luces que halla en la escena. Pero para aplicaciones interactivas, con dos luces adicionadas para el realismo y comprensibilidad de la escena es eficiente.

Si se adiciona más de dos luces puede darle complejidad a la escena y a la vez quitarle realismo. Como otros algoritmos que usan texturas, el mapeo de sombras está propenso a severos problemas de *aliasing* (líneas, especialmente las que están casi horizontales o verticales, que aparecen dentadas o irregulares debido a su representación por *píxeles*) a menos que se tome el cuidado apropiado.

La comparación entre aplicaciones interactivas puede también conducir a problemas. Desde que los valores son comparados son generados en distintos pasos con diferentes transformaciones matriciales, por lo tanto, debe usar un valor épsilon en la comparación. [9.2] Por ejemplo, en OpenGL se puede usar el comando glPolygonOffset para predefinir los valores de profundidad cuando el Mapa de Sombra es creado.

El camino para evitar problemas de precisión de profundidad con caras iluminadas es dibujar caras traseras cuando se esté construyendo un Mapa de Sombra. El valor de profundidad para estas superficies es normalmente bastante diferente desde el Mapa de Sombra.

A pesar de los inconvenientes, el MS es aun popular y efectivo en la generación de sombras.

OpenGL soporta Mapas de Sombras y un rango completo de modo de comparación de profundidad que puede ser usado con ellos. El Mapeo de Sombras puede ser ejecutado en OpenGL en una unidad de procesamiento programable. GLSL contiene funciones correspondientes para acceso a Mapas de Sombras dentro de un *shader*.

Deferred Shading para Sombras Volumétricas

Una de las desventajas del MS (mapeo de sombras) es que su funcionamiento depende del número de luces en la escena que sean capaces de brindar sombras. Con el MS, un paso de render debe ser ejecutado para cada fuente de luz. Estos Mapas de Sombras son utilizados en el último paso de render. Todos estos pasos de render pueden reducir el tiempo de la ejecución, particularmente si tienen un gran número de polígonos a ser rendereados. [9.2]

Existe una solución muy eficaz capaz de procesar un alto número de luces sin tener que enviar múltiples veces la geometría de la escena a la GPU: el *Deferred Shading* (DS). Con el DS la idea es determinar rápidamente la primera superficie que será visible en la escena final y aplicar efectos de *shader* complejos y que requieren de gran cantidad de tiempo solo para que los píxeles hagan esta superficie visible. En esta escena final, las operaciones de sombreado no se ejecutan hasta que pueda ser establecido cual píxel estará en la imagen final. Durante estos pasos iniciales, cualquier información almacenada es requerida para ejecutar las operaciones de render necesarias.

Las operaciones siguientes de render son solamente aplicadas a *píxeles* que están determinados para ser vistos en pasos iniciales de alto rendimiento. Esta técnica asegura que no se pierda tiempo en hacerle cálculos a *píxeles* que al final estarán ocultos.

Para renderear sombras suaves con esta técnica, se necesitan hacer dos pasos. [9.2]

En el primer paso se deben seguir estos procedimientos:

- Usar un *shader* para renderear la geometría de la escena sin sombras o iluminación dentro del frame buffer.
- Usar el mismo *shader* para almacenar los valores de profundidad de la cámara normalizada para cada píxel en un buffer separado. (esta separación es accedida como una textura en el segundo paso para el cálculo de sombras).

En el segundo paso, las sombras son calculadas con el contenido existente en el *frame buffer*. Para hacer estas operaciones de cálculo, se renderea el volumen de sombra para cada objeto con sombra. En el caso de una esfera, calcular el volumen de la esfera es relativamente fácil. La sombra de la esfera es en forma de un cono truncado, donde la cima del cono es la fuente de luz. Al final del cono truncado está el centro de la esfera (figura 16). Sin embargo es algo más complejo calcular el volumen de sombra para un objeto definido por polígonos, pero la aplicación es similar.

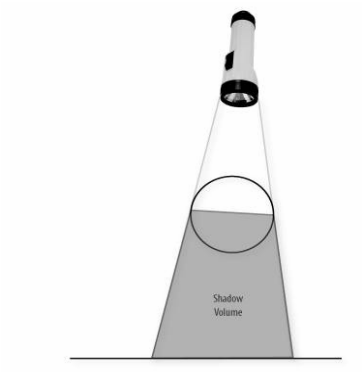

Figura 14: Sombra Volumétrica

Se puede mezclar sombras con geometrías existentes para el *render* de polígonos definido en las Sombras Volumétricas (SV). Esto permite aplicar el segundo paso sólo a regiones de imágenes que puedan estar en sombra.

Para dibujar una sombra, se usa el mapa de textura mostrado en la figura 15. Este mapa de textura expresa cuan visible es un punto de la superficie en relación a un objeto sombreado basado en una función de dos valores:

- La distancia al cuadrado desde el punto visible de la superficie hasta el eje central de la SV.
- La distancia desde el punto de la superficie visible al centro del objeto sombreado.

El primer valor es usado como coordenada **s** para acceder a la textura de sombra y el segundo valor es usado en la coordenada **t**. El resultado neto es que las sombras están relativamente sostenidas cuando el objeto sombreado esta muy cerca del fragmento examinado y los bordes llegan a ser suaves mientras aumenta la distancia.

Los *shader* requeridos son relativamente simples y rápidos. En vez de renderear la geometría una vez para cada fuente de luz, la geometría se hace solo una vez y todos los volúmenes de la sombra se pueden renderear en un solo paso de cálculo.

Los efectos localizados tales como Mapa de Sombra y texturas descriptivas se pueden lograr fácilmente. En vez de escribir el código para la figura a la cual el efecto se aplica, escribir el *shader* que se aplica a cada *píxel* y usarlo para renderear la geometría limitaría el efecto. Esta técnica se puede ampliar para renderear una diversa variedad de niebla, de iluminación y de productos cáusticos. [9.2]

1.6 Lenguajes de Programación

En esta sección del capítulo se analizarán dos de las bibliotecas gráficas que compiten en el mundo del desarrollo de gráficos 3D, y los lenguajes de programación a utilizar.

1.6.1 C++

El módulo se implementara en C++ debido a que la herramienta al que se acoplara este módulo esta implementada en dicho lenguaje; las principales características del C++ son el soporte para programación orientada a objetos y el soporte de plantillas o programación genérica (*templates*). Se puede decir que C++ es un lenguaje que abarca tres paradigmas de la programación: la programación estructurada, la programación genérica y la programación orientada a objetos.

Además posee una serie de propiedades difíciles de encontrar en otros lenguajes de alto nivel:

- Posibilidad de redefinir los operadores (sobrecarga de operadores)
- Identificación de tipos en tiempo de ejecución (*RTTI*)

C++ está considerado por muchos como el lenguaje más potente, debido a que permite trabajar tanto a alto como a bajo nivel, sin embargo es a su vez uno de los que menos automatismos trae (obliga a hacerlo casi todo manualmente al igual que C) lo que "dificulta" mucho su aprendizaje.

1.6.2 Librerías Gráficas: OpenGL y DirectX

A continuación se presentarán las características fundamentales de cada biblioteca:

OpenGL

OpenGL es una librería gráfica que provee a los programadores de una interfaz de acceso al *hardware* (HW) gráfico. Es poderoso, con *rendering* a bajo nivel y una librería de *software* de modelamiento, disponible en la mayoría de las plataformas, con un amplio soporte de HW. Es diseñado para ser usado en cualquier aplicación gráfica, desde juegos y simuladores hasta modelaciones CAD (*Computer Aided Design*). [1.1]

DirectX

"DirectX es un intento de Microsoft de brindar un acceso directo al HW en el entorno del sistema operativo Windows, a través de un conjunto de APIs que controlan un grupo de funciones que acceden al HW o lo simulan si no existe." [1.1]

Dichas funciones incluyen soporte para aceleración gráfica 2D y 3D, control de dispositivos de entrada, funciones para mezclar y probar sonidos y música, control para juegos en red y *multiplayer*, y control sobre varios formatos de *streaming* de multimedia (*streaming* es un método de transferencia de datos continuamente, que permite mostrar los datos antes de que el fichero entero haya sido transmitido). [1.1]

Ambas APIs usan la tradicional *graphics pipeline* (tubería gráfica). Es el mismo *pipeline* que se diseñó desde las primeras computadoras gráficas, que se ha ido modificando de acuerdo a los avances de HW aunque sin cambiar la idea básica. [1.1]

Ambas APIs describen los vértices como un grupo de datos consistente en coordenadas en el espacio que definen la localización del vértice. Las primitivas gráficas (puntos, líneas, y triángulos) están definidos como un grupo ordenado de vértices. Sin embargo, la diferencia entre

las APIs está en cómo los vértices son combinados para formar las primitivas: cada uno lo maneja diferente. [1.1]

1.6.3 OpenGL Shading Language

OpenGL Shading Language (GLSL) es un lenguaje de alto nivel diseñado específicamente por el entorno *OpenGL*. Este lenguaje permite aplicaciones para hardware gráfico. Contiene funciones que permiten expresiones abreviadas de algoritmos gráficos de manera que sea natural para programadores con experiencia en C y C++.

GLSL incluye tipos escalares, vectores y matrices; estructuras y arreglos; tipos de muestras para acceder a texturas; un tipo de dato que define entradas y salidas; constructores para inicialización y conversión; y operadores y controles declarados justo como en C y C++.

Ofrece opciones avanzadas en el diseño de gráficos al proporcionar acceso de alto nivel a las características programables de los procesadores de gráficos modernos, lo que representa un gran paso adelante en la creación de gráficos 3D fotorrealistas en tiempo real.

Posee implementaciones en UNIX, Windows, Linux y otros sistemas operativos. Esta amplia compatibilidad permite a los desarrolladores mover fácilmente sus trabajos entre los principales sistemas operativos comerciales y las plataformas hardware.

Entre sus principales características se encuentran:

- Posibilidad de crear sombreados asociados al aspecto (fragmentos) de la geometría (vértices) de un objeto 3D.
- De una sola vez pueden aplicarse sombreados sobre diferentes renderizados y generar distintos resultados que se almacenan en buffer.

- La aplicación de texturas no está condicionada por su tamaño que, a diferencia de lo que ocurría en el pasado, no tiene por qué ser potencia de dos. De esta forma ahora se soportan texturas rectangulares y se reduce el consumo de memoria.
- Se pueden aplicar patrones (*stencil*) sobre las dos caras de las primitivas geométricas, mejorando el rendimiento en el volumen sombreado y en los algoritmos de renderizado de geometría sólida.

1.6.4 High Level Shading Language

High Level Shading Language (HLSL) está desarrollado en base a las funciones de C y C++. Este lenguaje tiene muchas características de estos lenguajes estándares, tales como funciones, expresiones, declaraciones, tipos de datos estándares, tipos de datos creados por el usuario e instrucciones para el procesador.

HLSL soporta instrucciones para escribir expresiones matemáticas. Como otros lenguajes gráficos, las expresiones matemáticas son más eficientes con vectores y matrices. Este lenguaje sigue muchas reglas e instrucciones de C, así como otras propias para hacer la programación gráfica más intuitiva y compacta.

Conclusiones

En el transcurso de este capítulo, como base para el entendimiento del tema en que se desenvolverá este proyecto, se formularon una serie de conceptos básicos para el posterior entendimiento de lo expuesto, tales como Iluminación *Per-Vertex* y *Per-Píxel*, GPU. Además se mostraron las técnicas y tendencias actuales más utilizadas para el desarrollo de luces y sombras dinámicas en SRV, sus características, sus ventajas y desventajas.

Se trataron también las principales características de las librerías gráficas a usar, así como algunas particularidades de los dos lenguajes específicos que serán utilizados para el desarrollo de este trabajo.

Capítulo 2 Descripción de la Solución Propuesta

Introducción

En este capítulo se propone cual de los algoritmos mencionados anteriormente es el más apropiado según los objetivos planteados en el trabajo; algoritmos y técnicas para reducir los efectos secundarios del uso de algunos algoritmos existentes como son el alto consumo de memoria y el *fill-rate*. Se da a conocer como podría ser la estructura del módulo a partir de la definición de algunas clases.

2.1 Algoritmos de Iluminación

En el capítulo anterior se hizo referencia a los algoritmos más utilizados actualmente para la iluminación dinámica en visualizaciones gráficas. En este epígrafe se dará a conocer, por las características que posee y las funcionalidades que brinda, el algoritmo que más se adapta a las necesidades planteadas en la introducción de este trabajo: Iluminación con Esferas Harmónicas (EH) (*Spherical Harmonic Lighting*).

EH es eficiente debido a la simplicidad del algoritmo, ofrece resultados realistas y no exige un alto consumo de memoria en comparación con la Iluminación de Hemisferios; a pesar de que la Iluminación *Überlight* es similar a esta se decidió no utilizarla ya que para determinar la forma de la luz usa un par de súper elipses y el trabajo con estas puede resultar engorroso. Debido a estas características es la técnica de iluminación que más se ajusta para dar solución al problema planteado en el capítulo anterior en cuanto al tema de iluminación dinámica.

2.2 Algoritmos de Sombreado

En el capítulo anterior se hizo referencia a los algoritmos más utilizados actualmente para la iluminación dinámica en visualizaciones gráficas. En este epígrafe se dará a conocer, por las características que posee y las funcionalidades que brinda, el algoritmo que más se adapta a las necesidades planteadas en la introducción de este trabajo: Mapeo de Sombras (MS) (*Shadows Mapping*).

Esta técnica es una de las más usadas en la actualidad para aplicaciones gráficas gracias a las características que posee, con muchas fuente de luz puede resultar costoso el proceso de render pero con una o dos no retarda la visualización y los resultados son muy realistas; las otras técnicas también mencionadas se usan mucho, en el caso específico de Oclusión Ambiental los cálculos de los factores de oclusión que se le hace a un objeto son costosos, si el objeto tiene partes moviéndose, los factores de oclusión deben ser recalculados para cada posición lo que implica un retardo considerable en la visualización. Con *Deferred Shading* para sombras volumétricas la desventaja es que para generar la sombra se necesitan hacer dos pasos donde cada uno presenta un número considerable de cálculos complejos mientras que MS para una o dos fuentes de luz no presentan tanta complejidad. Debido a esto MS es la técnica que más se ajusta para dar respuesta a los objetivos propuestos en el capítulo anterior en cuanto a la generación de sombras dinámicas eficientes.

2.3 Lenguaje de Programación

GLSL es un lenguaje de alto nivel diseñado específicamente por el entorno OpenGL. Este lenguaje permite aplicaciones para hardware gráfico. Contiene funciones que permiten expresiones abreviadas de algoritmos gráficos de manera que sea natural para programadores con experiencia en C y C++.

Ofrece opciones avanzadas en el diseño de gráficos al proporcionar acceso de alto nivel a las características programables de los procesadores de gráficos modernos, lo que representa un gran paso adelante en la creación de gráficos 3D fotos realistas en tiempo real.

Aunque GLSL y HLSL son muy parecidos en cuanto a estructura y funcionamiento, como el módulo solo será implementado utilizando la biblioteca gráfica *OpenGL,* GLSL es el lenguaje con el que se implementará el *shader* a utilizar (ver epígrafe 1.6).

Conclusiones

Con este capítulo quedan sentadas las bases técnicas por las que se regirá este módulo. De aquí, se desprenden datos que deben contener los objetos para la realización de los algoritmos, y por tanto, partiendo de ellos, se diseñará una estructura de clases en correspondencia con las técnicas citadas y para lograr los objetivos del proyecto.

Capítulo 3 Construcción de la Solución Propuesta

Introducción

En este capítulo se comienza a tener la visión del sistema a desarrollar. Aquí se inicia la concepción práctica del producto a elaborar, sobre la base de las dificultades, necesidades y características del cliente, conociendo ya las técnicas y la estructura a utilizar descritas en el capítulo anterior.

3.1 Objeto de estudio

El **objeto de estudio** de este trabajo es dar solución a uno de los principales problemas que presenta el proyecto Herramientas de Desarrollo para Sistemas de Realidad Virtual, la inexistencia de un módulo generador de luces dinámicas; que sea, además, eficiente en cuanto a velocidad de *render*, al realismo de la escena, que utilice adecuadamente las potencialidades de las tarjetas gráficas y que esté dirigida a la biblioteca gráfica *OpenGL*.

En el capítulo anterior se hizo referencia, guiados por el estudio echo, al algoritmo que se considera como el candidato para lograr los efectos de sombras deseados, no se le dará implementación pero la estructura queda diseñada de tal manera que su posterior incorporación se puede realizar de la misma manera que con la iluminación.

Obtener un módulo capaz de generar luces dinámicas es el **campo de acción** de este trabajo. Actualmente el proyecto cuenta con un pequeño módulo de iluminación estática, el cual no brinda gran realismo, además de retardar la visualización debido a la utilización de técnicas ineficientes.

Los objetivos propuestos para este trabajo y que derivan en la obtención de un módulo de iluminación dinámica y eficiente que sustituya al actual son los siguientes:

- El empleo de algoritmos de iluminación que proveen a la escena virtual de gran realismo.
- La incorporación de técnicas eficientes para eliminar lentitud en las visualizaciones.
- El soporte de una de las dos bibliotecas gráficas mencionadas en el capítulo 1, respondiendo así a las tendencias que existen en el mundo.

3.2 Reglas del Negocio

El fichero *shader* a cargar para la iluminación y el sombreado deben ser de extensión *VERT* y *FRAG* para ambos casos, de no existir el fichero en el camino indicado, ser de otro formato, estar corrupto o estar vacío, la aplicación no se detiene, pero no se ejecuta ningún código *shader* por lo que no se obtiene el resultado esperado.

3.3 Modelo del Dominio

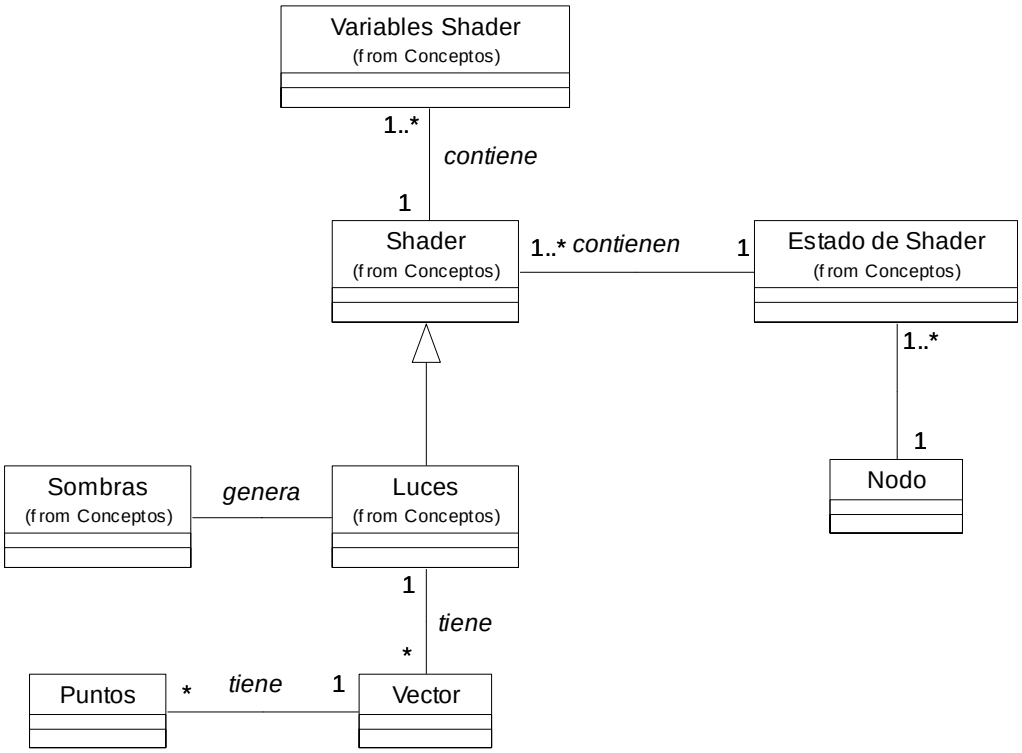

Figura 15: Modelo Del Dominio.

3.3.1 Glosario de términos del modelo del dominio

Estado de shader: encargado de almacenar lo que devuelve **Shader**, maneja las variables controladoras a través de una lista que posee. Define el estado en que se encuentra el shader cargado. Puede tener varios shader asociados a la vez.

Luces: estructura encargada de almacenar la posición, vector director, dirección y demás parámetros que posee una fuente de luz.

Shader: encargado de manejar los ficheros con el código a interpretar y almacenar para su posterior uso.

Sombras: estructura encargada de generar sombra a partir de una serie de estructuras **luces** relacionadas. Calcula, para cada fuente de luz, como debe quedar el sombreado.

Variables shader: estructura encargada de interpretar las variables utilizadas en el fichero con el código ***glsl*** y enlazarlas con el programa.

3.4 Captura de Requisitos

A continuación se expondrán los requisitos funcionales y no funcionales del sistema.

3.4.1 Requisitos Funcionales

1. Crear luces
2. Modificar luces
3. Generar sombras
4. Crear variables controladoras
5. Modificar variables controladoras
6. Crear estado de *shader*
7. Modificar estado de *shader*
8. Crear *shader*
9. Modificar *shader*
10. Cargar *shader*
11. Atachar datos del fichero al *shader*
12. Atachar *shader* al estado
13. Atachar variable *shader* a variable controladora
14. Adicionar variable controladora al estado de *shader*
    14.1. Adicionar variable a la lista de variable del estado.
15. Actualizar *shader*.

3.4.2 Requisitos no Funcionales

- **Usabilidad:** Los futuros usuarios del sistema serán programadores con conocimientos básicos de programación gráfica y de la terminología afín. El producto debe estar concebido para que el usuario piense en qué desea hacer y no cómo hacerlo, por lo que éste requerimiento debe estar presente en alto grado en el producto final.
- **Rendimiento:** Como aplicación de tiempo real, debe tener alto grado de velocidad de procesamiento o cálculo, tiempo de respuesta y de recuperación, y disponibilidad.
- **Soporte:** En una versión inicial deberá ser compatible con la plataforma Windows, pero debe estar preparado para que con rápidas modificaciones pueda migrar para Linux.
- **Hardware:** Compatibilidad con tarjetas gráficas de la familia NVIDIA (Quadro FX 500/FX 600).
- **Diseño e implementación**: Debe utilizar transparentemente la biblioteca gráfica OpenGL y DirectX, en su primera versión, y ser adaptable a trabajar con otras bibliotecas. Se harán llamadas a dichas bibliotecas desde C++ y se usara el GLSL (*OpenGL Shading Language)* para el manejo de los *shader*. Se regirá por la filosofía de Programación Orientada a Objetos.

3.5 Modelo de Casos de Usos del Sistema

En esta sección se reconocen los posibles actores del sistema a desarrollar y se conciben, a través de la agrupación de los requisitos funcionales anteriormente hallados, los posibles resultados de valor que le pueda brindar a sus actores, o lo que es lo mismo, los casos de uso del sistema.

Además, se seleccionan los casos de uso correspondientes al primer ciclo de desarrollo para hacerles sus especificaciones textuales en formato expandido.

Actor del sistema

Actores	**Justificación**
Programador	Es el que se beneficiará con las funcionalidades que brinda el módulo de clases, a grosso modo: cargar desde ficheros, establecer estado de *shader* y ejecutar el ciclo de la escena.

Tabla 1: Actor Del Sistema.

Casos de uso del sistema

1. Adicionar luz
2. Actualizar luz
3. Cargar *light-shadow shader*

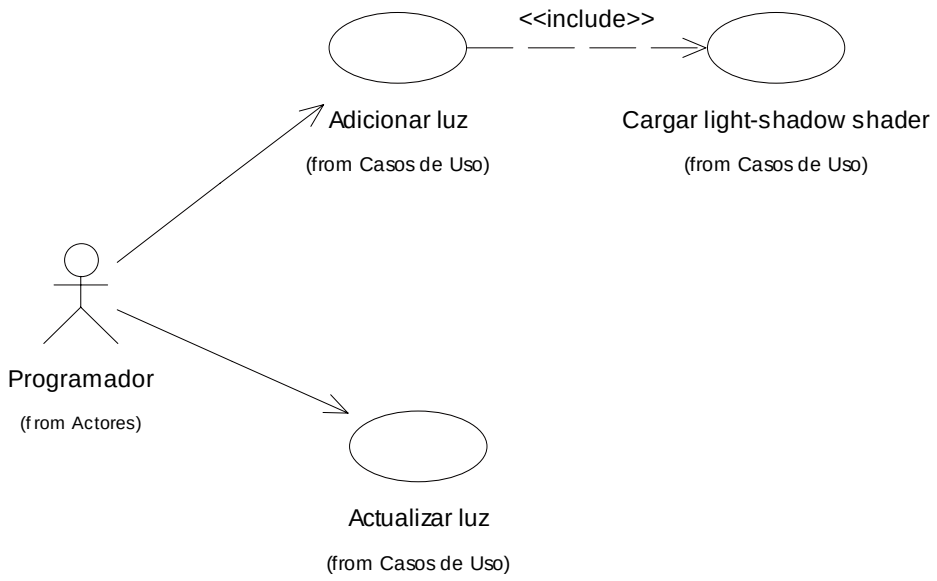

Figura 16: Diagrama de CU

El Programador inicia los CU Adicionar luz y Actualizar luz. El CU Adicionar luz se encarga de crear un objeto luz, crea un estado de *shader*, llama al CU incluido Cargar *light-shadow shader* y se le asigna el estado al nodo de la escena. El CU Cargar *light-shadow shader* carga el algoritmo con la técnica a utilizar para la iluminación y el sombreado de la escena. El CU Actualizar luz es el encargado de tomar los valores globales de la escena y pasárselo a las variables controladoras del shader.

Especificación de los casos de uso en formato expandido

Nombre del caso de uso	**Cargar light-shadow shader**
Actores	Programador
Propósito	Interpretar el *buffer* de datos del fichero y almacenarlos en las estructuras de datos de los shader.
Resumen: Se inicia cuando el Programador pasa una dirección de fichero. De ser válida la dirección se llenan los *buffers*. Se crean las estructuras necesarias para almacenar el *shader*, y se le pasa al estado de *shader*. El caso de uso finaliza cuando el estado de *shader* es actualizado con el nuevo *shader*.	
Referencias	R14. CU3.
Curso normal de los eventos:	
Acción del actor	Respuesta del sistema
1- Solicita cargar el *shader* pasando como parámetro la dirección del fichero.	
	2- Se cargan las variables a utilizar en el *vertex program*.
	3- Se normaliza la normal.
	4- Se obtienen las coordenadas de la textura activa.
	5- Se calcula el color difuso a partir de la multiplicación de los coeficientes de los valores RGB de la luz activa.
	6- Se actualiza el color difuso con la multiplicación del factor de escala, el cual va a dar la intensidad de la luz
	7- Se cargan las variables a utilizar en el *fragment program*.

	8- Se asocia el *Sampler2D* del *shader* con las coordenadas de texturas activas capturadas anteriormente.
	9- Se interpola el color difuso para cada *píxel* de la escena.
Poscondiciones	Nuevo *shader* cargado.

Tabla 2: CU Cargar light- shadow shader

Nombre del caso de uso	**Adicionar luces**	
Actores	Programador	
Propósito	Adicionar nueva estructura encargada de almacenar e interpretar los *shader*.	
Resumen: Se inicia cuando el Programador solicita la creación de una nueva fuente de luz. Se crea un estado de *shader*. Se inicializan los parámetros preparándolos para la posterior adición de los *shader.* Se llama al CU "Cargar *light – shadow shader*". Se asigna el estado a un nodo.		
Referencias	R1, R4, R8. CU1.	
Curso normal de los eventos:		
Acción del actor		Respuesta del sistema
1- Solicita la adición de una nueva fuente de luz.		
		2- Se crea el objeto luz.
		3- Se crea el estado.
		4- Se crea la lista de variables controladoras que manejara el estado.
		5- Se inicializan los parámetros.
		6- Se llama al CU "Cargar *light - shadow shader*"
		7- Se adicionan las variables controladoras a la lista de variables del estado.
		8- Se le asigna estado al nodo.
Poscondiciones	Nuevo estado asignado al nodo de la escena.	

Tabla 3: CU Adicionar luces

Nombre del caso de uso	**Actualizar luces**
Actores	Programador
Propósito	Definir que valores tendrán cada una de estas variables para ejecutar el *shader* y lograr el resultado deseado.
Resumen: Se inicia cuando el Programador solicita actualizar el *shader*. Para ello asigna valores a las variables controladoras según la función que estas cumplan. El CU concluye cuando todas las variables son actualizadas. En caso de algún error, o de no asignar ningún valor a las variables, no se detiene la aplicación, pero no se obtiene el resultado deseado.	
Referencias	R2, R20. CU2.
Curso normal de los eventos:	
Acción del actor	Respuesta del sistema
1- Solicita la actualización de las luces de la escena.	
	2- Se guarda el valor de la matriz de modelado de la escena.
	3- Se guarda el valor de la matriz de proyección de la escena.
	4- Se le asigna el valor de la matriz de modelado a la variable controladora de esta matriz para el *shader*.
	5- Se le asigna el valor de la matriz de proyección a la variable controladora de esta matriz para el *shader*.
	6- Se le asigna la posición de la luz del nodo de la escena a la variable controladora para el *shader*.
	7- Se le asigna color de la luz del nodo de la escena a la variable controladora para el

	shader.
	8- Se le asignan rangos del mapa de profundidad de sombra según las texturas utilizadas.
Poscondiciones	Luces actualizadas.

Tabla 4: CU Actualizar luces

Conclusiones

En el presente capítulo se definió qué es exactamente lo que espera el usuario con este módulo. Para ello quedaron establecidos los requisitos funcionales de éste, y descritos los casos de uso que le permitirán a su futuro usuario obtener los resultados esperados por el cliente.

Capítulo 4 Diseño del Sistema

Introducción

La primera parte del capítulo encierra los diagramas de clases de análisis del sistema propuesto. Su objetivo es brindar una primera visión de las posibles clases del diseño a un alto nivel, pero donde se dejan ver sus responsabilidades y relaciones, aún sin el mayor rigor entre ellas.

A continuación se presentan los diagramas de clases como resultado del refinamiento de las etapas anteriores. Se presentan además los diagramas de secuencia de la realización de los CU.

4.1 Diagrama de clases del Análisis

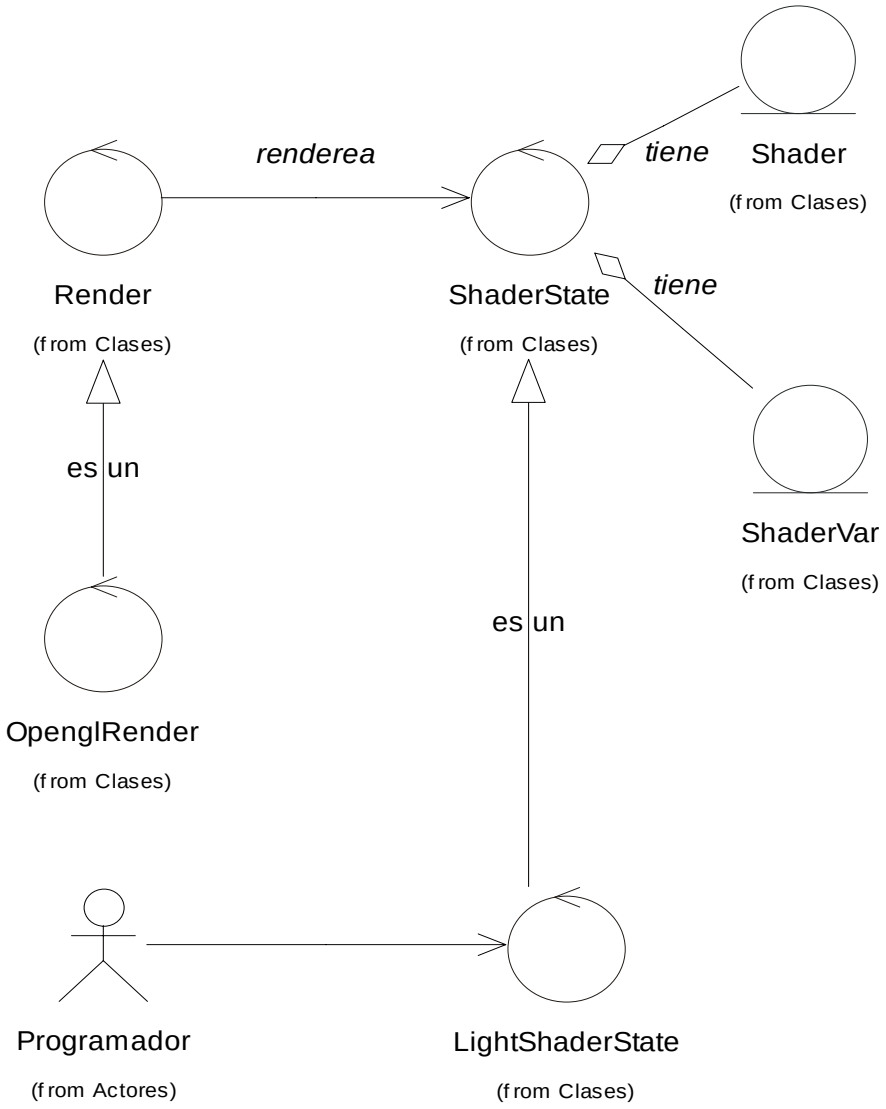

Figura 17: Diagrama de clases del Análisis

4.2 Diagrama de clases de Diseño

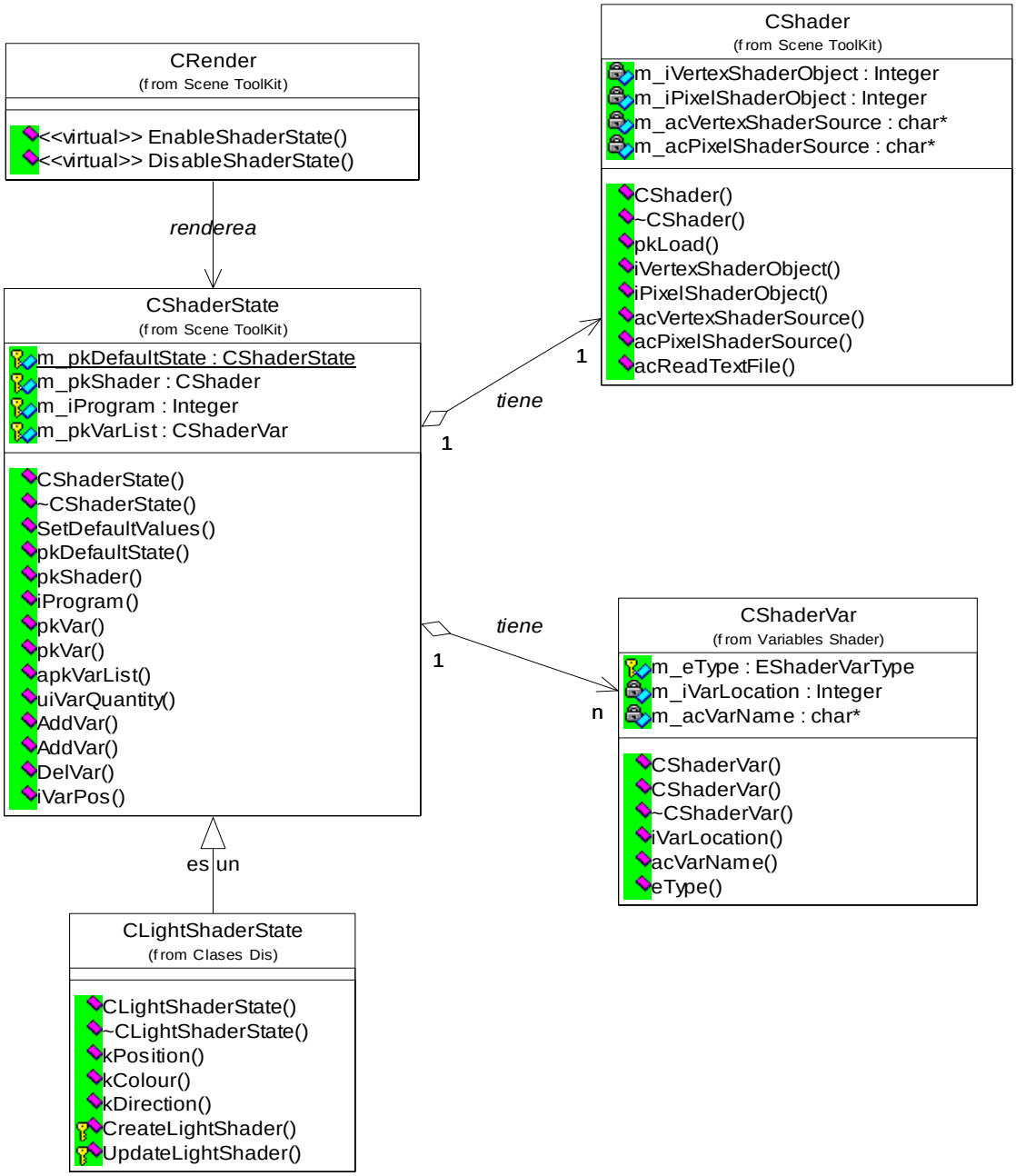

Figura 18: Diagrama de clases de diseño

Por la complejidad del Diagrama de clases de diseño y para su mejor entendimiento se han agrupado algunas clases en paquetes; el paquete Scene ToolKit encapsula las clases CRender y COpenGlRender de la herramienta y el paquete Variables *Shaders* que encapsula las clases encargadas de manejar los tipos de variables. Las clases CShader, CShaderState y CShaderVar son clases que fueron agregadas a la herramienta para el manejo de los *shader*.

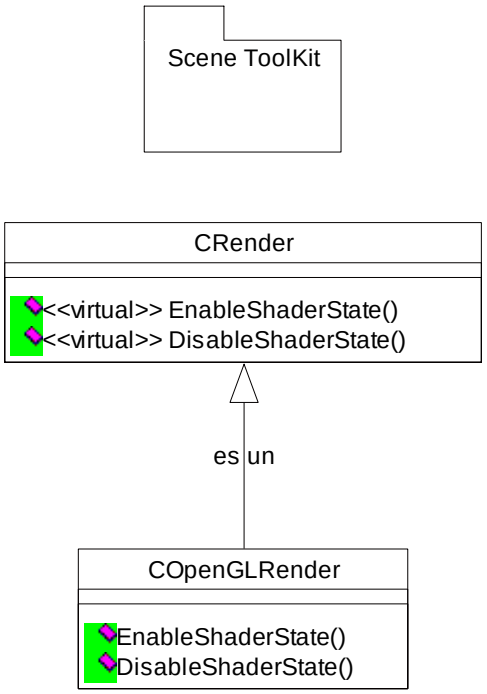

Figura 19: Diagrama de clases del paquete Scene ToolKit

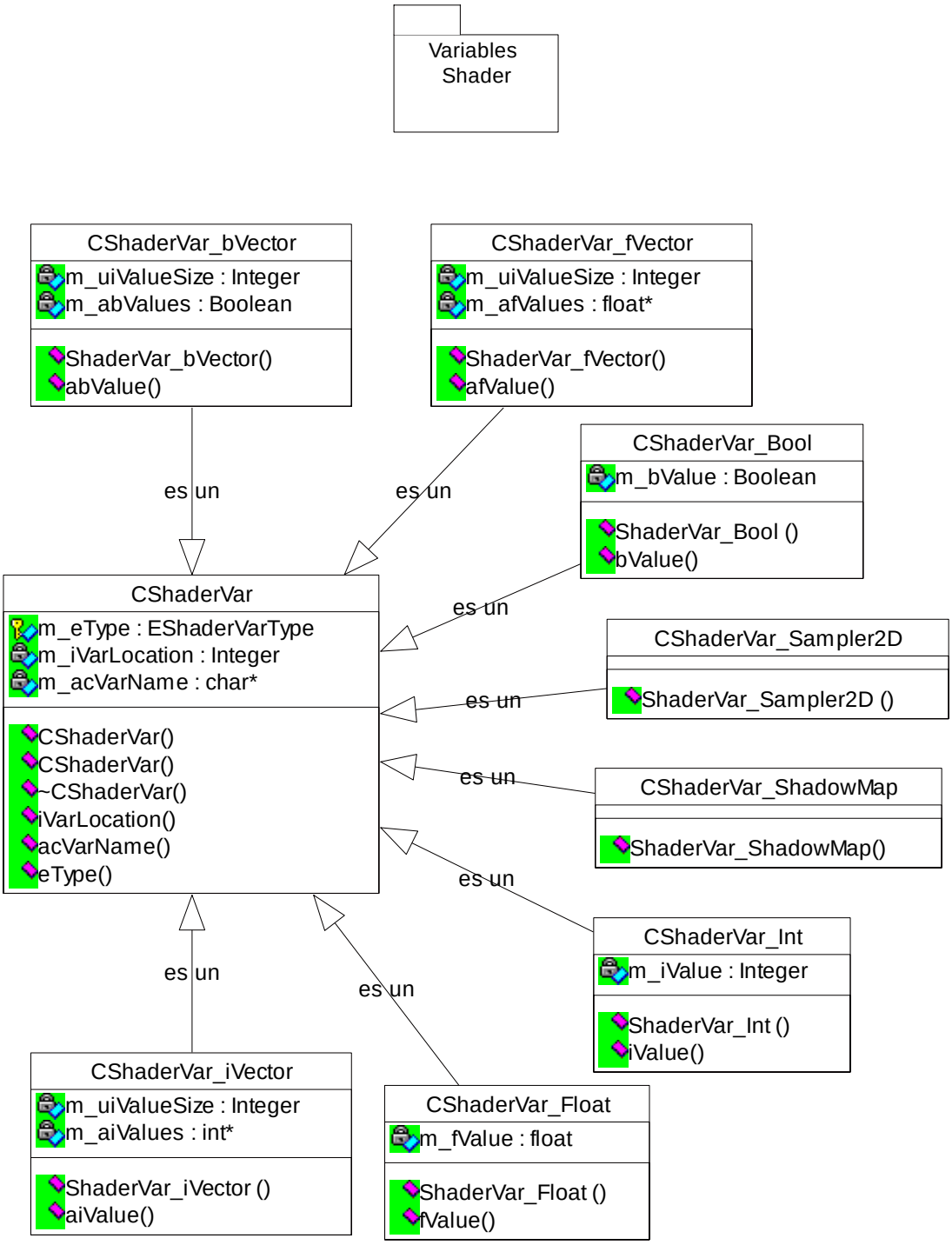

Figura 20: Diagrama de clases del paquete Variables Shader

4.2.1 Descripción de las clases del Diseño

Nombre: CLightShaderState	
Tipo de clase: Controladora	
Atributo	**Tipo**
pkLightState	CShadeState*
pkBumpMap	CShaderVar_Sampler2D*
pkScaleFactor	CShaderVar_Float*
Para cada responsabilidad:	
Nombre:	CreateLightShaders
Descripción:	Crea un estado de *shader*, carga un fichero *shader* y se lo asigna a ese estado, asocia las variables creadas con las variables del código *shader*, según corresponda y se adicionan a la lista de variables que tiene el estado
Nombre:	UpdateLightShaders
Descripción:	Actualiza el valor del factor de escala que modifica la intensidad de la luz.

Tabla 5: Descripción de la clase CLightShaderState

Nombre: CShaderState	
Tipo de clase: Controladora	
Atributo	**Tipo**
m_pkDefaultState	static CShadeState*
m_pkShader	CShader*
m_iProgram	int
m_apkVarList	CShaderVar*
Para cada responsabilidad:	
Nombre:	AddVar
Descripción:	Función que se encarga, pasándole el nombre de la variable y el tipo previamente definido, de llenar la lista de variables del estado.
Nombre:	pkVar
Descripción:	Función que se encarga dado el nombre de una variable retornar la variable.
Nombre:	pkDefaultState
Descripción:	Función que se encarga de inicializar los atributos antes de ser usados poniéndole valores por defecto.
Nombre:	pkShader
Descripción:	Función que se encarga de almacenar el shader asociado al estado.
Nombre:	iProgram
Descripción:	Función que se encarga de almacenar el programa asociado al estado, este es el identificador del código *shader* cargado anteriormente.

Tabla 6: Descripción de la clase CShaderState.

Nombre: CShader	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_iVertexShaderObject	int
m_iPixelShaderObject	int
m_acVertexShaderSource	char*
m_acPixelShaderSource	char*
Para cada responsabilidad:	
Nombre:	pkLoad
Descripción:	Función que se encarga de cargar los ficheros *.vert* y *.frag* que contienen el código *shader* a ejecutar.
Nombre:	acVertexShaderSource
Descripción:	Función que se encarga de almacenar el código del fichero *.vert*
Nombre:	acPixelShaderSource
Descripción:	Función que se encarga de almacenar el código del fichero *.frag*

Tabla 7: Descripción de la clase CShader.

Nombre: CShaderVar	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_eType	EShaderVarType
m_iVarLocation	int
m_acVarName	char*
Para cada responsabilidad:	
Nombre:	iVarLocation
Descripción:	Función encargada de almacenar la localización de la variable dentro del código *shader.*
Nombre:	acVarName
Descripción:	Función encargada de almacenar el nombre de la variable.
Nombre:	eType
Descripción:	Función encargada de almacenar el tipo de la variable según los tipos definidos previamente.

Tabla 8: Descripción de la clase CShaderVar.

Nombre: COpenGLRenderer	
Tipo de clase: Controladora	
Atributo	**Tipo**
Para cada responsabilidad:	
Nombre:	EnableShadeState
Descripción:	Función encargada de habilitar el estado de *shader*, después de compilar el programa para cada variable del estado se guarda su localizacion dentro del programa activo y posteriormente se le asigna los valores según el valor que tenga en la lista de variables.
Nombre:	bCompile
Descripción:	Función que a través de OpenGL y pasándole el estado de *shader*, crea el programa *shader*, crea el *vertex*, crea el *fragment* y para cada uno asocia el código, lo compila y lo atacha al programa y al final lo linkea.

Tabla 9: Descripción de la clase COpenGLRenderer.

4.3 Diagramas de Secuencia

A continuación se presentan los diagramas de secuencia correspondiente a cada caso de uso planteado en el capítulo anterior. Con ellos se puede tener una idea del tiempo de ejecución de cada algoritmo que interviene en los CU.

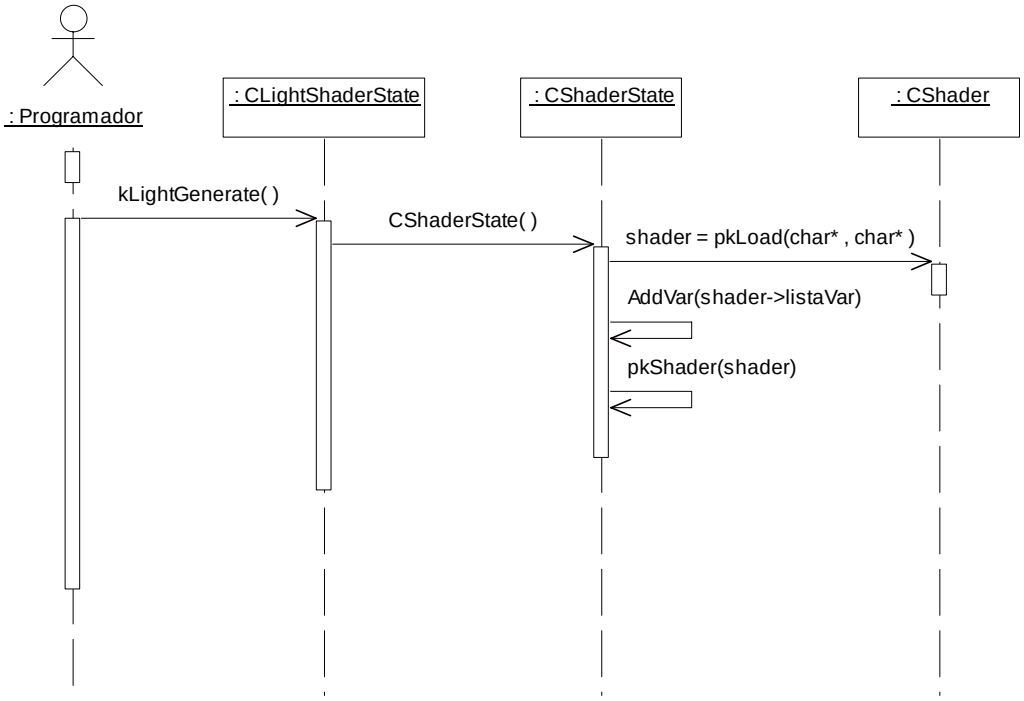

Figura 21: Diagrama de secuencia CU Adicionar Luz

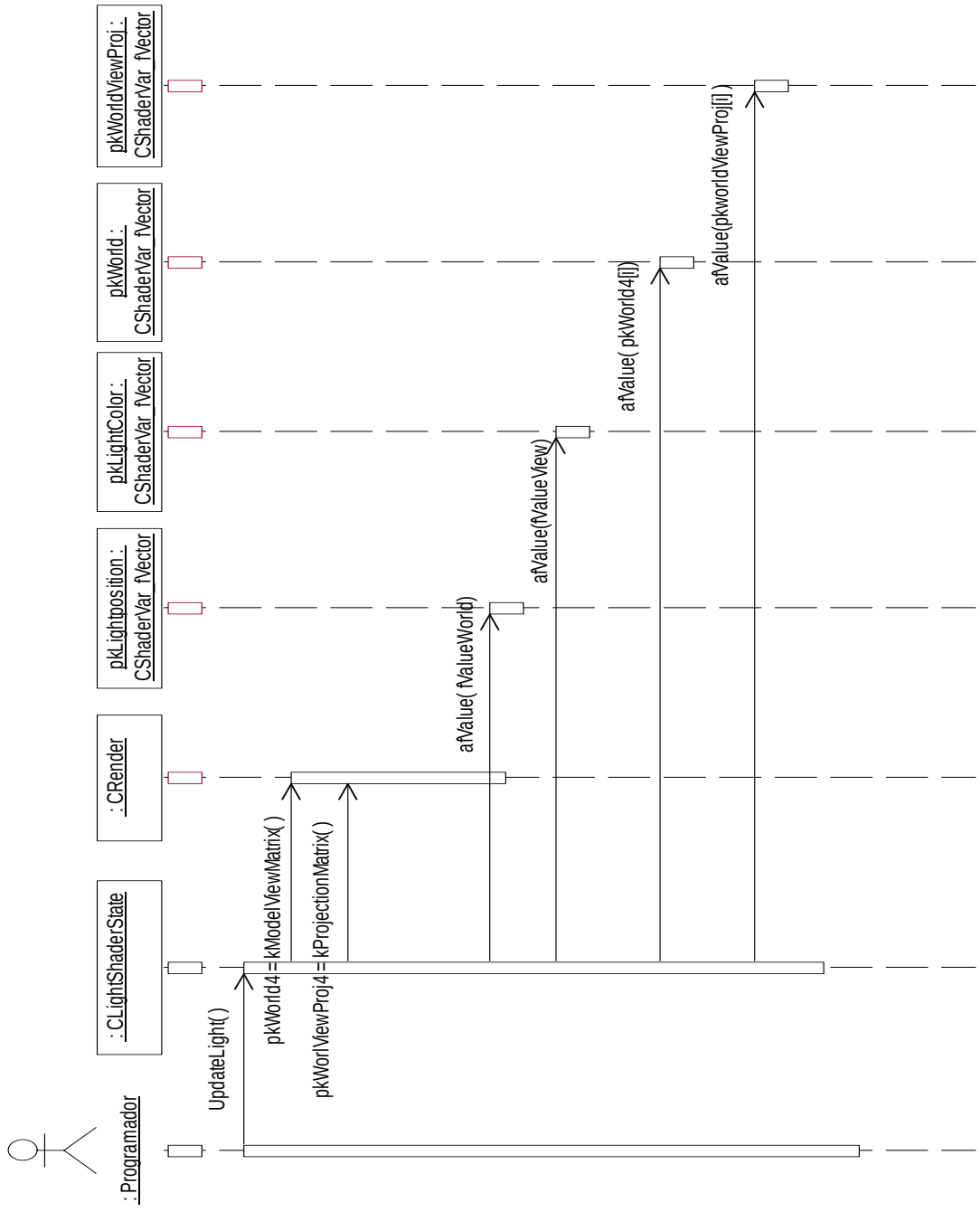

Figura 22: Diagrama de secuencia CU Actualiza Luz

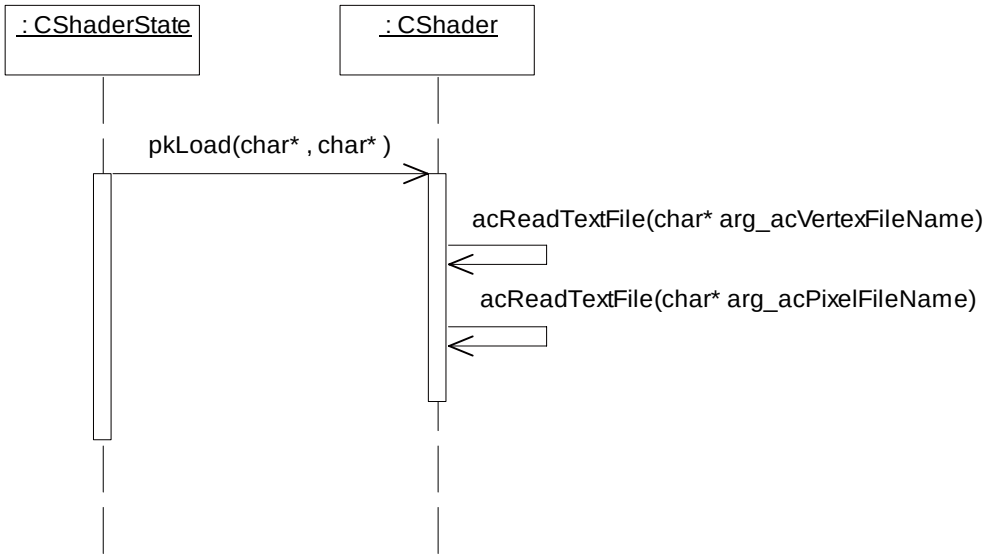

Figura 23: Diagrama de secuencia CU Cargar light-shadow shader

Conclusiones

Al concluir este capítulo se tiene concebido detalladamente el diseño completo del sistema y la secuenciación de pasos traducida a mensajes entre clases de los primeros casos de usos a desarrollar, con lo que se puede pasar a la etapa de implementación del proyecto.

Capítulo 5 Implementación del sistema

Introducción

Esta etapa del proyecto constituye el paso del diseño de clases a la creación de componentes físicos, que se traducen en ficheros .cpp correspondiente a la implementación en C++. Además ejecuta la fase de prueba.

5.1 Estándares de codificación

Se programará en inglés, debido que las palabras son simples, no se acentúan y es un idioma muy difundido en el mundo informático. Se respetarán los estándares de codificación para C++ (identado, uso de espacios y líneas en blanco, etc.).

El conocimiento de los estándares seguidos para el desarrollo del módulo permitirá un mayor entendimiento del código, y es una exigencia de los autores de la misma que cualquier módulo que se añada debe estar codificado siguiendo estos estándares.

Nombre de los ficheros:

Se nombrarán los ficheros .h y .cpp de la siguiente manera:

STKNameOfUnits.cpp

Se usará **GT** para identificar el nombre de la herramienta (en definición!!)

Constantes:

Las constantes se nombrarán con mayúsculas, utilizándose el "_" para separar las palabras: MY_CONST_ZERO = 0;

Tipos de datos:

Los tipos se nombrarán siguiendo el siguiente patrón:

Enumerados: enum **E**MyEnum {**ME**_VALUE, **ME**_OTHER_VALUE};

Indicando con "**E**" que es de tipo enumerado. Nótese que las primeras letras de las constantes de enumerados son las iniciales del nombre del enumerado. Véase otro ejemplo:

enum **E**Node**T**ype {**NT**_GEOMETRYNODE,…};

Estructuras: struct **S**MyStruct {…};

Indicando con "**S**" que es una estructura. Las variables miembros de la estructura se nombrarán igual que en las clases, leer más adelante.

Clases: class **C**ClassName;

Indicando con "**C**" que es una clase

Declaración de variables:

Los nombres de las variables comenzarán con un identificado del tipo de dato al que correspondan, como se muestra a continuación. En el caso de que sean variables miembros de una clase, se le antepondrá el identificador “” (en minúscula), si son globales se les antepondrá la letra “g”, y en caso de ser argumentos de algún método, se les antepondrá el prefijo “arg_”.

Tipos simples:

bool **b**VarName;

int **i**Name;

unsigned int **ui**Name;

float **f**Name;

char **c**Name;

char* **ac**Name; // arreglo de caracteres

char* **pc**Name; // puntero a un char

char** **aac**Name; // bidimensional

char** **apc**Name; // arreglo de punteros

bool **m_b**MemberVarName; //variable miembro

char **g**cGlobalVarName; //variable global, no se le antepone “”

short **s**Name;

Instancias de tipos creados:

EMyEnumerated **e**Name;

SMyStructure **k**Name;

CClassName **k**ObjectName;

CClassName* **pk**Name; //puntero a objeto

CClassName* **ak**Name; //arreglo de objetos

CClassName* **ak**Name; // variable miembro de clase

IMyInterface* **pI**Name; //puntero interfaces

Métodos

En el caso de los métodos, se les antepondrá el identificador del tipo de dato de devolución, y en caso de no tenerlo (void), no se les antepondrá nada. Solamente los constructores y destructores comenzarán con “”.

En el caso de los argumentos se les antepone el prefijo “arg_”

Constructor y destructor:

CClassName (bool **arg**_bVarName, float& **arg**_fVarName);

~**C**ClassName ();

Funciones:

bool bFunction1 (…);

int* **pi**Function2 (…);

CClassName* **pk**Function3 (…);

Procedimientos:

void Procedure4 (…);

Métodos de acceso a miembros

Los métodos de acceso a los miembros de las clases no se nombrarán “Gets” y “Sets”, sino como los demás métodos, pero **con el nombre** de la variable a la que se accede y sin “m_”:

```
int iMyVar;  //variable
```

Obtención del valor:

```
int iMyVar();
{
   return iMyVar;
}
```

Establecimiento del valor:

```
void MyVar(char* arg_iMyVar)
{
  iMyVar = arg_iMyVar;
}
```

Obtención y establecimiento del valor:

```
int& iMyVar();

{

   return iMyVar;

}
```

5.2 Diagrama de componentes

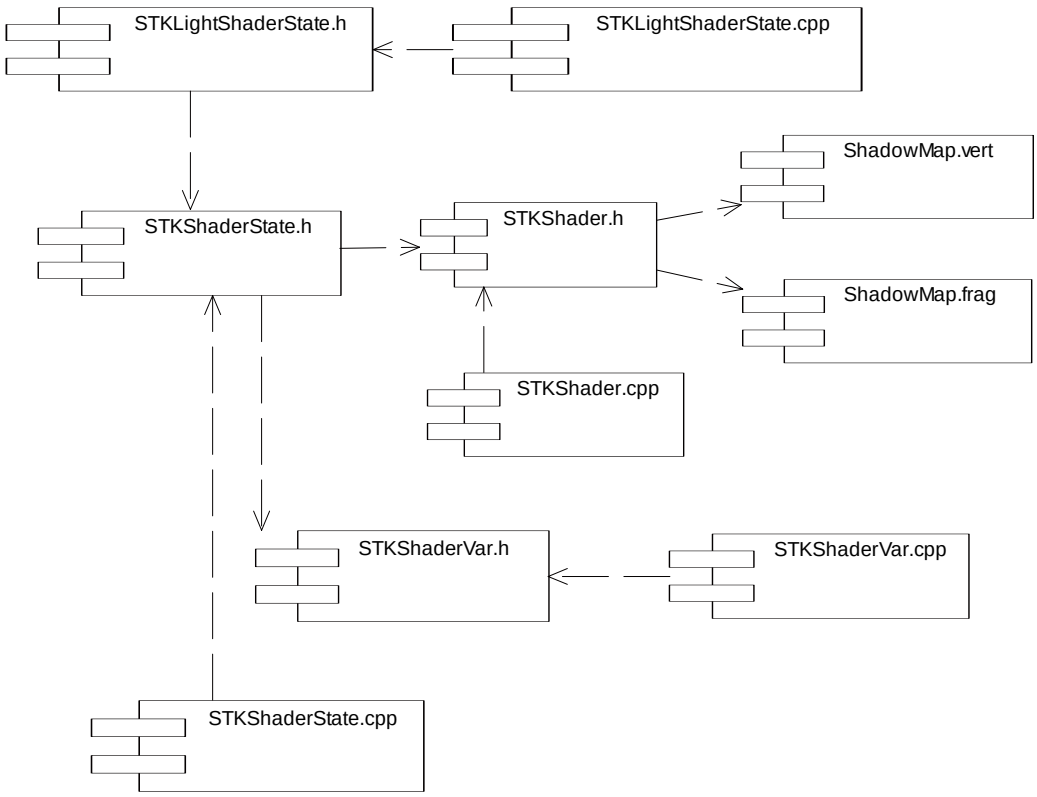

Figura 24: Diagrama de componentes

Conclusiones

En este momento se encuentra todo preparado para pasar a la etapa de programación de los casos casos de uso desarrollados en el primer ciclo. Como posibilidad adicional que brinda la herramienta Rational Rose, ya es posible generar el código fuente de los componentes relacionados con los casos de uso a desarrollar en el primer ciclo.

Conclusiones

Para dar cumplimiento a los objetivos de este proyecto, en correspondencia con las exigencias del cliente, fue necesario primeramente hacer un estudio de las técnicas, tecnologías y tendencias en cuanto a la implementación de efectos visuales en Sistemas de Realidad Virtual, específicamente, lo relacionado con luces y sombras dinámicas. Se analizaron los avances actuales en el hardware en cuanto al uso de las tarjetas gráficas y su máximo aprovechamiento para obtener efectos de alto realismo, los algoritmos más populares usados por las grandes compañías para la creación de entornos 3D, teniendo en cuenta las deficiencias de estos que debían ser erradicadas con este trabajo. A partir de esta investigación, se propone una solución factible.

Después se realizó el proceso de Ingeniería del Software utilizando el Proceso Unificado del Software (RUP) como metodología de desarrollo, a partir del cual se obtuvo un módulo funcional que cumple con los requisitos del cliente, y que permite al programador cambiar la técnica de iluminación sin afectar la estructura del módulo, solo con un conocimiento mínimo de como funciona el mismo; además este está concebido para poder adaptarlo fácilmente a otra biblioteca gráfica, sin que sufra su estructura, así como a nuevas funcionalidades.

Recomendaciones

Se recomienda la extensión del módulo a la biblioteca gráfica *DirectX.*

Profundizar un poco más en las características y particularidades del lenguaje *HLSL* anteriormente expuesto.

Se recomienda además la incorporación de las sombras guiadas por el estudio que se hizo acerca de los algoritmos actuales más usados, utilizando la misma estructura propuesta para la incorporación de la iluminación.

Referencias bibliográficas

Fuentes Electrónicas

[1]. CHOVER, Miguel. Informática Gráfica, 2004
[http://www3.uji.es/~jromero/grafica]

[2]. KAUFMANN , Morgan.
Computer animation: algorithms and techniques, 2004.
[http://www.cis.ohio-state.edu/~parent/book/outline.html]

[5]. UNIVERSIDAD DE VALENCIA, 2004. [http://informatica.uv.es]

[8]. Wales, Jimmy y Sanger, Larry. GPU, 2006.
[http://es.wikipedia.org/wiki/Graphics_Processing_Unit]

[9]. J. Rost, Randi. OpenGL® Shading Language, Second Edition, 2006.

[10]. Baudes, Jose. Light Maps, 2006.
[http://64.233.183.104/search?q=cache:D2YmGHEfn0EJ:dmi.uib.es/~josemaria]

[11]. Caro, Zamir. Realidad Virtual, 2005.
[http://www.monografias.com/trabajos28/realidad-virtual/realidad-virtual.shtml#element]

[12]. Budavari, Diego. Introducción a las 3D, 2001
[http://gargonscene.iespana.es/gargonscene/articulos/3d02.htm]

[13]. *Shader*, 2006. [http://en.wikipedia.org/wiki/Shader]

Libros

[3]. LUNA, Frank D. Introduction to 3D Game Programming with DirectX.
USA, WordWare Publishing, Inc. 2003.

[4]. ASTLE, Dave y HAWKING, Kevin. OpenGL Game Programming.
USA, Prima Tech Publishing. 2001.

[6]. Sense8 CORPORATION TEAM. WorldToolkit Release 9, Reference Manual.
USA, Sense8 Corporation (www.sense8.com). 1999.

[7]. BIRN, Jeremy. Digital lightning and rendering.
USA, New Riders Publishing. 2000.

Tesis

[14].Puig Placeres, Frank.
Empleo eficiente del hardware gráfico en la iluminación de entornos virtuales. Universidad de las Ciencias Informáticas. Ciudad de La Habana, 2006. 106.

Índice de figuras, ecuaciones y tablas

Índice de figuras

Índice de ecuaciones

Índice de tablas

Glosario de Abreviaturas

2D: Dos dimensiones.

3D: Tres dimensiones.

API: Application Programmer's Interface, (interfaces para programadores de aplicaciones).

CPU: Unidad Central de Procesamiento.

DS: Deferred shading. Técnica usada para la creación de iluminación y/o sombras dinámicas en SRV.

EH: Esferas Harmónicas. Técnica utilizada para la generación de iluminación de manera eficiente y real.

GLSL: Opengl Shading Language. Lenguaje de programación de alto nivel para realizar aplicaciones gráficas y lograr mayor eficiencia y realismo.

GPU: Unidad de Procesamiento Gráfico.

HLSL: *High Level Shading Language*. Lenguaje de programación de alto nivel para realizar aplicaciones gráficas y lograr mayor eficiencia y realismo.

HW: *Hardware*.

MS: Mapeo de Sombra. Técnica usada para la creación de sombras dinámicas en SRV.

OA: Oclusión Ambiental. Técnica usada para la creación de sombras dinámicas en SRV.

RGB: *Red Green Blue*. Los colores RGB consisten en tres números, que representan los niveles de rojo, verde y azul, respectivamente, conocidos como colores aditivos primarios.

RV: Realidad Virtual.

SRV: Sistemas de Realidad Virtual.

SV: Sombras Volumétricas. Definición del tipo de sombra generado por DS.

SW: *Software.*

VRAM: Video *Random Access Memory*: Memoria de Acceso Aleatorio dedicada a Video.

Glosario de Términos

A:

Aliasing: las líneas, especialmente las que están casi horizontales o verticales, aparecen dentadas o irregulares debido a su representación por *píxels*. Este escalonamiento es llamado *aliasing*.

Aplicaciones interactivas: Herramientas computacionales en las que el usuario tiene un papel activo, como tomar decisiones e interactuar con esta.

Arreglos: Es un conjunto de variables o registros del mismo tipo que puede estar almacenados en memoria principal o en memoria auxiliar.

Atenuación: Disminución en la magnitud de la intensidad de la fuente de luz.

*****

C:

Comportamiento (behavior): cambio de atributos o características de los objetos visibles y no visibles del mundo virtual y que cambian el aspecto visual de la escena logrando una animación.

*****

D:

Deferred Shading: Técnica utilizada para la creación de sombras dinámicas en SRV.

Desnormalizar: Lo opuesto a normalizar (ver normalizar).

Dinámica: Calificativo que sugiere la actividad, el movimiento, el cambio, la transformación estructural y funcional.

Distorsiones: Alteración de las cualidades y/o características de un electo.

*****

E:

Estado de Shader: Información de estados de iluminación y sombras para representar en las escena.

Estático: Lo opuesto a dinámico (ver dinámico).

Estructura: Conjunto de propiedades y funciones que definen en elemento.

*****

F:

Frame: cada uno de las imágenes que componen una animación.

Frame buffer: Memoria usada para retener uno o más *frames* para su posterior uso.

Frustum de cámara: conjunto de planos que definen el volumen de visión de la cámara.

*****

H:

Hardware gráfico: Dispositivos necesarios para crear aplicaciones gráficas.

*****

I:

Iluminación esferas harmónicas (Spherical Harmonics Lighting): Técnica utilizada para la creación de luces dinámicas en SRV.

Inmersión: Lograr acaparar toda la concentración y atención de un usuario de manera que crea que se encuentra en una situación real.

Interface: Una interfaz es la parte de un programa que permite a éste comunicarse con el usuario o con otras aplicaciones permitiendo el flujo de información.

Interpolación: algoritmo matemático que a partir de varios puntos en el espacio, describe una función que contiene a los puntos intermedios.

*****

L:

Luces dinámicas: Luces con un comportamiento dinámico. (Ver dinámica).

*****

M:

Malla: forma de representar un modelo a partir de polígonos. Colección de vértices, aristas y polígonos conectados de forma que cada arista es compartida como máximo por dos polígonos.

Mapa de texturas: correspondencia entre vértices de una textura y los vértices del modelo.

Mapa de luz (Light Map): Textura utilzaza para almacenar los cálculos de la incidencia de la luz sobre objetos y superficies.

Mapeado bump: Técnica utilizada para crear irregularidades en la geometría de algún objeto.

Mapeo de Sombras (Shadow Mapping): Mapeo de Sombra. Técnica usada para la creación de sombras dinámicas en SRV.

Material: combinación de luces y colores usados para definir una apariencia.

Matrices de transformación: matrices definidas para calcular nuevas coordenadas a partir de coordenadas existentes según una determinada transformación gráfica (rotación, trasladación, escalado y reflexión).

Matriz: Arreglo de elementos.

Modelo: Prototipo para la animación.

*****

N:

Normal: (ver vector normal).

Normalizar: Transformar coordenadas o parámetros que tengan un valor predeterminado.

*****

O:

Oclusión: Desaparición total o parcial de objetos en una escena.

*****

P:

Perturbación: Evento que altera o modifica los acontecimientos normales.

Pipeline gráfico: Conjunto de procedimientos para lograr obtener una aplicación gráfica.

Píxel: abreviatura de “picture element”. Es la menor unidad de información de una imagen digital.

Píxel shader: Encargado de realizar todas las operaciones a nivel de *píxel* como es el cálculo de la iluminación *per-píxel*, mapeo de texturas, uso de los mapas de normales para la determinación de la normal del píxel, etc.

*****

R:

Realidad Virtual: Simulación de un medio ambiente real o imaginario que se puede experimentar visualmente en tres dimensiones. La realidad virtual puede además proporcionar una experiencia interactiva de percepción táctil, sonora y de movimiento.

Recursos: Son los elementos de una computadora que utilizan los dispositivos para poder funcionar correctamente.

Reflexión: Fenómeno que ocurre cuando la luz incide sobre una superficie y es desviada por ésta sin cambiar de medio.

Reflexión Especular: La reflexión es especular cuando la superficie es lisa, y difusa cuando la superficie es rugosa.

Rendereado (rendering): crear en forma automática una imagen de acuerdo al modelo tridimensional que existe en el ordenador.

*****

S:

Shader: Un *shader* define las características finales de un objeto. Por ejemplo, un *shader* puede definir el color y la reflectividad de una superficie.

Sistema de funciones fijas (Fixed-Function): Es uno de los dos métodos que se utilizan actualmente para modificar los resultados gráficos; el otro es el Sistema de Funciones Programables, también conocido por *Shader*.

Sistema de funciones programables (Programmable-Function): (ver *shader*).

Sistema de Realidad virtual: sistema informático interactivo que ofrece una percepción sensorial al usuario de un mundo tridimensional sintético que suplanta al real.

Sombras dinámicas: Sombras con comportamiento dinámico (ver dinámica).

*****

T:

Tarjeta gráfica: Es una tarjeta de circuito impreso encargada de transformar las señales eléctricas que llegan desde el microprocesador en información comprensible y representable por la pantalla del ordenador.

Teselación: Composición de una o varias figuras planas que, repitiéndose con regularidad, pueden llenar el plano.

Textura: imagen que sirve de “piel” a los modelos en un mundo virtual.

Tiempo de procesamiento: Tiempo en que la aplicación se demora en procesar toda la información.

*****

U:

Überlight (ÜberLight Shader): Técnica utilizada para la creación de luces dinámicas en SRV.

*****

V:

Vector: cantidad que expresa magnitud y dirección.

Vector normal: vector cuyos puntos están en dirección perpendicular a una superficie.

Vertex shader: Encargados de transformar todos los vértices de la escena. En ellos se ejecutan las transformaciones de espacio objeto a espacio de mundo, de cámara, y finalmente se obtiene la posición en la pantalla.

Volumen de visión de la cámara: sección de espacio 3D visible por la cámara, definido por seis planos: cercano, lejano, izquierdo, derecho, arriba y abajo.