UNIVERSIDAD DE LAS CIENCIAS INFORMATICAS
UCI
FACULTAD 5 REALIDAD VIRTUAL

Luces y Sombras dinámicas para Sistemas de Realidad Virtual.

TRABAJO DE DIPLOMA PARA OPTAR POR EL TÍTULO DE
INGENIERO INFORMÁTICO

Autores

Yeniel Cabrera Amaya

Alexeidis Frómeta Estévez

Tutores

Ing. Lien Muguercia Torres

Ing. Yaíma Nodarse Valdés

Ciudad de la Habana

Julio, 2008

En el verdadero éxito,

la suerte no tiene nada que ver;

la suerte es para los improvisados y aprovechados;

y el éxito es el resultado obligado de la constancia,

de la responsabilidad, del esfuerzo,

de la organización y del equilibrio entre la razón y el corazón.

(Anónimo)

DECLARACIÓN DE AUTORÍA

Declaramos que somos los únicos autores 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:

Alexeidis Frómeta Estévez

_____

Yeniel Cabrera Amaya

_____

Tutores:

Ing. Lien Muguercia Torres.

_____

Ing. Yaíma Nodarse Valdés.

_____

Datos de Contacto

Autores

Nombre: Yeniel Cabrera Amaya

Correo Electrónico: ycamaya@estudiantes.uci.cu

Nombre: Alexeidis Frómeta Estévez

Correo Electrónico: afrometa@estudiantes.uci.cu

Tutores

Nombre: Lien Muguercia Torres

Institución: Universidad de las Ciencias Informáticas (UCI)

Título: Ingeniero en Informática

Categoría Docente: Profesor Instructor

Correo Electrónico: lmuguercia@uci.cu

Nombre: Yaíma Nodarse Valdés.

Institución: Universidad de las Ciencias Informáticas (UCI)

Título: Ingeniero en Informática

Categoría Docente: Profesor Instructor

Correo Electrónico: ynodarse@uci.cu

Dedicatoria

…A mi madre que además de darme la vida, ha estado siempre preocupándose por mí,
…A mi padre por su ejemplo y cariño,
…A mis hermanos,
…A mis abuelos por parte de papi (Lolo y Melecia) y por parte de mami (Maximino y Dora).
Alexeidis

… A mis padres por su apoyo y dedicación, a mis dos hermanitos que son mi razón de ser.
…A mis tutores por su ayuda en todo momento,
…A mis abuelitos, que aunque ya no están presentes, fueron como mis segundos padres los cuales supieron ganarse mi cariño,
…A mi abuelo Amaya por preocuparse por mi,
…A toda mi familia, a la Revolución, a la UCI y a Fidel.
Yeniel

Agradecimientos

Este trabajo es el resultado del apoyo de muchos amigos que han colaborado con su desarrollo. Mencionar todos los nombres sería una lista interminable, pero obviarlos sería una falta de reconocimiento a su esfuerzo. Sinceramente agradecemos la colaboración de las tutoras Lien y Yaíma por su preocupación por ayudarnos en todo momento lo cual hizo posible este trabajo, a todos los profesores que nos han guiado en el mundo de la investigación y los gráficos por computadoras. Y esta Universidad de que nos ha dado todo lo necesario para nuestra formación profesional.

Yeniel:

A mi hermanita Nisdy por quererme tanto y siempre estar a mi lado, ayudándome en todo y guiándome por el camino correcto, a mi hermanito guille , que aún siendo el más pequeño, siempre me ha cuidado y me ha ayudado mucho, y por quien trato de buscar lo mejor de mí para que pueda tomar un ejemplo a seguir .A mi mamá por siempre estar a mi lado y guiarme en todo momento, por su preocupación y amor infinito ,a mi papá por estar siempre dispuesto a ayudarme ,por aconsejarme y por ser un ejemplo a seguir . A la revolución, a la UCI, por darme la oportunidad de estudiar y de prepararme en la vida. A mis compañeros de grupo por compartir en los momentos buenos y no tan buenos, en especial a Daylin, Loiret y compañero de tesis Alexeidis.

Alexeidis:

Agradecerles principalmente a mis padres por guiarme en la vida y fundamentalmente en los estudios. A mis tíos Jorge, Rudelis, Rogelio, Joel, Niño, Tella, Delmis, Eloina, Lalia, Papito, Mercedes, por enseñarme convivir como una familia unida. A mis colegas de la universidad Rainer, Leonardo Rafael, Ricardo Ernesto, Yerandi, Lester Oscar, Yeniel, Yariel, Yunior Miguel, Yaself, Aylin, Fácil y al resto del mis compañeros de aula por su ayuda tanto en el aspecto docente, espiritual y la vida en general. A mis amigos del barrio Gaspar, Yannier, Annie, Huclides, a mis primos Jorgito, Wilmeris, Johannis y Cariuska, a mis hermanos. A todos ellos por apoyarme desde hace más de 10 años en todos los aspectos de la vida. A Yasnahi, por estar a mi lado en todo este tiempo y encontrar en ella apoyo, cariño y afecto.

Resumen

En los gráficos por computadora, lograr la iluminación y el sombreado de manera eficiente ha sido un tema altamente polémico. En el momento de hacer que las imágenes alcancen un alto nivel de realismo requiere de un gran consumo de recursos y en ocasiones no se logra cumplir con las expectativas. Existen muchos planteamientos para resolver este problema y se han desarrollado varios algoritmos con el fin de lograr entornos virtuales cada vez más realistas. Esta investigación se propone mejora el módulo que le da tratamiento a las luces y sombras dinámicas de la herramienta básica creada por el proyecto: Herramientas de Desarrollo para Sistemas de Realidad Virtual (HDSRV) que se desarrolla en la Facultad 5 de esta Universidad, con el objetivo de lograr mayor veracidad en las escenas 3D.

Todo el desarrollo derivado de investigaciones sobre el tema han conllevado a múltiples métodos unos requieren más memoria, algunos implican más tiempo de procesamiento y otros, solo se aplican en clases especiales de objetos. Decidirse por un método para una aplicación particular puede depender de factores como la complejidad de la escena, el tipo de objetos que se deben desplegar, el equipo disponible y la necesidad de generar despliegues animados o estáticos. A partir del estudio de estos algoritmos que solucionan el problema referente a la iluminación y sombreado de escenas virtuales de modo eficiente, se plantean ventajas y desventajas según sus características. Además se propone una vía para el manejo de las luces y sombras: los *Shaders*, que con la aparición de los procesadores gráficos (GPU), permite programar dicho hardware como si fuera una CPU, mediante pequeñas rutinas en ensamblador.

El resultado, centrado en la realización de un módulo de clases que permitiera el manejo de los *shaders* para lograr efectos de iluminación y sombreado, fue realizado con buenas expectativas a nivel nacional debido a la tecnología de primera línea que este aporta; siendo este una buena base para la perfección de la herramienta (HDSRV).

Palabras Claves:

Herramientas de Desarrollo para Sistemas de Realidad Virtual (HDSRV), Unidad de Procesamiento Gráficos (GPU), Shaders.

Índice

Introducción

El avance, en los últimos tiempos, de la informática ha propiciado la creación y extensión de un nuevo término: Realidad Virtual (RV). Iniciada en los programas de entrenamientos militares, simuladores de vuelo, centros de investigación, ha pasado a formar parte de programas muy variados en el ámbito profesional.

Se han descubierto, igualmente, distintas e importantes posibilidades educativas de la RV. Estas aplicaciones respaldan a la afirmación de que un conocimiento se retiene mucho mejor cuando se experimenta directamente, que cuando simplemente se ve o se escucha. Su potencial radica en la capacidad que tiene para permitirnos experimentar y, en cierta medida, palpar el resultado de nuestro desenvolvimiento y actividad dentro de un ambiente tridimensional, creado artificialmente.

A consecuencia de su desarrollo han revolucionado no solo la informática; actualmente se emplea en un número creciente de sectores: en la arquitectura, la educación, en la ingeniería, la medicina y dentro de la medicina se destaca la cirugía.

La Universidad de las Ciencias Informáticas (UCI) y específicamente la Facultad 5 ya hace algunos años han dedicado recursos, tanto humanos como materiales para el estudio y el desarrollo de la RV. Existe un proyecto: Herramientas de Desarrollo para Sistemas de Realidad Virtual (HDSRV), que se encarga de desarrollar una herramienta básica que reúne las funcionalidades comunes en los proyectos pertenecientes al perfil de Entornos Virtuales de la facultad, dicha herramienta se encuentra ya en estado de perfeccionamiento. Para el tratamiento de Luces y Sombras Dinámicas que son producidas por y sobre objetos que están en movimiento, ya hay implementado un módulo, y a partir de la necesidad de crear entornos virtuales cada vez más reales por parte de los programadores, se necesita darle una mejora tomando como concepción que el realismo del entorno virtual 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.

A partir de, cómo mejorar el módulo antes creado y que está siendo utilizado por el proyecto HDSRV, se propone la creación de uno nuevo que reúna los algoritmos que den solución a este problema, tomando como objeto de estudio el proceso de visualización de luces y sombras en entornos de realidad virtual y como campo de acción el trabajo con los *shaders* para el desarrollo de dicho proceso.

El objetivo principal de este trabajo es implementar un módulo de clases que permita un tratamiento más amplio de luces y sombras dinámicas en entornos virtuales, traduciéndose esto, en que el módulo debe permitir representar más de una fuente de luz en la escena y que permita lograr un sombrado dinámico, para ello se plantean un grupo de tareas que se resume a continuación:

- Estudiar las características básicas de los módulos de efectos visuales, particularizando en las luces y sombras.
- Analizar algoritmos utilizados en la creación y visualización de luces y sombras dinámicas.
- Estudiar las principales herramientas existentes para realizar la simulación de entornos virtuales en 3D en el mundo.
- Observar las potencialidades de los lenguajes de alto nivel utilizados en el mundo para desarrollo de efectos visuales.
- Explorar las potencialidades de las tarjetas gráficas a utilizar.
- Rediseñar la biblioteca de clases que de solución a la visualización de las luces y sombras dinámicas.

Como resultado de este trabajo se pretende obtener un módulo que aporte mayor realismo a un entorno virtual a través de las luces y las sombras dinámicas en una escena 3D utilizando los *shaders*. Esta investigación servirá de base para mejorar el módulo de luces y sombras dinámicas que incluye la herramienta HDSRV permitiendo aumentar la velocidad en el procesamiento y la calidad en la escena.

1

Capítulo 1 Fundamentación Teórica

Introducción

En el mundo actual, la computadora logra un papel ponderante al poder simular realidades y visualizarlas. De esa manera, se logra una aproximación bastante aceptable a los modelos que de otra manera sólo estarían en nuestra imaginación. Pero para lograr tal simulación es necesario que sea lo más real posible. A partir de la iluminación de una escena 3D los objetos que en ella se encuentran reflejan regiones en sombra lo cual proporcionan una referencia de gran valor para la interpretación espacial de la imagen sintetizada, y aportan mayor realismo y credibilidad a la misma. Por esto, el problema de la determinación y representación (en tiempo real) de las regiones en sombra en una escena 3D se ha convertido en uno de los más interesantes y más estudiados.

Partiendo de la programación gráfica de las tarjetas se puede 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, pero basándose en que para lograr generar imágenes cada vez más reales, se necesita hacer modelos que permitan una representación de la iluminación, de las sombras y de los efectos de reflexión cada vez más realistas.

En este capítulo se tratan temas relacionados con la representación de las luces y sombras en Entornos Virtuales, para ello se dedica un epígrafe para los materiales, así como también se hace referencia a la implementación de luces usando los shaders, se profundiza el estudio del lenguaje Cg que permite aprovechar los avances experimentados por los procesadores gráficos programables, de ahí se hace necesario el estudio de la GPU para un mejor entendimiento de todo este proceso.

A partir de que es una tesis que le da continuidad a otra [13] solo se dará un breve resumen de los algoritmos más importantes de iluminación y sombreado como es el caso de Iluminación de Hemisferios con Mapeado Bump, los Controles Überlight, profundizando en el estudio del algoritmo Esferas Harmónicas, Phong y el Mapeo de Sombras (*Shadows Mapping*) que serán la solución propuesta de esta investigación.

1.1 Objetos & Materiales

Como para un desarrollador de entornos virtuales, ya sea, desarrollador de juegos, de simuladores, etc., es interesante reproducir ambientes que la mayoría de la gente nunca ha visto. Se podría comenzar dando una intuición adelantada sobre esto, puesto que cómo son materiales diferentes deberían lucir como si fueran observados en su ambiente. Se puede mejorar la forma y materia de las escenas haciéndolas corresponder con objetos que nos son familiares.

A continuación se muestra un listado de algunos de los materiales de los que se harán referencia a lo largo de este epígrafe:

- Plástico
- Madera
- Hojas y Vegetación
- Metales
- Concreto y Piedra
- Pelo y Piel
- Materiales Transparentes

Plásticos

Muchos de los objetos vistos en escenas 3D tienen una apariencia muy parecida al plástico, esto se debe a que se utilizan algunos modelos de iluminación que producen este efecto. A partir de que en el mundo hay varios tipos de plástico, de madera, de metal, y de materiales orgánicos, los objetos plásticos vienen en muchos colores, formas y acabados, pero el material en sí es bastante coherente en términos de la composición.

La mayoría de plásticos están hechos de un substrato blanco o transparente que ha sido infundido con partículas de tinte, como se muestra en Figura 1. Cuando la luz incide en un objeto plástico, algunos rayos de luz se reflejan fuera de la superficie extrema. De un ángulo específico, el espectador verá la luz, y esa parte del objeto parecerá blanco. Algún rayo de la luz penetrará en el objeto y se reflejará completamente en las partículas de tinte. [5]

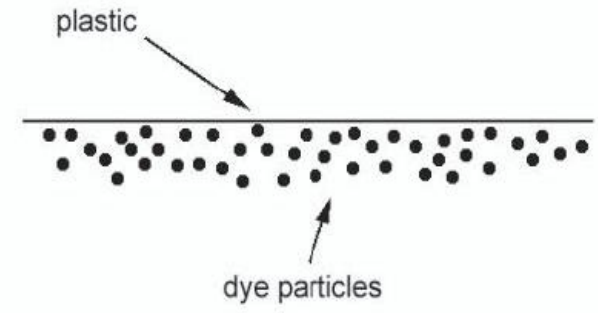

Figura 1: Plástico con partículas de tinte.

Estas partículas absorberán algunas longitudes de onda de la luz. Como se muestra en la Figura 2, algunos ángulos, se ven en el substrato plástico, y los otros ángulos, se ven en las partículas de tinte.

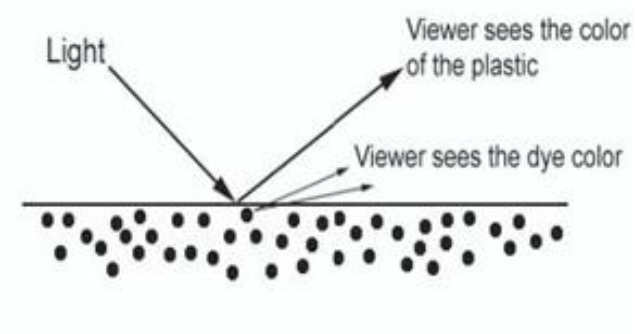

Figura 2: Textura plástica vista de diferentes ángulos.

Este efecto es más pronunciado en objetos suaves porque el ángulo de la luz relativo a la superficie es medianamente coherente sobre grandes porciones de la superficie. El plástico también puede ser no brilloso. En este caso, la superficie es rugosa, la luz penetra y se refleja menos uniformemente sobre la superficie, se obtendrán puntos iluminados, pero que se harán cada vez menos pronunciados.

Hay dos razones diferentes para esto. La primera, las superficies rugosas causan una desigual reflexión. Y la segunda esta dada por la rugosidad de la superficie que causan algunos baches los cuales arrojan pequeñas sombras en la superficie. Al final, se obtendrá puntos de interés sutiles. [5]

Madera

La madera es un material muy común tanto como en entornos virtuales como reales. Lamentablemente, es también muy difícil hacerla de forma tal que sea lo mas realista posible porque los patrones y propiedades de la superficie puede ser muy complejos. Para sintetizar el problema, la madera aparece en muchas formas diferentes. Se pueden ver en árboles de un bosque, muebles barnizados en un lobby de un hotel, o en la madera utilizada para construir. Cada forma tiene un aspecto diferente definido por sus atributos físicos. [5]

Corteza

Una de las más importantes características visuales que separa a los árboles de los metales esta dada por la existencia de su corteza. En la naturaleza, el patrón áspero de la corteza se debe al

crecimiento del árbol. La mayor parte del detalle visual está compuesto de las áreas de luz y de oscuridad en los altibajos de la corteza como se muestra en la Figura 3.

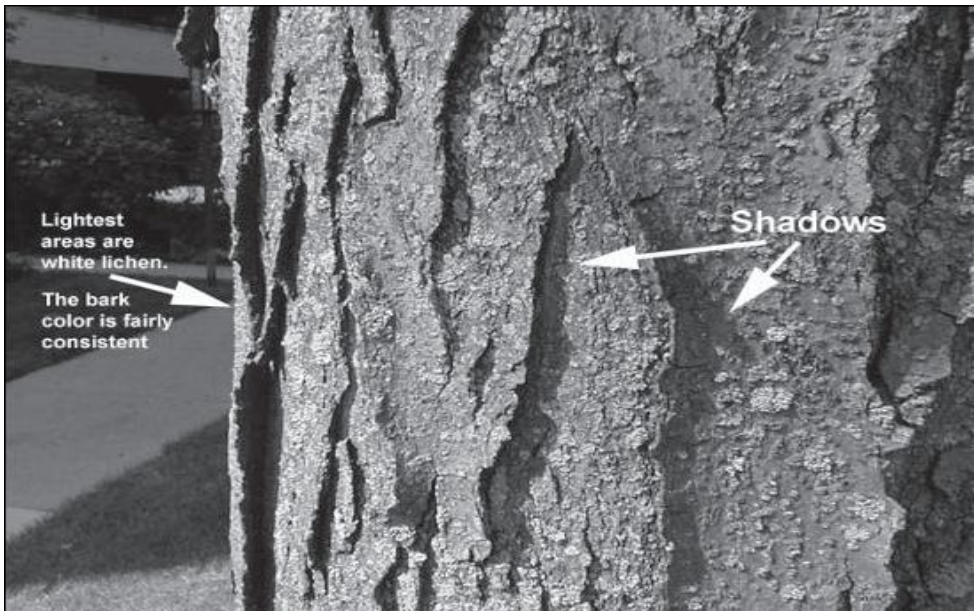

Figura 3: Corteza de árbol.

Es un punto de vista muy simple el que se muestra en la Figura 3 donde se muestran zonas iluminadas y oscuras en la corteza. Este no es el caso, el color de la corteza realmente no se altera tanto sobre la superficie del árbol. En cambio, casi todo el detalle proviene de la geometría de la corteza. Las áreas de luz y de oscuridad son causadas por diferencias en la iluminación, no por los cambios en el color de la corteza en sí. Si al tomar una fotografía de una corteza y luego intenta utilizarla como una textura, se deben codificar las condiciones de iluminación de la textura.

La corteza de un árbol tiene áreas de diferentes colores y aspereza, donde tienden a ser más brillantes las superficies con cortezas más delgadas.
En una escena real, este tipo de corteza sería bastante difícil de reproducir con precisión. En general, se desea incluir todos estos factores, pero lo que se quiere es que los árboles no tengan una apariencia de un plástico brillante. [5]

Tablas

Se define como tablas a la madera desnuda es decir sin cáscara, que se utiliza para la construcción. La característica más notable de las tablas es que es sacada del árbol y muestra la estructura de la madera por dentro. En la Figura 4 se muestran varios ejemplos.

Figura 4: Varios ejemplos de Madera.

Los patrones que se estructuran pueden ser mirados como cambio en color, pero a menudo contienen otros rasgos. Algunas partes de la fibra podrían tener más abolladuras y cordilleras. Otras partes de la madera podrían ser más brillantes o más pronunciadas que otras (quizás debido a cambios en concentraciones de savia o aceites), aunque la mayoría de los materiales de madera tiene un acabado mate. Además, los pedazos de madera exhiben cierta cantidad de radiación, lo que significa que la dirección de la fibra de la madera afecta a la reflexión de la luz desde diferentes ángulos. [5]

Hojas & Vegetación

Además de árboles y madera, muchas escenas interiores y externas incluyen hojas, pastos, y otras formas de vegetación. Ahora, la mayoría de plantas en escenas de tiempo real son tratadas con texturas verdes de la hoja, pero las cosas no son tan simples. Como en la corteza de árbol, gran parte de los detalles sobre las hojas proviene de la manera en que luz interactúa con las características de su superficie. Muchas hojas tienen una cara brillante y una cara mate. Otras hojas tienen pequeños pelos o espinas que interactúan con la luz de una forma muy interesante.

Las hojas de la Figura 5 son en su mayoría homogéneas en color. El detalle que se muestra es a causa de la iluminación de la superficie. Además, la mayoría de las hojas son translúcidas, lo que significa que permiten que una pequeña cantidad de luz pase por ellas. Esto provoca dos efectos. El primero, una hoja puede ser más brillante cuando una luz aparece detrás de ella. Además, la luz puede revelar la estructura interna de la hoja. En este caso, se ven aspectos de la hoja que no tienen nada que ver con las superficies exteriores.

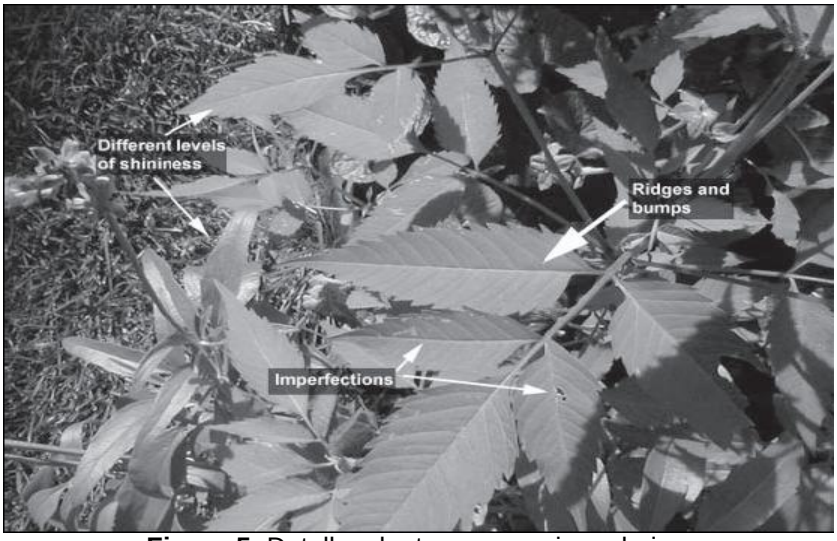

Figura 5: Detalles de transparencia en hojas.

Finalmente, muchos ambientes frondosos son densos y bloquean la luz. Un ejemplo común es el de un césped cubierto de hierba. El color de las hojas de hierba es bastante constante a lo largo de la hoja, pero aparece como si la hierba se hiciera más oscura a medida que se acerca al suelo. Esto se debe a

que las hojas colectivamente bloquean la luz y llega menos a sus partes inferiores. Si se observa arriba, el follaje podría parecer mucho más brillante que si se mira en las partes inferiores. Hay muchas más características de las plantas, muchos de los cuales son muy costosos a la hora de reenderear en tiempo real. [5]

Metales

Entre los metales pueden existir pequeñas diferencias en cuanto al modo en que reflejan la luz. Estas diferencias son causadas por el hecho de que diferentes metales tienen diferentes estructuras moleculares y por lo tanto reflejan la luz de forma diferente en un nivel microscópico. Estas estructuras pueden ocasionar que la superficie metálica refleje colores diferentes de un modo distinto. Encima de eso, los objetos diferentes que tienen acabados diferentes, hacen que también se reflejen de modo diferente. Por lo tanto, el cobre podría ser diferente al bronce. La Figura 6 muestra dos ejemplos de acero con distintos acabados. Uno está muy refinado, mientras el otro tiene una apariencia tosca. Ambos son esencialmente el mismo material con diferentes propiedades en sus superficies. [5]

Figura 6: Dos muestras de acero

Concreto

Probablemente vemos más materiales basados en minerales en nuestras vidas cotidianas que el metal. En las calles de una ciudad, se podrían ver cientos de tipos diferentes de concreto, ladrillo, y

piedra. Cada uno de estos materiales son muy diferente, pero hay algunas características de un nivel superior que son coherentes en todos ellos.

Si se toma una apariencia muy parecida a la acera, entonces se puede ver que, hay mucho del detalle visual que proviene de la rugosidad y la mezcla de materiales que conforman la superficie. Muchas de las áreas iluminadas y oscuras de la acera se deben a las grietas o las pequeñas sombras que una parte de la superficie lanza sobre la otra.

Por otra parte, el concreto es un material que es producido por la ligadura de diferentes compuestos entre los que se incluye piedras pequeñas. Por tanto, se puede tener manchas de color, densidad y mezcla de la cual depende el concreto. En la Figura 7 se muestra un ejemplo.

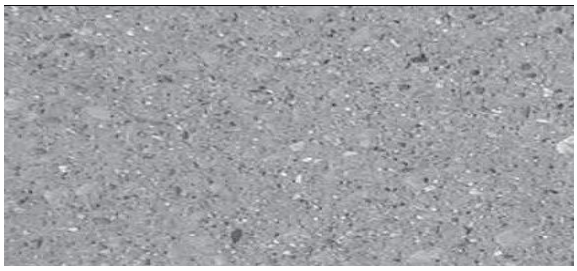

Figura 7: Fotografía de una acera**.**

De mismo modo, se podría tener pequeñas áreas donde el color es más o menos el mismo, pero el brillo se altera. En algunas ciudades, se puede ver que las aceras brillan donde están ubicadas las luces. Esto se debe a una concentración alta de material reflector en la mezcla. [5]

Piel

La piel es en su mayor parte uniforme en cuanto a su color, pero muchos de los otros detalles de la superficie se alteran. Algunas zonas podrían ser más rugosas que otras. Algunas zonas podrían ser brillantes. Además de ser visiblemente diferente, esos rasgos nos comunican algo en torno a la personalidad. La cara de un soldado podría estar llenas de hoyos y cicatrices, mientras que el rostro

de la reina podría ser muy suave. Un bajo grado de brillo podrían utilizarse para dar apariencia de suciedad.

Los detalles más pequeños como los poros podrían contribuir con efectos mayores. Por ejemplo, cuando usted hace expresiones faciales, las arrugas y los poros en la región de la cara se estiran de diferentes maneras.

Finalmente, las interacciones entre las diferentes capas translúcidas producen una dispersión de efectos de iluminación sobre su superficie. La luz penetra en la superficie de la piel, se dispersa, y entonces refleja hacia atrás. Esto contribuye en algún sentido en que la piel deba de tener profundidad. Si usted se para al lado de un maniquí con un tono de piel similar, encontrará que su piel es suave y que tiene una mirada profunda. Esto es debido al hecho de que su piel permite que penetre la luz, mientras que la piel plástica del maniquí no. Si hace la misma comparación en un museo de cera, verá menos la diferencia porque la cera es también translúcida (de ahí la elección de cera en primer lugar).

Los efectos de translucidez y la dispersión son visibles desde cualquier ángulo, pero son más evidentes cuando la luz está detrás de la piel. Esto es especialmente cierto si el objeto es delgado o la luz es muy brillante como se muestra en la Figura 8. [5]

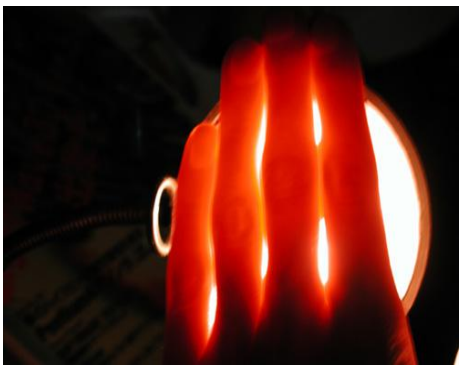

Figura 8: Efectos de translucidez

Pelo

Si se observan los pelos individualmente, resulta muy difícil sacar de entre un conjunto de propiedades cómo reenderearlos eficientemente. Cuando son vistos como una masa de pelos, algunos patrones se aclaran. Cuando muchos cabellos están en la misma dirección, como se muestra en Figura 9, se puede tratar como una superficie con muchas ranuras pequeñas. Esto causa un brillo muy característico.

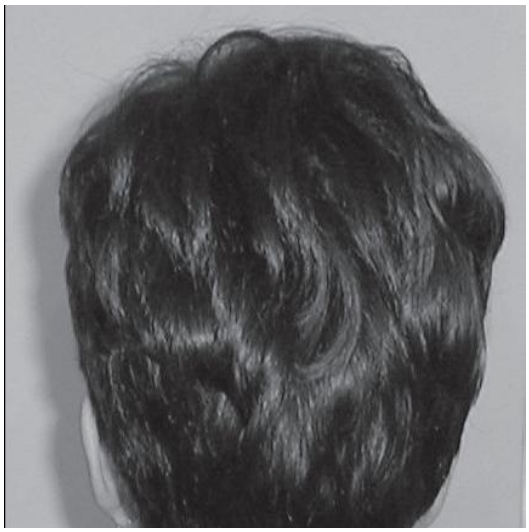

Figura 9: Cabello brillante.

Si se mira el cabello cuando te levantas en la mañana, verás que no hay mucho orden en él. En ese punto, el aspecto de su cabello depende de propiedades espaciales del cabello, la uniformidad del color, la cantidad de sombras y la reflexión entre los pelos entre otras. Como la piel, el pelo puede ser utilizado para comunicar características acerca del carácter. [5]

Materiales Transparentes

Los materiales transparentes son caracterizados por el hecho que se puede ver a través de ellos hasta cierto punto. La habilidad a crear objetos semitransparentes en escenas del 3D ha sido asequible por mucho tiempo, pero sólo recientemente ha sido posible crear otros efectos que acompañan la transparencia como por ejemplo la reflexión y la refracción.

Vidrio

El cristal ha formado parte de escenas virtuales, pero sólo en el sentido de que es dado como un material transparente. El cristal real exhibe una gama mucho más amplia de efectos. Primero, todos los materiales transparentes refractan la luz. La Refracción se vuelve mucho más evidente cuando se observan superficies curvas y objetos con un espesor variable. Si observa una copa, se verá que la refracción le da indicios visuales acerca de su espesor.

El vidrio transparente refleja el medio ambiente que existe a su alrededor. Sin embargo, la cantidad de luz que refleja desde el vidrio (vs. la cantidad que puede transmitir) depende de la inclinación de la luz. Cuando mira un vidrio curvo, verá que el vidrio es más reflexivo en los bordes que en los lados más cercanos a usted, como se muestra en la Figura10. Esto es debido a que sólo se esta viendo las reflexiones de luz que son notables en el vidrio en pocos ángulos.

Figura 10: Vidrio Curvo

Hay muchos tipos de cristales con efectos diferentes. Los espejos están hechos de un vidrio con un recubrimiento altamente reflector en el lado inverso. [5]

Agua

El agua también puede tener muchas cualidades, muchas de las cuales son causadas por la refracción. Por ejemplo, a causa de la refracción cuando sumergimos un objeto en el agua y lo observamos desde arriba da una apariencia de que están desplazados. Si tenemos un pescado dentro de un barril no seria fácil percibir el sonido que este emite. La refracción también provoca patrones cáusticos que se pueden percibir desde el fondo de una piscina. Las ondas en la superficie del agua reorientan luz en patrones diferentes basados en la profundidad del agua en diferentes puntos.

Estos mismos patrones de la superficie también reorientan la luz que se reflejó en la superficie del agua. Los ondulados sobre su superficie podrían decirnos que el viento es fuerte o que esta muy agitada.

Finalmente, el agua puede tener otros materiales o sustancias que recrean su vista. Puede tener un brillo aceitoso en su superficie o un tener un aspecto de suciedad lo que haría que los reflejos disminuyan. También puede llenarse de una materia que impide ver a través del agua, un río crecido tiene un aspecto muy similar a esto. [5]

1.2 GPU

GPU, acrónimo utilizado para abreviar *Graphics Processing Unit* (Unidad de Procesado de Gráficos), es un procesador dedicado exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo del procesador central de aplicaciones como los videojuegos y aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la CPU puede dedicarse a otros tipos de cálculo.

Una GPU está altamente segmentada, lo que indica que posee gran cantidad de unidades funcionales. Estas unidades funcionales se pueden dividir principalmente en dos: aquéllas que procesan vértices, y aquéllas que procesan píxeles. Por tanto, se establecen el vértice y el píxel como las principales unidades que manejan la GPU.

Adicionalmente, y no con menos importancia, se encuentra la memoria. Ésta se destaca por su rapidez, y juega un papel relevante a la hora de almacenar los resultados intermedios de las operaciones y las texturas que se utilicen.

Inicialmente, a la GPU le llega la información de la CPU en forma de vértices. El primer tratamiento que reciben estos vértices se realiza en el *vertex shader*. Aquí se realizan transformaciones como la rotación o el movimiento de las figuras. Tras esto, se define la parte de estos vértices que se va a ver, y los vértices se transforman en píxeles mediante el proceso de rasterización. Estas etapas no poseen una carga relevante para la GPU. [21]

1.3 Iluminación y Transformación (T & L)

Unos de los aspectos más importantes que deben tomar en cuenta para mejorar el realismo en los entornos virtuales, es el correcto manejo de luces y sombras. El principal logro deseado por los gráficos computarizados, es crear en el usuario la percepción de una realidad alternativa. Los efectos de iluminación mejoran el impacto en los entornos virtuales, ya que confieren mayor realismo a los detalles de una escena, debido a que el ojo humano es más sensible a los cambios de brillo que a los de color .Por ello, la iluminación es uno de los pasos más importantes en la representación gráfica de entornos virtuales, ya que sitúa las imágenes procesadas más cerca de nuestra percepción del mundo real.

En los gráficos 3D, la iluminación consta de dos componentes principales: la *iluminación difusa* y la *iluminación especula* (Ver Figura 11)*:*

- La iluminación difusa es aquella que incide sobre un objeto y se dispersa de igual forma en todas las direcciones, de tal manera que la luz reflejada no depende de la posición del observador. La difusa también sirve para calcular el brillo de los objetos en una escena 3D.
- La iluminación especular, a diferencia de la difusa depende de la posición del espectador, la dirección de la luz y la orientación del triángulo en el que se refleja; la iluminación especular muestra las propiedades reflectantes de los objetos y permite crear efectos de reflejos y resplandores.

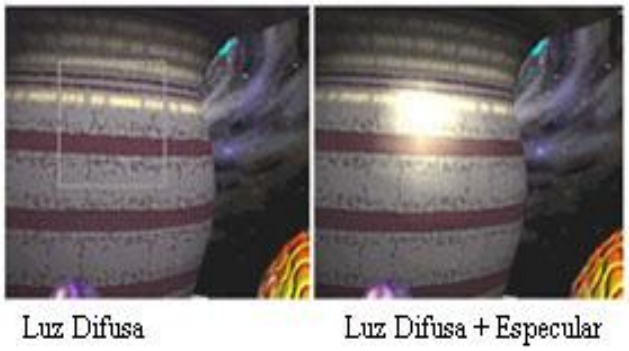

Figura 11: Tipos de Iluminación.

Los reflejos creados con luz difusa se mueven a través de los objetos cuando el observador o el objeto se mueven con respecto a la fuente de luz, por lo que deben ser calculados en tiempo real y ser dinámicos a los cambios de posición en el entorno. La iluminación especular se usa para crear efectos de desplazamiento y para dar la apariencia a los distintos materiales de los objetos o los reflejos en el mármol pulido que serian distintos a los de un mármol sin pulir. El pulido no afecta el color ni la textura del mármol, pero si en el modo en que la luz se refleja en él.

Es tal la importancia que se le ha dado a la iluminación, que la mayoría de las tarjetas gráficas actuales incluyen, en su GPU, unidades (motores) de procesamiento especialmente dedicadas a la iluminación y transformación (T & L).

La iluminación y el sombreado dependen del tipo y posición de la fuente de luz, así como de otros factores adicionales, como son: posición de los objetos, complejidad de los objetos, etc., debido a que las luces y sombras generalmente son dinámicas.

La transformación e iluminación forman parte de uno de los primeros y más importantes pasos en el flujo de procesamiento gráfico 3D de una GPU. Durante este proceso es necesario ejecutar un conjunto concreto de instrucciones miles de millones de veces por segundo para procesar una escena. Es tanto el poder de procesamiento que se necesita para calcular las transformaciones e iluminaciones

de los entornos, que se optó por incluir motores de transformaciones e iluminación en las GPU, cuyas funciones son:

- ✓ Transformaciones matemáticas entre los diferentes sistemas de coordenadas o "espacios en los entornos (Ver Figura 12), los cuales pueden ser:
  - Espacio del mundo, el cual contiene todos los objetos 3D del entorno.
  - Espacio del ojo, el cual se usa para iluminar y seleccionar los objetos.
  - Espacio de la pantalla, que es la vista que se muestra al usuario y se almacena en el búfer de tramas gráficas.

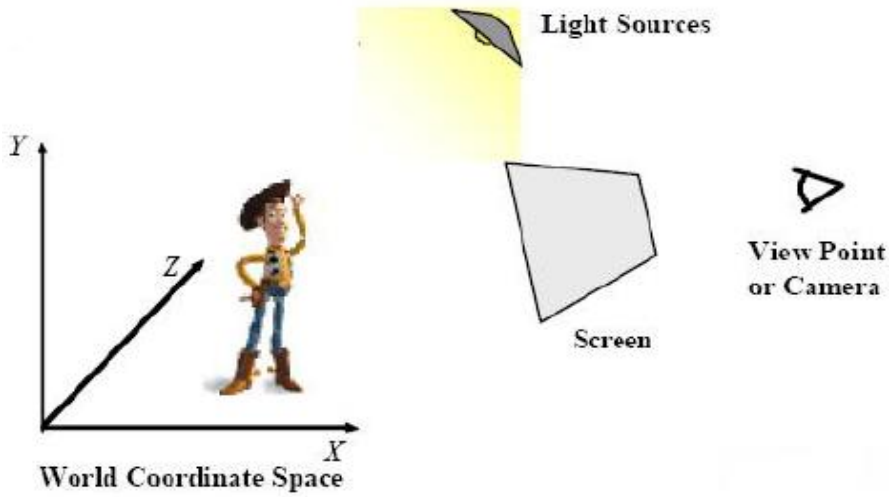

Figura 12: Tipos de coordenadas ("espacios")

- ✓ Cálculos de luces, los cuales realiza el motor de iluminación mediante el cálculo de los vectores de distancia entre la luces y los objetos 3D del entorno, y entre los objetos y los ojos del usuario; como ya se sabe, los vectores contienen información sobre la dirección y la distancia, datos que son obtenidos por el motor de iluminación .

La ventaja que se tiene al usar motores T&L, es que todos los procesos que anteriormente se realizaban en la Unidad Central de Procesos(CPU), ahora se llevan a cabo de manera más rápida y eficiente en la GPU, liberando así al microprocesador de esta tarea para que pueda encargarse de otras más especifica. [8]

1.4 Implementación de Luces usando Shader

Con la aparición de los procesadores gráficos (GPU) surge la posibilidad de programar dicho hardware como si fuera una CPU, mediante pequeñas rutinas en ensamblador que se denominan *Shaders*, que reemplazan una sección del hardware de vídeo que típicamente se denomina *pipeline* de función fija (*Fixed-Function Pipeline* o FFP), esto quiere decir que se sustituye tanto la transformación de vértices como la iluminación y mapeado de textura “fijos” del *hardware* (la no programable) por una forma programable. Igual que sucediera con las CPUs, estas pequeñas rutinas del lenguaje ensamblador se va sustituyendo por sus equivalentes de alto nivel que encapsulan adecuadamente el hardware subyacente, como GLSL, CG o HLSL.

A partir de aquí, con el fin de incrementar aun más el realismo en los entornos virtuales y videojuegos, se desarrollan dos nuevas tecnologías: Vertex Shaders y Pixel Shaders, las cuales podrían representar una gran cantidad de efectos visuales de alto impacto en los usuarios. Microsoft implementó la capacidad de utilizar estas nuevas tecnologías en su API DirectX 8.0 y posteriores; y fueron NVIDIA, ATI, MATROX, entre otras compañías de tarjetas gráficas, las que les dieron soporte por hardware.

Mediante la tarjeta de aceleración gráfica una computadora de escritorio es capaz de recrear una gran cantidad de impresionantes efectos visuales en tiempo real, lo cual no era posible anteriormente. Estos efectos van desde niebla y efectos de movimiento, hasta agua, sombras dinámicas, pelo, rugosidad, etc. [13]

1.4.1 Vertex Shaders

Vertex Shaders es una función de procesamiento gráfico que genera efectos especiales en los objetos en una escena 3D, permitiendo a los programadores ajustar dichos efectos mediante el uso directo de nuevas instrucciones de software en motores de sombreado por vértice, las cuales están orientados a realizar operaciones matemáticas precisamente con los datos almacenados en los vértices de los objetos. Estos datos pueden ser coordenadas x, y, z, color, canal alfa, textura, características de iluminación entre otros.

El Vertex Shaders programable es esencialmente una extensión de los motores de iluminación y trasformación, los cuales nos permiten realizar un número casi infinito de efectos visuales de alto detalle, sin afectar las velocidades de la trama en tiempo real. Todos los cálculos de sombreado por vértice programable son realizados por la GPU, lo que permite que la CPU se ocupe de realizar otros cálculos matemáticos más específicos. [13]

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.

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. 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. [7]

1.4.2 Pixel Shaders

Pixel Shaders también conocido como *Fragment Shaders* es otra técnica que se inventó e implementó en las tarjetas aceleradoras gráficas con el fin de aumentar el realismo en los entornos virtuales. Esta técnica permite crear efectos especiales de alto realismo mediante pequeños programas (de bajo nivel) que manipulan ciertos píxeles en la escena; algo que habría resultado imposible de conseguir con el hardware 3D anterior.

Muchos efectos creados con este tipo de tecnología hacen uso de más de una tecnología aplicada a un objeto, por lo que es necesario que el hardware grafico soporte multitexturizado, el cual permite aplicar, en una sola pasada, varia texturas a un mismo objeto.

Los motores nfiniteFX (llamado así por el numero casi infinito de efectos posibles de crear) de NVIDIA y SMART SHADERS de ATI, cuenta con la capacidad de gestionar 4 o más texturas en una sola pasada, lo que posibilita la creación de efectos *pixel* a *pixel* anteriormente imposible de lograr en plataformas convencionales. La capacidad de aplicar 4 texturas en una sola pasada aumenta el rendimiento en el proceso de *Rendering*, ya que al hacerlo en varias pasadas, este se vuelve más lento.

Un efecto visual de alto impacto creado mediante el uso de los *pixel shaders* y la capacidad de multitexturizado, es un efecto *Bump Mapping* o mapeado de relieve, el cual crea la ilusión de geometría adicional que puede observarse en los relieves de los pisos, paredes rugosas, ladrillos, arrugas en la piel de animales u objetos, etc. [13]

Iluminación Per-Pixel

Contrario a la iluminación "*per-vertex*", este método ejecuta las ecuaciones directamente en el *pixel*. 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 *pixel*.

La iluminación "*per-píxel"*, no introduce errores de interpolación ni brillos en los bordes de los polígonos. Sin embargo, 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. [7]

1.5 Algoritmos de Iluminación

1.5.1 Iluminación con Esferas Harmónicas

Iluminación por Esferas Harmónicas (*Spherical Harmonics Lighting, SH-L*) es una técnica de renderizado en tiempo real que utiliza un paso de pre-proceso, la proyección de las luces, del modelo y de la función de transferencia en la base de SH para hacer las escenas realistas usando cualquier tipo de fuente de luz. Se utiliza primeramente para reproducir la 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. [4]

1.5.2 Modelo de Phong

Phong Bli-Tong desarrolló un modelo de iluminación local para reflectores imperfectos. El modelo supone que la máxima reflectancia especular ocurre cuando α es cero y decrece rápidamente conforme aumenta α. Esta caída rápida se aproxima por $\cos^{n}\alpha$, donde n es el *exponente de reflexión especular* del material. En la Figura 13 se representa esquemáticamente este fenómeno.

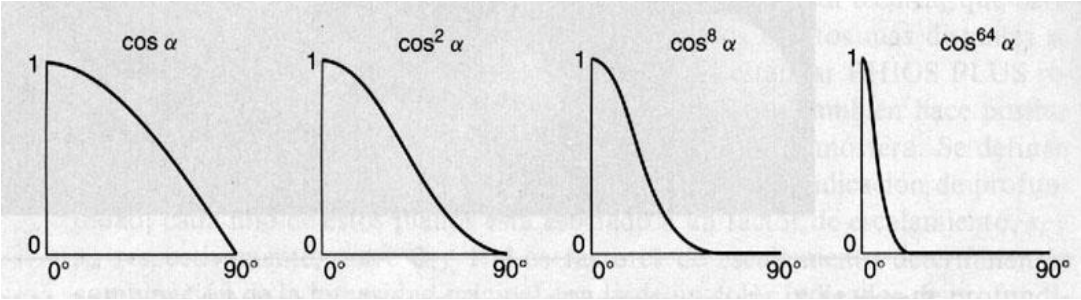

Figura 13: Exponente de reflexión especular.

Phong para iluminación local puede producir cierto grado de realismo en objetos tridimensionales combinando tres elementos: luz difusa, especular y ambiental para cada punto en una superficie. Además emplea algunas suposiciones, como por ejemplo que todas las luces son puntuales, considerada solo una superficie geométrica, solo para modelos locales difusos y especulares, que el color especular es el mismo que el color de luz y el ambiente es constante y global.[17]

1.5.3 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.

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 1: Súper elipse

Una súper elipse es una función que varía su forma desde una elipse a un rectángulo, basado en valores de parámetros redondeados.

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, algo así como lo que se muestra en la Figura 14. [13]

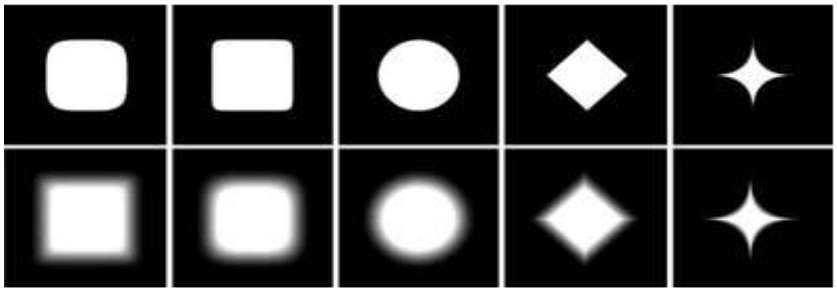

Figura 14: Überlight

1.5.4 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 representa el color reflejado del ambiente. Esto logra simular una iluminación por irradiación del color. (Figura 15).

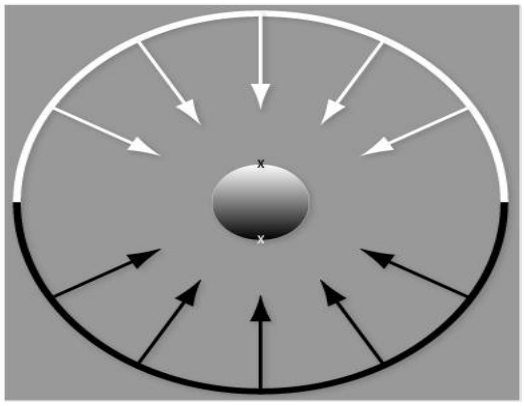

Figura 15: Iluminación de Hemisferios

Posteriormente aplica el mapeado *Bump*, el cual, utilizando una textura de normales, simula irregularidades en la superficie inexistentes en la geometría. 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. [13]

1.6 Algoritmos de Sombreado

La generación de sombras es un problema clásico de los gráficos por computador. Las sombras son básicas para la percepción del mundo real, pues nos ayudan a percibir la profundidad en las imágenes generadas por computador y además nos dan una referencia para entender la posición relativa de los objetos y de las fuentes de luz que pueden estar iluminado una determinada escena.

Una sombra siempre va ligada a la luz que la produce, aunque en 3D podamos trabajar cada concepto por separado y atribuir a un tipo de luz una serie de características de sombreado. Entre las posibilidades que se encuentran tenemos el sombreado duro o *hard*, cuyas sombras plasmadas por una fuente de luz tendrán una definición muy alta en sus bordes .Mientras que las sombras de tipo *sotf* o suaves, recrean bordes mas difuminados, con un grado de suavidad diferente según el tamaño del mapa de sombras.

En la actualidad existen múltiples métodos que son capaces de generar sombras en tiempo real, sin embargo no pasa lo mismo con las penumbras (Ver Figura 16). La mayoría de los métodos que generan penumbras no son capaces de generarlas a una velocidad suficiente para que se puedan aplicar en entornos muy complejos en tiempo real.

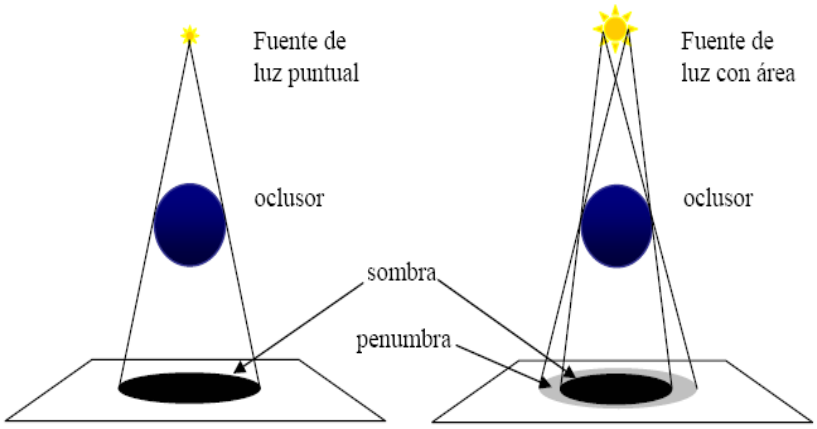

Figura 16: Zonas de sombra y penumbra.

Existen diversas técnicas para conseguir sombrados realistas. A grandes rasgos, podemos dividir las técnicas de generación de sombras en dos grandes grupos, las técnicas clásicas y técnicas que explotan la potencia del hardware gráfico existente. [7]

1.6.1 Técnicas Clásicas

Hay dos técnicas clásicas que permiten la generación de penumbras: el trazado de rayos y la radiosidad. La técnica de trazado de rayos consiste en lanzar rayos desde la posición del observador a la escena, y calcular la intersección y subsecuentes rebotes de los rayos con los objetos de la escena. Para determinar si un punto visible de la escena está o no en sombra lo que se hace es trazar un rayo hacia cada fuente de luz .Si el rayo llega sin ninguna intersección a ningún objeto, el punto está iluminado, de lo contrario esta en sombra.

Sin embargo esta técnica es muy costosa debido a la complejidad del algoritmo de trazado de rayos. Para cada rayo, se tiene que intentar intersecarlo con cada uno de los objetos de la escena para ver si el punto de intersección está entre el punto visible y la fuente de luz como se puede observar en la Figura 17.

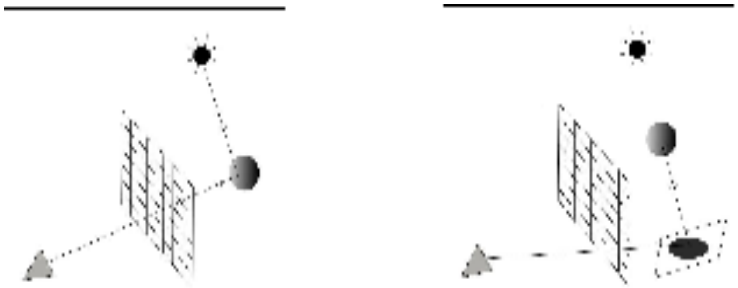

Figura 17: Técnica de trazado de rayos.

Otro conjunto de técnicas muy conocido y usado son los métodos basados en la radiosidad. Este conjunto de métodos usa una simulación basada en la física del comportamiento de la luz en una escena cerrada. Las superficies de la escena se subdividen en parches(*o patches en inglés*), donde cada patch tiene un área, un coeficiente difuso y emite una cierta cantidad de energía o radiancia. Para cada patch, se calculan las interacciones con todos los otros patches de la escena usando la denominada ecuación de la radiosidad, en la Figura 18 se representa la subdivisión e interacción entre los parches. [7]

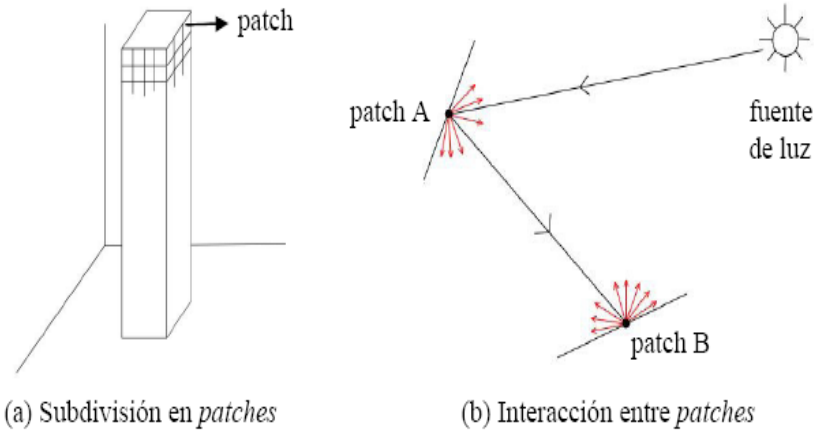

Figura 18: Método de Iluminación usando radiosidad.

1.6.2 Técnicas con aceleración por hardware

Existen técnicas de muy diversa índole que usan la potencia de las tarjetas gráficas para simular penumbras. La mayoría de los métodos están basados en dos técnicas generales, el mapa de sombra (*shadow map*) y el volumen de sombra (shadow volume).El primero es una técnica de las que llamamos que trabaja en especio imagen mientras que el segundo trabaja en especio objeto.

1.6.3 Mapeo de Sombra

A grandes rasgos, el mapa de sombra consiste en proyectar la escena desde el punto de vista de la fuente (o fuentes) de luz y guardar los valores del buffer de profundidad .Una vez hecho esto se procede a proyectar la escena desde el punto de vista del observador y se transforman las coordenadas de los puntos visibles a las coordenadas equivalentes cuando se tiene como punto de vista la fuente de luz. En este punto se compara el valor de profundidad y si son iguales, el punto está iluminado, mientras que si el segundo valor es mayor quiere decir que el punto está en sombra. En la Figura 19 se representa lo anteriormente explicado. [6]

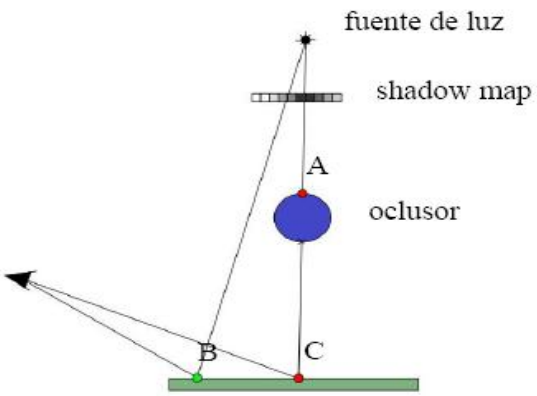

Figura 19: Proyección de la escena desde la fuente de luz y del observados.

Aparte del mapeo de sombras existen otras variantes mejoradas o adaptadas para generar sombras suaves, como:

- Mapa de sombras de segunda profundidad (*second-depth shadow mapping*). Una variante del *mapeo de sombras* básico, que considera la profundidad del primer y segundo objeto más cercano a la vista de la luz en lugar de sólo la profundidad del primero, con objetivo de evitar ciertos defectos antiestéticos derivados de la imprecisión de la solución.

- Mapas de sombras para luces lineales (*Shadow maps for linear lights*).Una extensión del mapa de sombras básico para producir sombras suaves, restringidas a las proyectadas por luces lineales (en forma de varilla o tubo).

- Mapas de atenuación capas (*layered attenuation maps*) .Consiste en tomar varias luces puntuales sobre la superficie de la luz no puntual, producir varios mapas de sombras, y combinarlos en una imagen de profundidad en capas (*layered depth image*), sobre la cual se proyectan los píxeles y se selecciona el nivel de iluminación apropiado, pudiendo así obtener sombras suaves.

1.6.4 Volumen de Sombra

Este método consiste básicamente en generar el volumen que engloba las zonas de la escena donde no llega la luz. Todo lo que esté en el volumen de sombra estará a oscuras. Para obtener el volumen de sombra se necesita obtener la silueta del objeto u objetos oclusores desde el punto de vista de la fuente de luz. Una vez obtenida la silueta, se hace una extrusión de la misma siguiendo la dirección de la recta que va desde la fuente de luz al punto de la silueta correspondiente. El algoritmo que se usa generalmente para la visualización usa el *stencil buffer* para los cálculos, donde se generan los valores pintando primero las caras frontales del volumen de sombra, si se pasa el test de profundidad (son visibles), se incrementa el valor correspondiente del *stencil buffer.* Luego se hace lo mismo para las caras posteriores del volumen de sombra decrementando en este caso el valor del *stencil buffer* si se pasa el test de profundidad. Después se proyecta en la escena teniendo en cuenta que cuando el *stencil buffer* hay un valor diferente de cero, es un punto de sombra (Ver Figura 20).

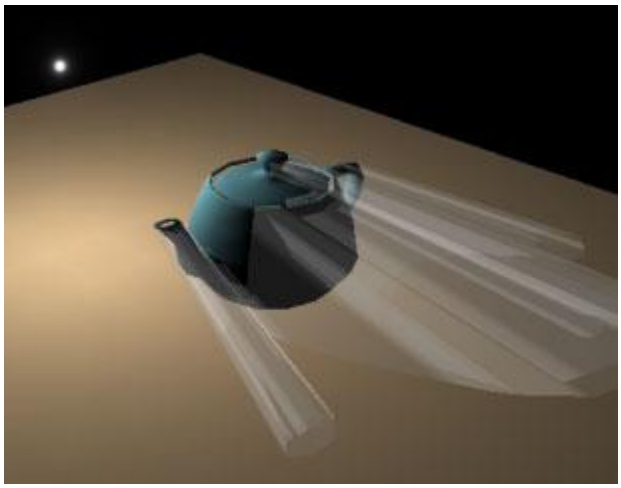

Figura 20: Volumen de Sombra.

Uno de los problemas principales de los algoritmos basados en volumen de sombra es la generación de las siluetas de los objetos. Existen diversos métodos, pero suelen ser dependientes del número de polígonos de la escena y eso es problemático para escenas muy grandes. Además para el cálculo de siluetas los objetos con agujeros también representan problemas. [10]

Conclusiones

A lo largo de este capítulo, como base para el estudio y entendimiento del tema sobre el se realizara este proyecto, se presentaron una serie de conceptos básicos para el posterior entendimiento de lo expuesto, tales como Iluminación *Per-Vertex* y *Per-Píxel*, GPU. Además se describieron las principales técnicas y tendencias actuales más utilizadas para la implementación de las luces y sombras dinámicas en SRV, alguna de sus características, ventajas y desventajas.

También se profundizo en el estudio del lenguaje Cg que se basa principalmente en estándares y diseñado para aprovechar los enormes avances experimentados por los procesadores gráficos programables que soportan DirectX y OpenGL.

2

Capítulo 2 Descripción de la Solución Propuesta

Introducción

En este capítulo se propone los algoritmos que más se adecuan para dar solución a los objetivos planteados en el trabajo, se quiere incorporar más de una fuente de luz en la escena donde el algoritmo seleccionado para ello lo permite, así como también proyectar las sombras respectivas, aunque cada algoritmo tiene sus ventajas y desventajas cada uno funcionará mejor en una situación u otra.

2.1 Algoritmo de Iluminación

En el anterior capítulo se trataron algunos de los algoritmos más utilizados para la iluminación dinámica en entornos 3D. A partir del estudio realizado, los algoritmos que más se adaptan a las necesidades planteadas en la introducción de este trabajo referentes a la representación de las luces dinámicas es el modelo de Phong y para modelar la iluminación global se utiliza Esferas Harmónicas (SH).

2.1.1 Esferas Harmónicas (Spherical Harmonics Lighting)

La iluminación con Esferas Harmónicas (SH) 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 SH 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 2: Reflexión difusa de SH.

Donde las constantes c1-c5 se obtuvieron a partir del resultado de la derivación de esta fórmula. L representa a los 9 coeficientes harmónicos que fueron obtenidos a partir de imágenes de alta resolución. [5]

En la Figura 21 se representan los más usados actualmente, aunque existen otros más. Las variables x, y, z son las coordenadas de la superficie normalizada en el punto que debe ser sombreado.

Coefficient	Old Town Square			Grace Cathedral		
L00	.87	.88	.86	.79	.44	.54
L1m1	.18	.25	.31	.39	.35	.60
L10	.03	.04	.04	-.34	-.18	-.27
L11	-.00	-.03	-.05	-.29	-.06	.01
L2m2	-.12	-.12	-.12	-.11	-.05	-.12
L2m1	.00.	.00	.01	-.26	-.22	-.47
L20	-.03	-.02	-.02	-.16	-.09	-.15
L21	-.08	-.09	-.09	.56	.21	.14
L22	-.16	-.19	-.22	.21	-.05	-.30

Figura 21: Coeficientes Harmónicos

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. [5]

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

La idea básica es tratar de evaluar una aproximación de la ecuación de renderizado en cada vértice de la geometría de la escena por la proyección de sus diferentes componentes sobre la base de la esfera harmónica durante una etapa previa al proceso. El término geométrico y de la función de visibilidad en la ecuación de renderizado se puede simplificar o aproximar de tres maneras diferentes.

En la primera aproximación, el término geométrico se reduce a su forma más simple: al término del coseno o el producto escalar entre la normal de la superficie en un vértice dado y la dirección de la luz. Esto es exactamente lo mismo que el estándar de iluminación difusa usando la *Ley de Lambert's*. Sólo

que en este caso, no hay necesidad de manipular la normal de la superficie ya que se codifica en los coeficientes SH. Esta primera técnica se la conoce como iluminación difusa. SH sin sombreado (*SH Diffuse unshadowed lighting*) y puede generar imágenes como la que se muestra en la Figura 22:

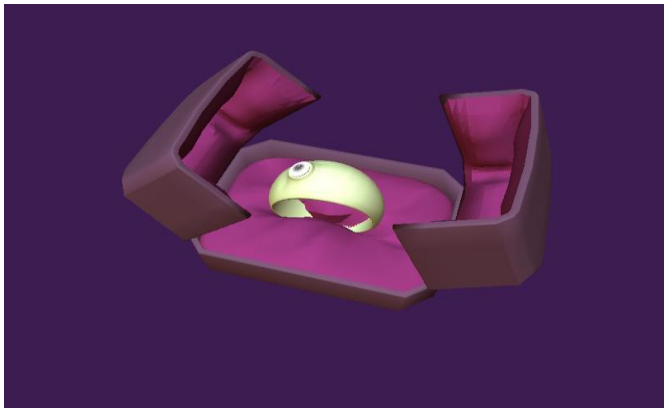

Figura 22: Iluminación Difusa SH sin sombreado**.**

En la segunda forma, el término de visibilidad se agrega para determinar un sombreado de vértices. Esto puede lograrse fácilmente mediante el uso de un raytracer. Esto permite generar fácilmente sombras realistas en la misma escena, en el costo del tiempo extra de cálculo. Esta segunda técnica se la conoce como iluminación Difusa SH con sombreado (*SH Diffuse Shadowed Lighting*) y donde se pueden generar imágenes como la que se muestra en la Figura 23:

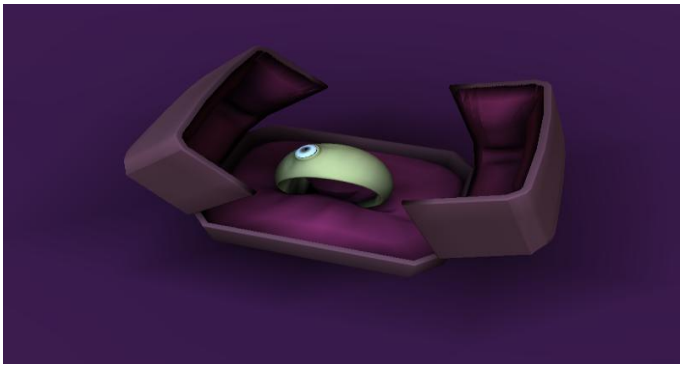

Figura 23: Iluminación Difusa SH con sombreado.

En la última aproximación, la iluminación indirecta procedentes de los otros objetos en la escena es tomada en cuenta. El paso de pre-procesado en este caso se debe tener en cuenta la iluminación directa e indirecta, en este caso proporciona una manera de calcular la iluminación global de la escena. Esta tercera técnica se la conoce como Iluminación Difusa SH Inter-Reflectada (*SH Diffuse Inter-Reflected Lighting*).

Una vez que todos los coeficientes SH que se han generado para los tres modos, en tiempo de ejecución, la ecuación de renderizado se puede ir reconstruyendo a partir de estos diferentes coeficientes SH simplemente añadiéndolos y multiplicándolos juntos. La radianza emitida en cada vértice puede ser calculada.

Si usando el sombreado de Gouraud (es decir: la interpolación del color de un triángulo sobre los colores en cada vértice de la misma), entonces es fácil y rápido de renderizar la escena completa en tiempo real. La imagen de la Figura 24 es una comparación entre diferentes modelos de la iluminación aplicada a la misma escena: *standard OpenGL lighting (Phong shading)*, 3DS Max *scanline rendering*, modo SH D*iffuse* U*nshadowed* utilizando una luz de prueba HDRI, modo SH *Diffuse Shadowed* utilizando la misma luz de prueba HDRI:

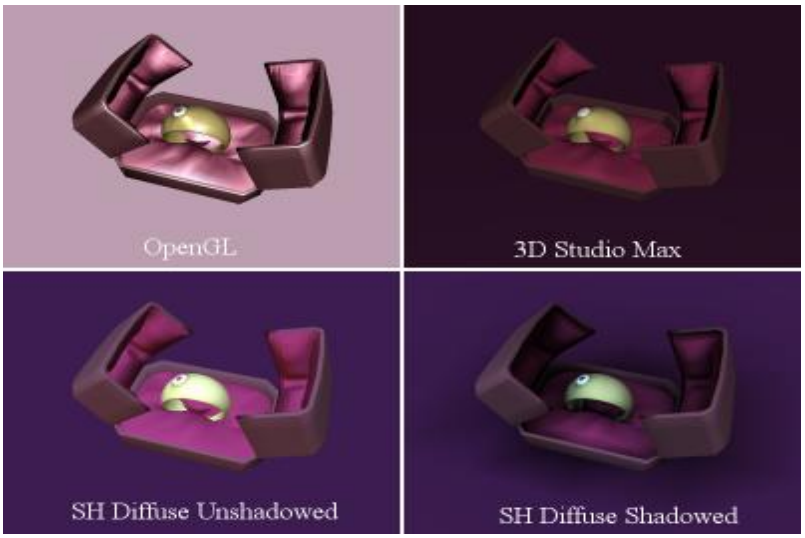

Figura 24: Diferentes modelos de la iluminación aplicada a la misma escena.

SH 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 global. [5]

2.1.2 Phong

El modelo de Phong como se dice en el epígrafe anterior vincula tres componentes de la iluminación: luz ambiental, luz difusa y luz especular. A continuación se propone un estudio mas profundo de cada uno de ellos para un mejor entendimiento del modelo como tal.

Luz Ambiental

Corresponde al modelo en el cual cada objeto se presenta con una intensidad intrínseca. Se puede considerar este modelo, que no tiene una fuente de luz externa, como la descripción de un mundo ligeramente irreal de objetos no reflejantes y autoluminosos. En este caso cada objeto aparece como una silueta monocromática.

Un modelo de iluminación se puede expresar con una ecuación de iluminación de variables asociadas con el punto en el objeto que se sombrea. La ecuación de iluminación que expresa este sencillo modelo es: $I = K_i$; donde I es la intensidad resultante y el coeficiente K_i es la intensidad intrínseca del objeto.

Como esta ecuación de iluminación no contiene términos que dependan de la posición del punto que se sombrea, se puede evaluar I una vez por cada objeto.

El proceso de evaluación de la ecuación de iluminación en uno o más puntos de un objeto se conoce como *Iluminación del Objeto*.

Imaginemos ahora que un lugar de *Autoluminosidad* hay una fuente luminosa difusa no direccional, producto de reflexiones múltiples de la luz en las superficies presente en el ambiente. Esto se conoce como *luz ambiental*. Si suponemos que la luz ambiental afecta de la misma forma a todas las superficies desde todas las direcciones, nuestra ecuación se convierte en:

$$I = I_a * K_a$$

Ecuación 3: Componente Ambiental

I_a es la intensidad de la luz ambiental (constante para todos los objetos) y K_a es la cantidad de luz ambiental reflejada por la superficie de un objeto, su valor está entre 0 y 1 y se conoce como el *Coeficiente de Reflexión Ambiental*. Este valor es una propiedad material no una propiedad física

Luz Difusa

En este caso se requiere una fuente luminosa puntual cuyos rayos emanan uniformemente en todas las direcciones a partir de un único punto. La brillantez de un objeto varia de una parte a otra, dependiendo de la dirección y la distancia de éste con respecto a la fuente luminosa (Ver Figura 25).

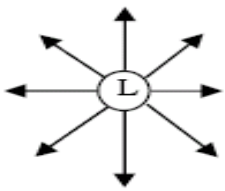

Figura 25: Iluminación difusa.

$$I_d = I_i k_d \cos\theta$$
$$\cos\theta = (N \bullet L)$$

Ecuación 4: Componente Difuso de Phong

Donde:

N y L son vectores normalizados, N normal a la superficie en el punto a calcular y L apuntando a la fuente de luz.

I_i es la intensidad de la fuente luminosa puntual.

K_d es el *coeficiente de reflexión difusa* del material, el cual es una constante entre 0 y 1 y varía de un material a otro.

El ángulo θ debe estar entre 0º y 90º para que tenga efecto directo en el punto sombreado (superficie *autocluyente*).

En la Figura 26 de izquierda a derecha: K_d toma los valores 0.4, 0.55, 0.7, 0.85 y 1.0.

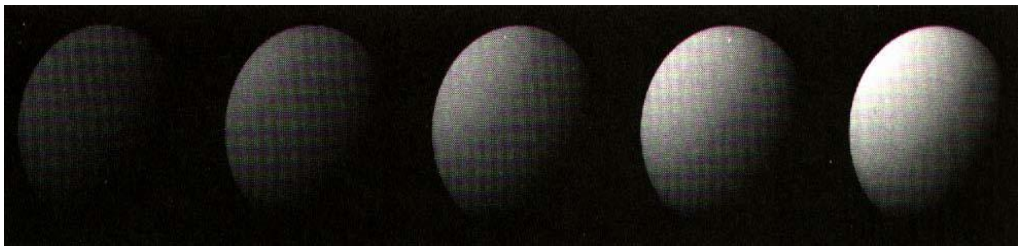

Figura 26: Esferas iluminadas usando modelo de reflexión difusa.

Para múltiples luces:

$$I_d = k_d \sum_n I_{i,n}(L_n \cdot N)$$

Ecuación 5: Múltiples fuentes de luces

Donde n representa el número de fuentes de luces.

Luz Especular

La reflexión especular se puede observar en cualquier superficie brillante. Por ejemplo, al iluminar una manzana con una luz blanca brillante: el punto de máximo brillo (*high light*) es ocasionado por la reflexión especular, mientras que la luz reflejada del resto de la manzana es el resultado de la luz difusa. En el punto de máximo brillo la manzana aparece blanca, el color de la luz incidente.

En un espejo perfecto α = 0, solo en la dirección del vector R se puede ver la reflexión de la luz. En este caso el vector V representa la dirección del observador (Ver Figura 27).

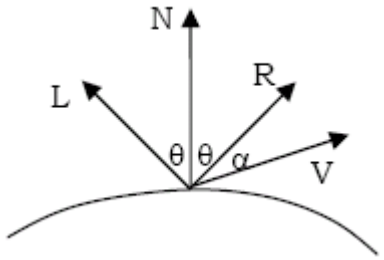

Figura 27: Representación de los vectores L, N, R y V

$$I_s = I_i k_s \cos^n \Omega = I_i k_s (R \cdot V)^n$$

Ecuación 6: Componente Especular

Donde n indica la refractividad de la superficie. Infinito significará un espejo perfecto, omega toma valor cero.

Omega es el ángulo entre el espejo y el visor. R es la dirección de la luz especular y V es el actual vector.

I_i es la intensidad de la fuente luminosa puntual, que es la misma que de la Ecuación 3.

Combinando los componentes de las ecuaciones 2, 3 y 5 tenemos:

$$I = I_a k_a + I_i (k_d (L \cdot N) + k_s (R \cdot V)^n)$$

Ecuación 7: Modelo de Phong

Esta ecuación puede ser presentada de forma gráfica como se puede ver en la Figura 28, así tendríamos que:

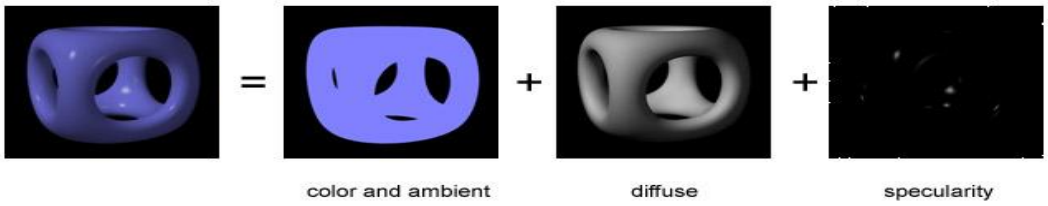

Figura 28: Explicación gráfica de la fórmula de Phong

Phong es un modelo que genera brillos convincentes en superficies brillantes normales, suaviza las aristas entre caras, puede representar con precisión mapas de relieve, opacidad, brillo, reflexión y efectos especulares. Todas estas cualidades permiten que este algoritmo muestre resultados bastantes certeros y ofrezca una buena percepción al usuario. El shader que lo implementa es sencillo, fácil de entender y de consumo de memoria bajo. De aquí se deriva la elección de este modelo para representar las luces dinámicas. [17]

2.2 Algoritmo de Sombreado

En este epígrafe se dará a conocer el algoritmo que más se adapta a las necesidades planteadas en el presente trabajo: Mapeo de Sombras (SM) (*Shadows Mapping*). 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.

El algoritmo involucra dos o tres pasos. El primer paso dibuja el *shadow map* desde el punto de vista de la luz. El segundo paso dibuja la escena desde el punto de vista de la cámara iluminada normalmente, pero al dibujar cada píxel se determina si esta sombreado o no usando para esto el *shadow map* obtenido en el primer paso. El tercer paso, el cual pude ser incorporado en el segundo, redibuja la escena afectando solamente aquellas áreas sombradas determinadas en el segundo paso.

1. **Paso uno**

   El paso uno involucra dibujar la escena desde el punto de vista de la luz. Si se usa una luz direccional (como el sol) se debe utilizar proyección ortográfica, pero si se usa un punto de luz se debe usar proyección de perspectiva. Primero se debe desactivar la escritura a todos los

buffers excepto al *Depth buffer*. Adicionalmente, se puede elegir que objetos proyectarán sombras para ahorrase tiempo de rendering. Se aplica un *depth offset* para evitar el *Z-figthing* entre el *shadow map* y la superficie que recibe la sombra. Cuando el rendering de los "objetos sombreadores" (*shadow casters*) esta completo, el depth buffer es almacenado como un mapa de textura (*texture map*). Esta textura es el mapa de sombra (*shadow map*) que será usado por el resto del proceso.

2. **Paso dos**

   El segundo paso dibuja la escena iluminada normalmente desde el punto de vista de la cámara. Por cada vértice rendereado, se necesitan las coordenadas que pondrá la textura de profundidad (*depth texture*) en la escena, exactamente como aparecen desde el punto de vista de la luz y su campo de visión. Esto es hecho por transformar los vértices a través de una serie de matrices multiplicadas. Las coordenadas para el mapa de profundidad o *depth map*(s, t, r, q) son determinadas al multiplicar el vértice (x, y, z, w) por el producto de las matrices de proyección y de vista (*modelview*) de la luz (en el momento del render del *depth map*) y escalados por una matriz (para dejarlo en el rango 0 - 1).

El coste de crear el mapa de sombra es lineal con el número de primitivas a renderizar y el tiempo de acceso es constante .La calidad depende de la resolución en píxeles de la textura en la que se guarda la sombra, y la de la precisión numérica del z-buffer, del cual sólo se pueden guardar 8 o 16 bits al introducir esa información en el canal de color de la textura .A menos que la arquitectura soporte los mapas de sombras directamente.

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. 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.

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. Debido a esto, SM es la técnica que más se adecua para dar respuesta a los objetivos planteados en el capítulo anterior referente a la generación de sombras dinámicas eficientes para más de una fuente de luz.

Esta técnica es una de las más usadas en la actualidad para aplicaciones gráficas gracias a las características que posee, aunque para muchas fuentes de luz puede resultar costoso el proceso de render. [1]

2.3 Lenguajes de Programación

2.3.1 C++

El módulo se implementará en C++ debido a que la herramienta al que se acoplará este módulo está 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).*

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.

Existen principalmente tres lenguajes que se utilizan para desarrollar aplicaciones gráficas en 3D: Lenguaje Ensamblador, C y C++, por ser los que con más velocidad ejecutan el código (menor costo

de ejecución del programa). A estos se ha unido recientemente el Java como una opción para el desarrollo de este tipo de aplicaciones. [15]

Es decisión de la entidad cliente, implementar este proyecto mediante el lenguaje C++, que ha estado en su línea de trabajo con magníficos resultados. Es el lenguaje en el que se tiene mayor experiencia por parte de los desarrolladores del sistema que ocupa esta investigación.

Si se estudian las características de este lenguaje, se podrá apreciar lo acertado de la elección, dado que C++ es un lenguaje de programación de propósito general, especialmente indicado para la programación de sistemas por su flexibilidad y potencia. Es uno de los más utilizados por la comunidad de desarrollo de *software,* incluyendo la programación gráfica.

C++ es la evolución de C adaptada a la programación orientada a objetos. Tiene algunas cuestiones más pulidas como el control más estricto en el manejo de datos, y otras características que ayudan a la programación libre de errores.

En general puede llegar a ser un lenguaje tan rápido como C (el mas rápido después del lenguaje ensamblador), sin embargo, si se maneja herencia múltiple, funciones virtuales y polimorfismo en forma inadecuada, o se accede mucho en niveles de profundidad en la llamadas a objetos (Objeto1, Objeto2, Objeto3…), puede llegar a hacerse más lento, lo cual no es conveniente para una aplicación en tiempo real.

2.3.2 GLSL (*OpenGL Shading Language*)

Para la programación de los shader se escogió GLSL que es un lenguaje de alto nivel diseñado específicamente por el entorno OpenGL. 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 fotorealistas 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. [12]

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.

Aunque el lenguaje utilizado para la programación de los *shaders* fue GLS a partir de que el módulo solo será implementado utilizando la biblioteca gráfica OpenGL y que es actualmente es muy utilizado para el desarrollo de aplicaciones gráficas, existen otros como HLSL (*High Level Shading Language*) de DirectX y Cg (*C for Graphic*) de NVIDIA que son muy parecidos a este en cuanto al funcionamiento.

2.4 Lenguajes de Modelado

Para modelar el análisis y el diseño del software se escogió el lenguaje UML (*Unified Modeling Language*, Lenguaje Unificado de Modelación). Esta decisión se debe a que se ha convertido en un estándar que tiene las siguientes características:

- Permite modelar sistemas utilizando técnicas orientadas a objetos (OO).
- Permite especificar todas las decisiones de análisis y diseño, construyéndose así modelos precisos, no ambiguos y completos.
- Puede conectarse con lenguajes de programación (Ingeniería directa e inversa).
- Permite documentar todos los artefactos de un proceso de desarrollo (requisitos, arquitectura, pruebas, versiones, etc.).
- Es un lenguaje muy expresivo que cubre todas las vistas necesarias para desarrollar y luego desplegar los sistemas.
- Existe un equilibrio entre expresividad y simplicidad, pues no es difícil de aprender ni de utilizar.

UML es independiente del proceso, aunque para utilizarlo óptimamente se debería usar en un proceso que fuese dirigido por los casos de uso, centrado en la arquitectura, iterativo e incremental.

2.5 Metodologías y herramientas de desarrollo

Para la realización de la tesis se hizo un estudio de cada una de las posibles metodologías y herramientas a utilizar. A continuación se presenta la metodología que se utilizó y cada una de las herramientas con sus características distintivas que se tuvieron en cuenta para su selección.

2.5.1 Metodologías

La metodología de desarrollo utilizada en la realización de esta Tesis es: RUP (Proceso Unificado de Desarrollo). RUP sirve de guía para realizar el análisis y diseño de la aplicación, debido a que es una metodología que ha probado su efectividad durante muchos años, ya que numerosos proyectos la han utilizado para desarrollar su software. Además, tiene un gran número de documentos publicados que

se pueden consultar para esclarecer dudas. También porque siguiendo sus pasos propuestos se obtiene una buena documentación de la tesis.

A continuación se muestran las características que más influyeron en la selección de esta metodología:

- Guiado por casos de uso: Los casos de uso reflejan lo que los usuarios futuros necesitan y desean**,** constituyen la guía fundamental establecida para las actividades a realizar durante todo el proceso de desarrollo del sistema.
- Centrado en arquitectura: La arquitectura muestra la visión común del sistema completo. Permite además, implementar el Framework (plataforma sobre la que se implementa el soporte para todas las funcionalidades del sistema) y luego ir desarrollando cada uno de los módulos según se van necesitando.
- Iterativo e Incremental: RUP divide el proyecto en fases de desarrollo, propone además que cada una de ellas se desarrolle en iteraciones, las cuales aportan un incremento en el proceso de desarrollo y terminan con el cumplimiento del punto de control trazado en la fase.
- Utilización de un único lenguaje de modelado: UML.

2.5.2 Herramientas

Microsoft Visual Studio 2003

Posee numerosas herramientas asociadas que ayudan a escribir, analizar y distribuir el código, es un compilador rápido y con muy buena detección y corrección de errores. Posee facilidad de trabajo con los elementos visuales y buena integración de estos con el código. Contiene muchas librerías con códigos pre-escritos que ayudan en la escritura del código de la aplicación.

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*), diseño asistido por computador.

OpenGL ha revolucionado con el Hardware gráfico. La interfaz es extensible, lo que significa que pueden añadirse funcionalidades de forma incremental. Hoy en día, ofrece extensiones para trabajar con shaders para vértices y fragmentos. Con la versión 2.0 de la interfaz, se introdujo el lenguaje de sombreado de alto nivel GLSL (GL *Shading Language*).

Rational Rose

Rational Rose es una herramienta de desarrollo del software para el Modelado Visual. Rational tiene gran ventaja para el equipo de desarrollo ya que los unifica a través del modelamiento el cual está basado en el *Unified Modeling Language* (UML). El UML es la notación estándar para arquitectura de software. Esto significa que con Rational Rose, todo el equipo puede comunicarse con un lenguaje y una herramienta. Rational Rose domina el mercado de herramientas para el análisis, modelamiento, diseño y construcción orientado a objetos

Rational Rose permite visualizar, entender, y refinar los requerimientos y arquitectura antes de enfrentar el código. Esto permite, evitar esfuerzos desperdiciados en el ciclo de desarrollo. El modelo arquitectónico puede ser rastreado hacia el modelo de procesos de negocios y los requerimientos de sistema. Esta herramienta permite especificar, analizar, diseñar el sistema antes de codificarlo. Tiene varias características que lo distinguen como son el de chequear la sintaxis UML, mantiene la consistencia de los modelos del sistema del software, genera la documentación automáticamente, la generación de código a partir de los modelos, permite la ingeniería inversa (crear modelo a partir código) entre otras.

Conclusiones

Como se pudo apreciar, en este capítulo se presentaron los algoritmos con el cual se implementaran las luces y las sombras dinámicas para escenas virtuales. A partir del estudio realizado y tomando como guía los objetivos propuestos, se determino que estos serían las técnicas que se adecuan a la situación, y por tanto, partiendo de ellos, se diseñará una estructura de clases en correspondencia con las técnicas citadas anteriormente.

3

Capítulo 3 Construcción de la Solución Propuesta

Introducción

Luego del análisis realizado en los capítulos anteriores se comenzara a sintetizar los conceptos y objetos en la visión del sistema a desarrollar. A partir de las dificultades, características y necesidades del cliente, y utilizando las técnicas analizadas, se inicia la concepción práctica del producto a elaborar.

Además se mostrará un modelo de dominio que abarcará los objetos e entidades que se relacionan entre si para dar solución al problema planteado, así como también veremos algunas reglas del negocio y los principales requerimientos con los que el sistema debe cumplir.

3.1 Objeto de estudio

Para el tratamiento de Luces y Sombras Dinámicas que son producidas por y sobre objetos que están en movimiento, ya hay implementado un módulo, y a partir de la necesidad de crear entornos virtuales cada vez más reales por parte de los programadores, se necesita darle una mejora tomando como concepción que el realismo del entorno virtual 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.

A partir de, cómo mejorar el módulo antes creado y que está siendo utilizado por el proyecto HDSRV, se propone la creación de uno nuevo que reúna los algoritmos que den solución a este problema, tomando como objeto de estudio el proceso de visualización de luces y sombras en entornos de realidad virtual y como campo de acción el trabajo con los *shaders* para el desarrollo de dicho proceso.

El objetivo principal de este trabajo es implementar un módulo de clases que permita un tratamiento más amplio de luces y sombras dinámicas en entornos virtuales, traduciéndose esto, en que el módulo debe permitir representar más de una fuente de luz en la escena y que permita lograr un sombrado dinámico, para ello se plantean un grupo de tareas que se resume a continuación:

- Estudiar las características básicas de los módulos de efectos visuales, particularizando en las luces y sombras.
- Analizar algoritmos utilizados en la creación y visualización de luces y sombras dinámicas.
- Estudiar las principales herramientas existentes para realizar la simulación de entornos virtuales en 3D en el mundo.
- Observar las potencialidades de los lenguajes de alto nivel utilizados en el mundo para desarrollo de efectos visuales.
- Explorar las potencialidades de las tarjetas gráficas a utilizar.
- Rediseñar la biblioteca de clases que de solución a la visualización de las luces y sombras dinámicas.

3.2 Modelo del Dominio

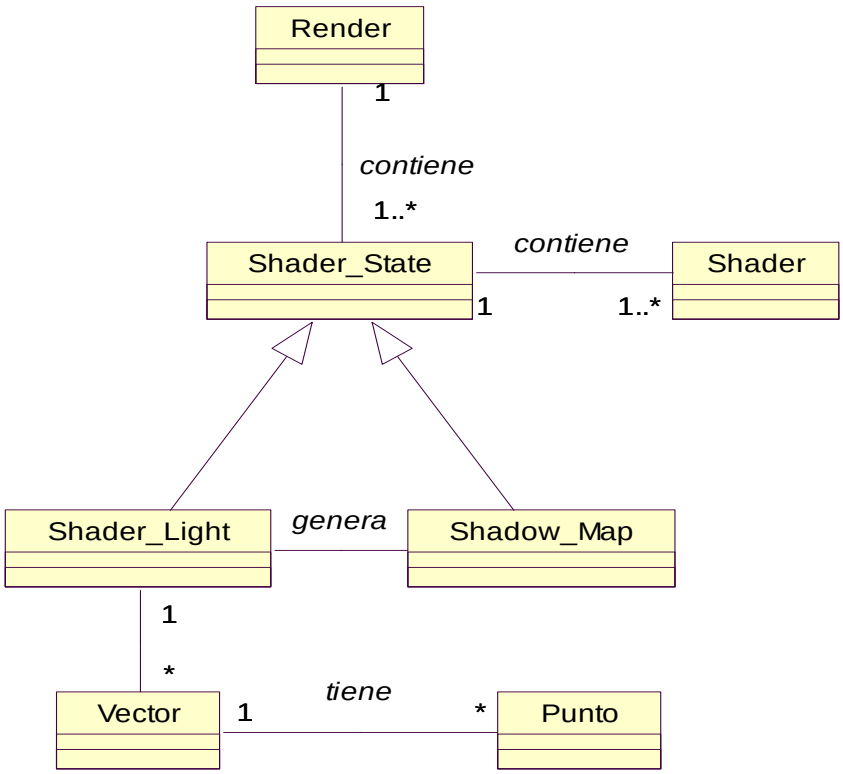

Figura 29: Modelo Del Dominio.

3.2.1 Glosario de términos del modelo del dominio

Render: es el encargado de reenderear toda la escena incluyendo las fuentes de luces que se quieran incluir, así como también las sombras que puedan arrojar los objetos que en ella se encuentren.

Shader_State: es la estructura encargada de almacenar los diferentes tipos de shaders, ya sean tanto de iluminación como de sombreado y para la manipulación de estos contiene a la estructura manejadora de los shader.

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

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

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

3.3 Captura de Requisitos

A partir de las necesidades del cliente y considerando el funcionamiento como tal del sistema se obtuvo una serie de requisitos funcionales que serian las capacidades o condiciones que el sistema debe cumplir los que se muestran más adelante, así mismo pero teniendo en cuenta las necesidades de hardware, de soporte, usabilidad, etc. se extrajeron los no funcionales siendo estos las propiedades o cualidades que el producto.

3.3.1 Requisitos Funcionales

1. Crear luces
2. Modificar luces
3. Eliminar luces
4. Habilitar luces
5. Deshabilitar luces
6. Crear las sombras
7. Habilitar las sombras
8. Deshabilitar sombras.
9. Cargar *shader*
10. Asociar los *shader* al proyecto
11. Cargar fichero
12. Atachar datos del fichero al *shader*

3.3.2 Requisitos no Funcionales

1. **Hardware:** Compatibilidad con tarjetas gráficas de la familia nVIDIA (*Quadro FX 500/FX 600*)
2. **Usabilidad:** Los futuros usuarios del sistema serán todas las aplicaciones finales que introduzcan en sus implementaciones el trabajo con luces y sombras dinámicas y deseen utilizar este pequeño módulo de clases.
3. **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.
4. **Rendimiento:** Como aplicación de tiempo real, debe tener alto grado de velocidad de procesamiento o cálculo, tiempo de respuesta y de recuperación, y disponibilidad.
5. **Diseño e implementación:** Debe utilizar transparentemente la biblioteca gráfica OpenGL, en su primera versión, y ser adaptable a trabajar con otras bibliotecas. Se harán llamadas a dichas bibliotecas desde C++ y se usará GLSL para el manejo de los *shader*. Se regirá por la filosofía de Programación Orientada a Objetos.

3.4 Modelo de Casos de Usos del Sistema

En este epígrafe se muestran los actores del sistema y se conciben, a partir de los requisitos funcionales planteados anteriormente, así como también 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.

3.4.1 Actor del sistema

Tabla 1: Actor del Sistema.

Actores	Justificación
Aplicación Final	Es el que se beneficiará con las funcionalidades que brinda el módulo de clases, a grandes rasgos: cargar desde ficheros, adicionar más de una fuente de luz a la escena así como también las sombras correspondientes.

3.4.2 Casos de uso del sistema

1. Gestionar luces
2. Gestionar sombra
3. Cargar *vertex-fragmet shader*
4. Cargar fichero

3.4.3 Diagrama de Casos de Uso del Sistema

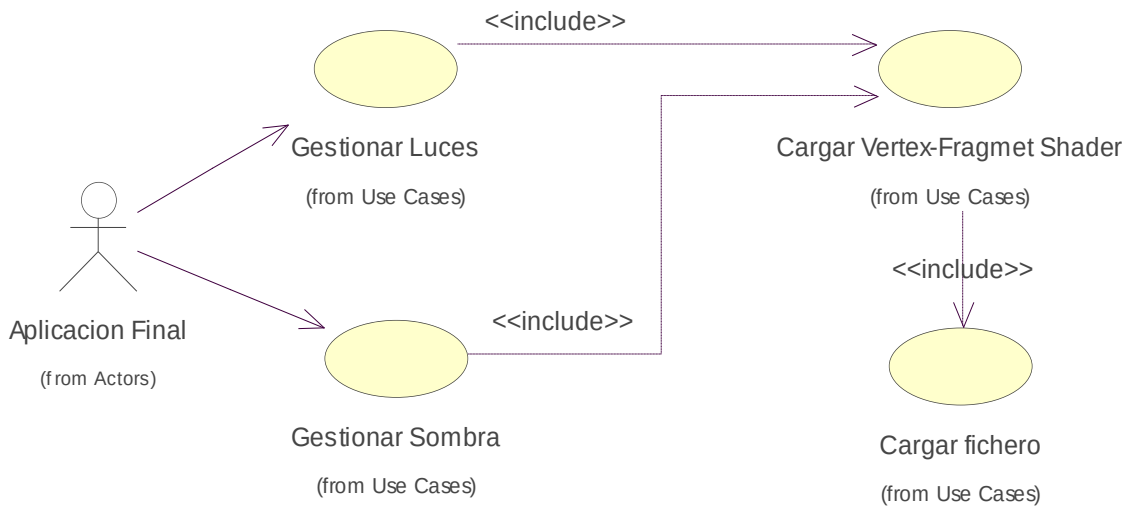

Figura 30: Diagrama de Casos de Uso.

3.4.4 Especificación de los casos de uso en formato expandido

Tabla 2: CU1 Gestionar Luces

Caso de Uso Gestionar Luces	
Actores	Aplicación Final
Propósito	Adicionar luces a la escena, poder actualizarlas, habilitarlas o deshabilitarlas.
Resumen: El caso de uso se inicia cuando la Aplicación Final solicita la creación de una nueva fuente de luz o requiere que esta sea actualizada, en otro caso habilitarla o deshabilitada.	
Referencias	R1, R2, R3, R4, R5,R10,12
Precondiciones	Shader de luz previamente creado.
Poscondiciones	Shader de luz con las actualizaciones requeridas.
Curso normal de los eventos:	
Acción del actor	**Respuesta del sistema**
1. La Aplicación Final solicita adicionar una fuente de luz, actualizarla o deshabilitarla luego de creada.	1.1 El sistema ejecuta las siguientes acciones: a) Para adicionar una 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 *vertex-fragmet shader*" y se le asigna el estado de activado. b) Para actualizar luz se asigna valores a las variables controladoras según la función que estas cumplan. El CU concluye cuando todas las variables son actualizadas. c) Para habilitar las luces se busca en la lista que almacenan los *shader* de iluminación por el identificador, se habilita y luego se usa. d) Para deshabilitar las luces se busca en la lista que

	almacenan los *shader* de iluminación por el identificador, se deshabilita y luego se deja de usar.
Sección "Adicionar Luz "	
Acción del actor	**Respuesta del sistema**
2. Solicita la adición de una nueva fuente de luz.	2.1 Se crean las estructuras necesarias para almacenar el shader(*vertex* y *fragment*)
	2.2 Se hace referencia al "CU Cargar *vertex-fragmet shader*"
	2.3 Se le asigna un manejador (*handel*) al código *shader* cargado.
	2.4 Se compila el código del *shader* y verifica que el estado de la compilación fue correcto.
	2.5 Se crea el objeto luz, se adicionan los objetos (*vertex* y *fragment*) anteriormente creados y se crea un programa *shader*.
	2.6 Se asocian los objetos (*vertex* y *fragment*) al programa creado.
	2.7 Se enlaza el programa y se verifica que el enlazado fue correcto.
	2.8 Se comprueba la localización de los parámetros para transformar en el *shader*.
	2.9 Se le asigna al objeto un tipo (luz), un identificador, se habilita y se guarda en la lista de objetos.

Curso alternativo de los eventos:	
	2.2 En caso de no haber ningún *shader* cargado en la lista de objetos de este tipo, no se ilumina la escena, la aplicación se ejecuta paro el resultado no es el esperado.
	2.4 Si el estado de la compilación no fue correcto se devuelve un mensaje de error y finaliza el programa.
	2.7 Si el estado del enlazado no fue correcto se devuelve el mensaje de error y finaliza el programa.
Sección "Actualizar Luz "	
Acción del actor	**Respuesta del sistema**
3. Solicita la actualización de las luces de la escena.	3.1 Se guardan los valores de la posición de luz y de la vista.
	3.2. Se vincula el *Sampler2D* del *shader* con la textura activa, se envían las variables controladoras de la posición de la luz y de la vista.
	3.3 Se cargan las variables a utilizar en el *vertex program*.
	3.4 Se obtienen las coordenadas de la textura activa.
	3.5 Se transforma la posición de los vértices al espacio pantalla.
	3.6 Se obtiene la normal, la dirección de la luz y de la vista para ser enviados al *fragment program*.
	3.7 Se cargan las variables a utilizar en el *fragment program*.

	3.8 Se asocia el *Sampler2D* del *shader* con las coordenadas de la textura activa capturada anteriormente.
	3.9 Se normaliza la normal, la dirección de la luz y la dirección de la vista.
	3.10 Se determinan los componentes ambiental, especular y difuso.
	3.11 Se interpola el color resultante de la suma de los componentes para cada *píxel* de la escena.
Curso alternativo de los eventos:	
Sección "Habilitar Luz "	
Acción del actor	**Respuesta del sistema**
4. Solicita habilitar la luz en la escena.	4.1 Se busca el objeto luz en la lista de objetos según el identificador dado.
	4.2 Se habilitado y se activa el programa shader.
Curso alternativo de los eventos:	
Sección "Deshabilitar Luz "	
Acción del actor	**Respuesta del sistema**
5. Solicita deshabilitar la luz de la escena.	5.1 Se busca el objeto luz en la lista de objetos según el identificador dado.

	5.2 Se le cambia el estado a deshabilitado y se desactiva el programa shader.
Curso alternativo de los eventos:	
Prioridad:	Crítica

Tabla 3: CU2 Gestionar Sombra

Caso de Uso Gestionar Sombra	
Actores	Aplicación Final
Propósito	Crear, actualizar, habilitar y deshabilitar las sombras en una escena.
Resumen: El caso de uso se inicia cuando la Aplicación Final quiere crear las sombras arrojadas por los objetos existentes en la escena o quiere actualizarlas, habilitarlas o deshabilitarlas.	
Referencias	R6, R7, R8, R10,12
Precondiciones	Estado de luz creado.
Poscondiciones	Shader de sombra cargado y aplicado.
Curso normal de los eventos:	
Acción del actor	**Respuesta del sistema**
1. Solicita crear, actualizar, habilitar y deshabilitar las sombras de la escena.	1.1 El sistema ejecuta las siguientes acciones: a) Para crea las sombras se carga el shader de sombra, se verifica la localización de los parámetros del shader y activa la textura para copiar el buffer de profundidad. b) Para actualizar las sombras se captura la posición de la luz y se envían al shader para proyectar la luz.

	c) Habilita las sombras para ello busca en la lista de *shader* e identifica los de sombra y los habilita. d) Deshabilita las sombras para ello busca en la lista de *shader* e identifica los de sombra y los deshabilita.
Sección "Crear Sombra"	
Acción del actor	**Respuesta del sistema**
2. Solicita crear las sombras en la escena.	2.1 Se crean las estructuras necesarias para almacenar el *shader* (*vertex* y *fragment*).
	2.2 Se hace referencia al "CU Cargar *vertex-fragmet shader*"
	2.3 Se le asigna un manejador (*handle*) al código *shader* cargado.
	2.4 Se compila el código del *shader* y verifica que el estado de la compilación fue correcto.
	2.5 Se crea el objeto sombra, se adicionan los objetos (*vertex* y *fragment*) anteriormente creados y se crea un programa *shader*.
	2.6 Se asocian los objetos (*vertex* y *fragment*) al programa creado.
	2.7 Se enlaza el programa y se verifica que el enlazado fue correcto.
	2.8 Se verifica que exista al menos una fuente de luz y se le asigna dicho objeto al objeto sombra.

	2.9 Se activa el mapa de sombra para copiar el buffer de profundidad en la textura.
	2.10 Se comprueba la localización de los parámetros para transformar en el *shader*.
	2.11 Se le asigna al objeto un tipo (sombra), se habilita y se guarda en la lista de objetos.
Curso alternativo de los eventos:	
	2.2 En caso de no haber ningún *shader* cargado en la lista de objetos de este tipo (sombra) no se realiza el sombreado, la aplicación se ejecuta paro el resultado no es el esperado.
	2.4 Si el estado de la compilación no fue correcto se devuelve el mensaje de error y finaliza el programa.
	2.7 Si el estado del enlazado no fue correcto se devuelve el mensaje de error y finaliza el programa.
Sección "Actualizar Sombra"	
Precondiciones	Matrices de modelado y de proyección guardadas según la posición de la luz.
Acción del actor	**Respuesta del sistema**
3. Solicita actualizar las sombras en la escena.	3.1 Se carga el valor de la matriz de proyección de la escena.
	3.2 Se carga el valor de la matriz de modelado de la escena.

	3.3 Se desactiva la escritura de todos los buffers excepto al *Depth buffer.*
	3.4 Se aplica un *depth offset* para evitar el Z-*figthing* entre el mapa de sombra y la superficie que recibe la sombra.
	3.5 Se captura el buffer de profundidad y se construye la matriz de transformación según la luz.
	3.6 Se fija la matriz de textura con la matriz de transformación de la luz para transformar en el *shader*.
	3.7 Se vincula el S*ampler2DShadow* del *shader* con la textura activa y se envía la variable controladora de la posición de la luz.
	3.8 Se cargan las variables a utilizar en el *shader*.
	3.9 Se transforma la posición del vértice al espacio vista y se calcula la sombra.
Curso alternativo de los eventos:	
Sección "Habilitar Sombra"	
Acción del actor	**Respuesta del sistema**
4 Solicita habilitar las sombras en la escena.	4.1 Se busca el objeto de tipo sombra en la lista de objetos.
	4.2 Se habilitado y se usa programa a partir del manejador del programa.
Curso alternativo de los eventos:	

Sección "Deshabilitar Sombra"	
Acción del actor	**Respuesta del sistema**
5 Solicita deshabilitar las sombras en la escena.	5.1 Se busca el objeto de tipo sombra en la lista de objetos.
	5.2 Se deshabilita y se deja de usar el programa.
Curso alternativo de los eventos:	
Prioridad:	Crítica

Tabla 4: CU3 Cargar light- shadow shader.

Caso de Uso Cargar vertex-fragmet shader	
Actores	
Propósito	Interpretar el *buffer* de datos del fichero y almacenarlos en las estructuras de datos de los *shader*.
Resumen: El caso de uso se inicia cuando algún CU requiere de la creación de un shader de iluminación o de sombra. Este permite cargar un fichero con extensión ".vert" y ".frag" escrito todo en minúscula o mayúscula, es decir que puede ser ("vert" "FRAG", "vert" "frag" o cualquier combinación de este).	
Referencias	R9,CU 1,CU2
Precondiciones	Dirección correcta de los ficheros que contiene los *shaders.*
Poscondiciones	Nuevo *shader* cargado.
Curso normal de los eventos:	

Acción del actor	**Respuesta del sistema**
	1.1 Se chequea que el fichero no esté vacío.
	1.2 Almacena en una variable la extensión, es decir después del punto que la delimita del nombre.
	1.3 Se compara que la extensión tenga el formato ".vert" y ".frag" escritos todo en mayúscula o minúscula o en cualquier otra combinación (".vert" y ".FRAG" o ".VERT" y ".frag").
	1.4 Se hace referencia al "CU Cargar fichero" para leer el fichero y almacenar el código del *shader* cargado.
Curso Alternativo:	
	1.2 Si el fichero está vacío no se carga el *shader* de sombra o de luz y por tanto no se obtiene los resultados esperados
	1.3 Si el fichero no presenta la extensión indicada no se carga el *shader* de sombra o de luz y por tanto no se obtiene los resultados esperados
Prioridad:	Crítica

Tabla 5: CU4 Cargar fichero.

Caso de Uso Cargar fichero	
Actores	Cargar vertex-fragmet shader
Propósito	Cargar un fichero.

<table>
<tr><td colspan="3">Resumen: Este CU se inicia cuando “Cargar vertex-fragmet shader” necesita cargar los shader, ya sea de iluminación o de sombreado, este permite leer de un fichero.</td></tr>
<tr><td>Referencias</td><td colspan="2">R11, CU3(include)</td></tr>
<tr><td>Precondiciones</td><td colspan="2">Que el fichero no este vacío y con la extensión indicada para que el CU se ejecute de forma correcta.</td></tr>
<tr><td>Poscondiciones</td><td colspan="2">Que el fichero contengas los datos necesarios para asociar las variables al shader que venía en el fichero cargado.</td></tr>
<tr><td colspan="3">Curso normal de los eventos:</td></tr>
<tr><td colspan="2">Acción del actor</td><td>Respuesta del sistema</td></tr>
<tr><td colspan="2" rowspan="4">1. Solicita la cargar del fichero pasado por parámetro.</td><td>2.1 Si el fichero no esta vacío</td></tr>
<tr><td>2.2 Se abre en forma de lectura.</td></tr>
<tr><td>2.3 Se lee el fichero y se guarda el contenido.</td></tr>
<tr><td>2.4 Se cierra el fichero</td></tr>
<tr><td colspan="3">Curso Alternativo:</td></tr>
<tr><td colspan="2"></td><td>2.1 Si el fichero está vacío no se ejecuta la lectura del fichero indicado</td></tr>
<tr><td>Prioridad:</td><td colspan="2">Crítica</td></tr>
</table>

Conclusiones

En este capítulo se definieron las funcionalidades del sistema a partir de la recogida de los requisitos más importantes, a los que la aplicación debe responder para dar un total cumplimiento a los objetivos propuestos. Es importante resaltar que la funcionalidad de adicionar más de una fuente de luz a la escena es una de las que más peso tiene en la investigación, así como también lo es representar las sombras a partir del algoritmo *shadow mapping*.

4

Capítulo 4 Diseño e Implementación del Sistema

Introducción

La primera parte del capítulo encierra los diagramas de clases de análisis del sistema propuesto, donde se identifican las clases que describen la realización de los casos de uso, es decir se traducen los requisitos a una especificación que describe cómo implementar el sistema. Partiendo de que el análisis consiste en obtener una visión del sistema que se preocupa de ver QUÉ hace, de modo que sólo se interesa por los requisitos funcionales. Por otra parte el diseño es un refinamiento del análisis que tiene en cuenta los requisitos no funcionales, en definitiva CÓMO cumple el sistema sus objetivos. Con esta información se construye el diagrama de clases del análisis el cual tiene un impacto por lo general en el diseño e implementación de la solución.

Ahora bien, en la etapa de implementación que es la etapa del proyecto que 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++, se relaciona el modelo de implementación y el modelo de diseño y se determina cuales elementos de implementación son suficientemente importantes para modelarlos. Para esta fase se tiene en cuenta un estándar de codificación al se hacer referencia al final del epígrafe.

4.1 Diagrama de clases del Análisis

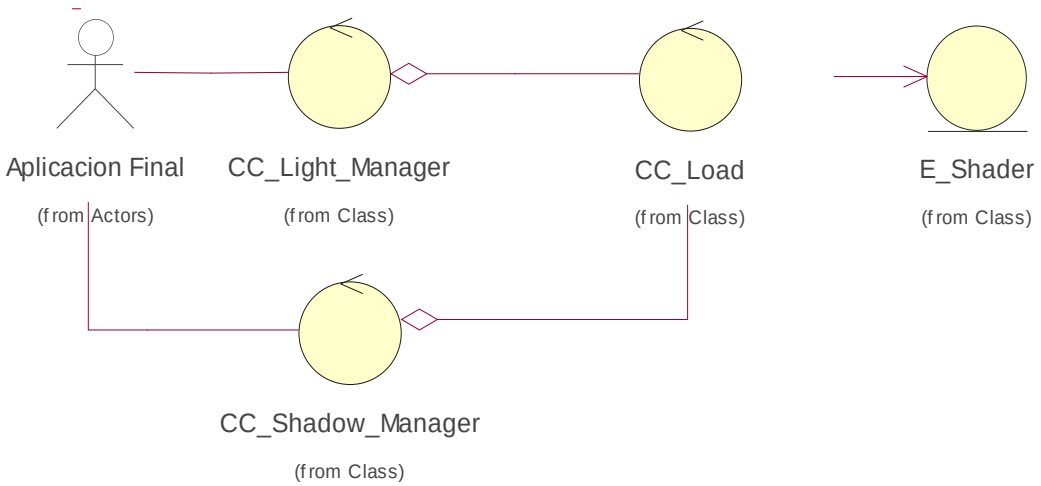

Figura 31: Diagrama de clases del Análisis

4.2 Diagrama de clases de Diseño

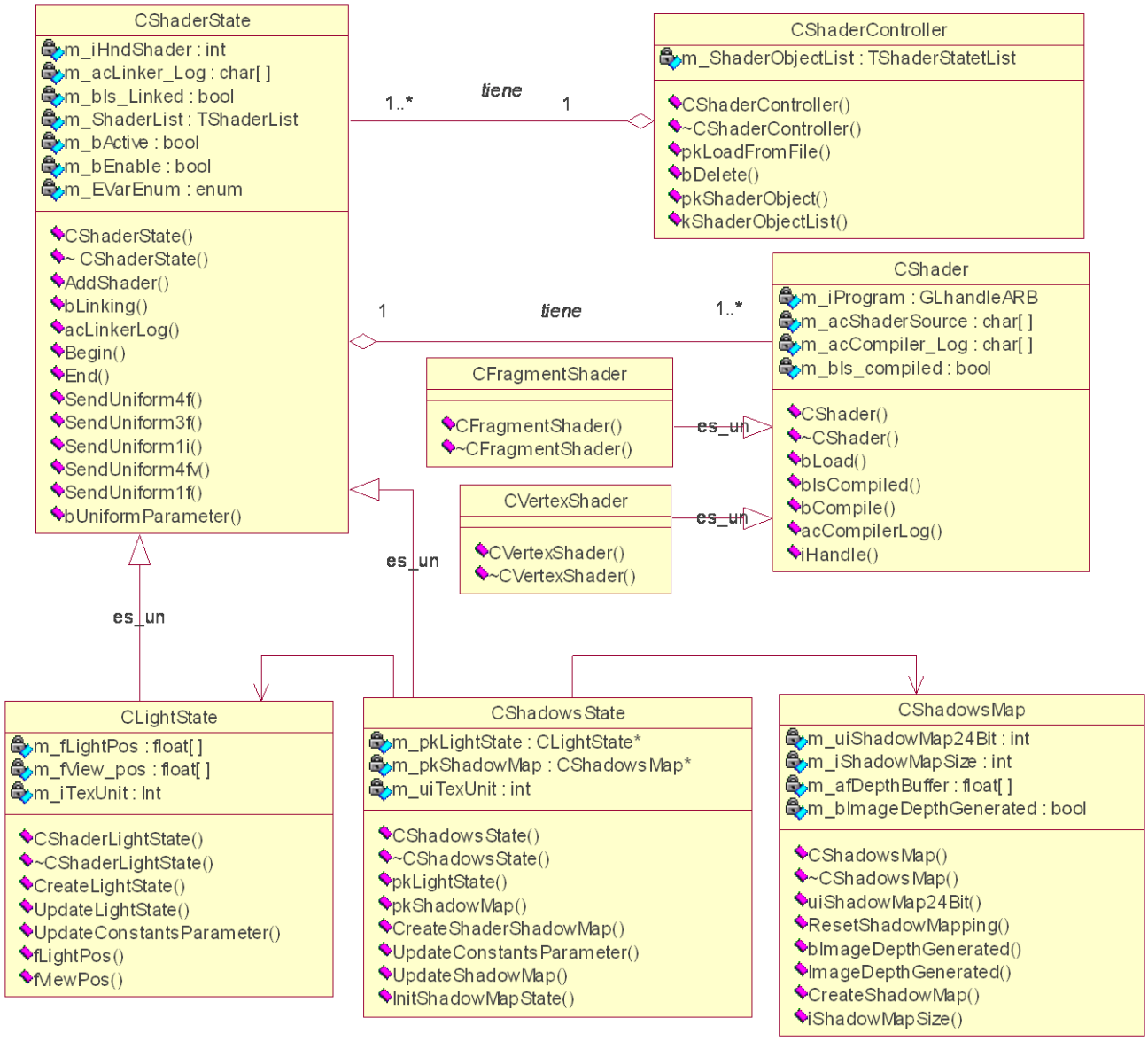

Figura 32: Diagrama de clases de diseño

4.2.1 Descripción de las clases del Diseño

Tabla 6: Descripción de la clase CShaderController.

CShaderController	
Tipo de clase: Controladora	
Atributo	**Tipo**
m_ShaderObjectList	TShaderStatetList
Para cada responsabilidad:	
Nombre:	CShaderController ()
Descripción:	Constructor de la clase.
Nombre:	~CShaderController ()
Descripción:	Destructor de la clase.
Nombre:	pkLoadFromFile(string, string, enum)
Descripción:	Método que pasándole como parámetro un *vertex* , un *fragment shader* y un variable de tipo enum que indica el tipo de shader (luz o sombra) lo carga, lo compila, lo adiciona a la lista de shader, lo linkea y luego le asigna el tipo(luz o sombra).
Nombre:	bDelete (CShaderState*)
Descripción:	Método que devuelve verdadero si elimina el shader pasado como parámetro de la lista de objetos y falso en caso contrario.
Nombre:	pkShaderObject(int)
Descripción:	Retorna el shader que se encuentra en la posición pasada como parámetro.
Nombre:	kShaderObjectList()
Descripción:	Método de acceso a miembro m_ShaderObjectList, variable donde se guardan los diferentes tipos de *shaders*.

Tabla 7: Descripción de la clase CShaderState

CShaderState	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_iHndShader	int
m_acLinker_Log	char[]
m_bIs_Linked	bool
m_ShaderList	TShaderList
m_bActive	bool
m_bEnable	bool
m_EVarEnum	enum
Para cada responsabilidad:	
Nombre:	CShaderState ()
Descripción:	Constructor de la clase donde se crea un programa ARB y se ponen las variables m_bIs_Linked y m_bActive en falso y m_EVarEnum toma el valor TS_NONE del enum.
Nombre:	~CShaderState()
Descripción:	Destructor de la clase donde se recorre la lista de *shader* y se desatacha y luego se elimina.
Nombre:	AddShader(CShader*)
Descripción:	Devuelve un mensaje de error si el *shader* no fue compilado en caso contrario lo adiciona a la lista de *shaders.*

Nombre:	bLinking()
Descripción:	Método que atacha el programa, lo vincula y retorna verdadero si fue vinculado correctamente, de lo contrario retorna falso.
Nombre:	acLinkerLog()
Descripción:	Método que devuelve un mensaje de error de no ser un objeto válido, de estar fuera de memoria o tener algún error de compilación.
Nombre:	Begin()
Descripción:	Método que verifica que el shader esté vinculado para luego usarlo y le asigna verdadero a la variable m_bActive.
Nombre:	Begin()
Descripción:	Método que verifica que el shader esté vinculado para luego usarlo y le asigna verdadero a la variable m_bActive.
Nombre:	End()
Descripción:	Método que deshabilita el shader, le asigna a la variable m_bActive falso.
Nombre:	SendUniform4f(char*,float, float, float, float)
Descripción:	Método que localiza la variable por el nombre especificado y luego envía 4 parámetros de tipo *float* al shader además del valor de localización.
Nombre:	SendUniform3f(char*, float, float, float)
Descripción:	Método que localiza la variable por el nombre especificado y luego envía 3 parámetros de tipo *float* al shader además del valor de localización.
Nombre:	SendUniform1i(char*, int)

Descripción:	Método que localiza la variable por el nombre especificado y luego envía un parámetro de tipo entero al shader además del valor de localización.
Nombre:	SendUniform4fv(char*, int, float*)
Descripción:	Método que localiza la variable por el nombre especificado y luego envía dos parámetros al shader (uno entero y un arreglo de *float*) además del valor de localización.
Nombre:	SendUniform1f(char*, float)
Descripción:	Método que localiza la variable por el nombre especificado y luego envía un parámetros al shader (uno entero).
Nombre:	bUniformParameter(char*,int)
Descripción:	Método que retorna verdadero si se localiza la variable pasada por parámetro a partir de su nombre y su manejador.
Nombre:	bIsActive()
Descripción:	Método de acceso a miembro m_bActive, que puede tomar valor falso o verdadero; cuando toma valor verdadero indica que está activado el shader y si toma valor falso indica lo contrario.
Nombre:	TShaderList ()
Descripción:	Método de acceso a miembro m_ShaderList, que es la estructura para almacena los shader.
Nombre:	bEnable()
Descripción:	Método de acceso a miembro m_bEnable, que toma valor verdadero o falso en dependencia de que si lo que se desea es habilitar o deshabilitar el shader.
Nombre:	ETypeVarEnum()
Descripción:	Método de acceso a miembro m_EVarEnum, que indica el tipo de shader (Luz o Sombra)

Nombre:	GLhandleARB iProgram()
Descripción:	Método de acceso a miembro m_iHndShader, que es una variable entera que constituye el manejador del programa.

Tabla 8: Descripción de la clase CLightState

CLightState	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_fLightPos	float[ ]
m_fView_pos	float[ ]
m_iTexUnit	Int
Para cada responsabilidad:	
Nombre:	CShaderLightState()
Descripción:	Constructor del objeto donde se crea espacio para guardar la posición de la luz y de la vista.
Nombre:	~CShaderLightState()
Descripción:	Destructor del objeto
Nombre:	CreateLightState()
Descripción:	Lanza una excepción si los parámetros de: textura, posición de la luz y la de la vista no están en la localización indicada.
Nombre:	UpdateLightState()
Descripción:	Envía los parámetros: m_iTexUnit como “tex”, m_fLightPos como “light_pos" y m_fView_pos como "view_pos".

Nombre:	UpdateConstantsParameter (CGeometryLight*, cCamera)
Descripción:	Método que controla la posición de la cámara y de la geometría.
Nombre:	fLightPos()
Descripción:	Método de acceso miembro m_fLightPos donde se guarda la posición de la luz.
Nombre:	fViewPos()
Descripción:	Método de acceso miembro m_fView_pos donde se guarda la posición de vista.

Tabla 9: Descripción de la clase CShadowsState

CShadowsState	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_pkLightState	CLightState*
m_pkShadowMap	CShadowsMap*
m_uiTexUnit	int
Para cada responsabilidad:	
Nombre:	CShadowsState ()
Descripción:	Constructor del objeto, donde se crean las variables necesarias para almacenar los objetos de tipo luz y de tipo sombra donde también se inicializa la variable m_uiTexUnit en 0.
Nombre:	~CShadowsState ()
Descripción:	Destructor de la clase.
Nombre:	pkLightState()

Descripción:	Método de acceso a miembro m_pkLightState, que es un puntero que almacena un estado de luz.
Nombre:	pkShadowMap()
Descripción:	Método de acceso a miembro m_pkShadowMap, que es un puntero que almacena un objeto de tipo sombra.
Nombre:	CreateShaderShadowMap()
Descripción:	Lanza una acepción si los parámetros de lightPos y ShadowMap no están en la localización indicada.
Nombre:	UpdateConstantsParameter (CGeometryLight*)
Descripción:	Método que controla la posición del objeto que emite luz en la escena.
Nombre:	UpdateShadowMap()
Descripción:	Envía la variable donde se va a guardar la textura como shadowMap.
Nombre:	InitShadowMapState()
Descripción:	Método que comprueba que el estado de sombra este habilitado, y si no esta habilitado crea una y luego cambia el valor de la variable m_bImageDepthGenerated a verdadero.

Tabla 10: Descripción de la clase CShadowMap

CShadowMap	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_uiShadowMap24Bit	int

m_iShadowMapSize	int
m_afDepthBuffer	float[ ]
m_bImageDepthGenerated	bool
Para cada responsabilidad:	
Nombre:	CShadowMap()
Descripción:	Constructor de la clase que inicializa las variables m_uiShadowMap24Bit, m_iShadowMapSize y m_bImageDepthGenerated.
Nombre:	~CShadowMap()
Descripción:	Destructor de la clase.
Nombre:	uiShadowMap24Bit()
Descripción:	Método de acceso a miembro m_uiShadowMap24Bit, que es la variable que se utilizara para guardar la textura generada.
Nombre:	ResetShadowMapping()
Descripción:	Método que activa la textura generada por el shadow map.
Nombre:	bImageDepthGenerated()
Descripción:	Método de acceso a miembro m_bImageDepthGenerated, variable booliana que toma valor verdadero cunado se habilita el estado de sombra y falso en caso contrario.
Nombre:	ImageDepthGenerated(bool)
Descripción:	Método que cambia el valor de la variable m_bImageDepthGenerated, cuando se le asigna valor verdadero habilitar el estado de sombra y falso para deshabilitarlo.
Nombre	CreateShadowMap()
Descripción:	Método donde se activa la textura para crear el mapa de sombra.

Nombre:	iShadowMapSize()
Descripción:	Método de acceso a miembro m_iShadowMapSize, que es la variable donde se almacena el tamaño del mapa de sombra.

Tabla 11: Descripción de la clase CShader

CShader	
Tipo de clase: Entidad	
Atributo	**Tipo**
m_iProgram	GLhandleARB
m_acShaderSource	char[ ]
m_acCompiler_Log	char[ ]
m_bIs_compiled	bool
Para cada responsabilidad:	
Nombre:	CShader()
Descripción:	Constructor de la clase
Nombre:	~CShader()
Descripción:	Destructor de la clase que libera la memoria, elimina el código shader y el manejador del programa.
Nombre:	bLoad(string)

Descripción:	Método que retorna verdadero si lee el fichero pasado por parámetro para ello comprueba que no esta vació, y que la extensión es correcta, es decir, que sea ".vert" o ".frag" escrito todo en minúscula o mayúscula; de no cumplir con estas especificaciones retorna falso.
Nombre:	bIsCompiled()
Descripción:	Método de acceso a miembro m_bIs_compiled, variable booliana que es verdadera si el código shader es compilado de lo contrario es falsa.
Nombre:	bCompile()
Descripción:	Método que retorna verdadero si compila el código shader y falso en cualquier otro caso.
Nombre:	acCompilerLog()
Descripción:	Lanza un mensaje de error en caso de que el programa no sea válido, o si esta fuera de memoria o el error ocurrido es desconocido.
Nombre:	iHandle()
Descripción:	Método de acceso a miembro m_iProgram, manejador del *vertex* y *fragment shader*.

Tabla 12: Descripción de la clase CVertexShader

CVertexShader	
Tipo de clase: Entidad	
Atributo	**Tipo**
Para cada responsabilidad:	
Nombre:	CVertexShader ()
Descripción:	Constructor de la clase que un objeto de tipo los *vertex shader*.

Nombre:	~CVertexShader ()
Descripción:	Destructor de la clase

Tabla 13: Descripción de la clase CFragmentShader.

CFragmentShader	
Tipo de clase: Entidad	
Atributo	**Tipo**
Para cada responsabilidad:	
Nombre:	CFragmentShader ()
Descripción:	Constructor de la clase que crea un objeto de tipo *fragment shader.*
Nombre:	~CFragmentShader()
Descripción:	Destructor de la clase.

4.3 Diagramas de Secuencia

Con motivo de aclarar la interacción entre objetos de las clases del análisis se realizan los diagramas de secuencia de cada uno de los CU, capturados en el capítulo anterior.

4.3.1 Diagrama de Secuencia CU Gestionar Luces

Sección "Adicionar Luz"

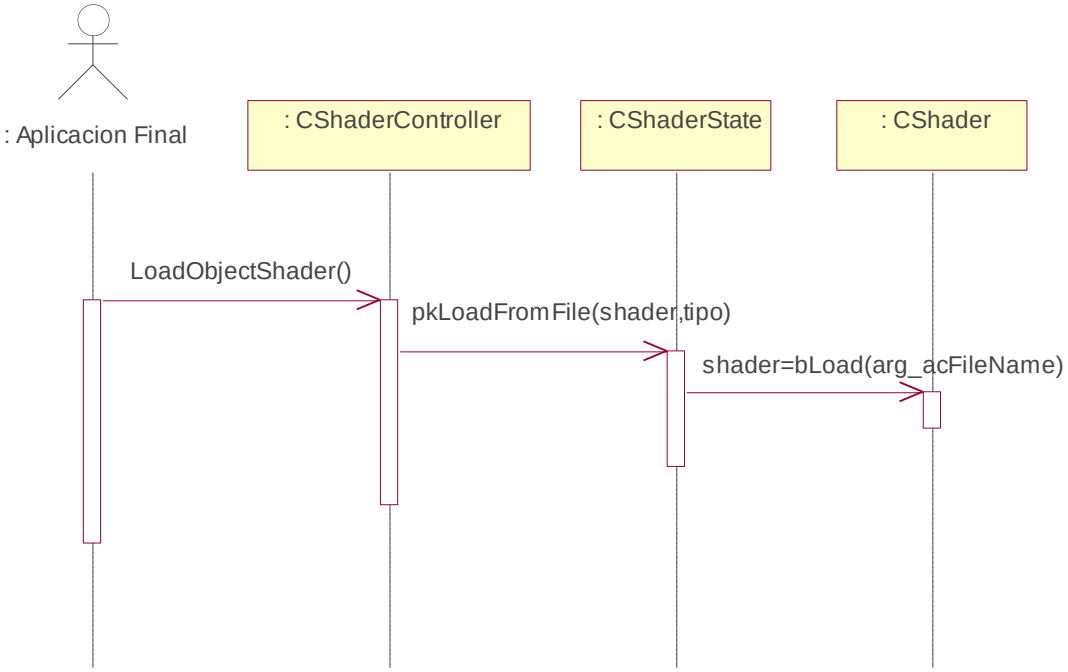

Figura 33: Sección Adicionar Luz

Sección “Actualizar Luz”

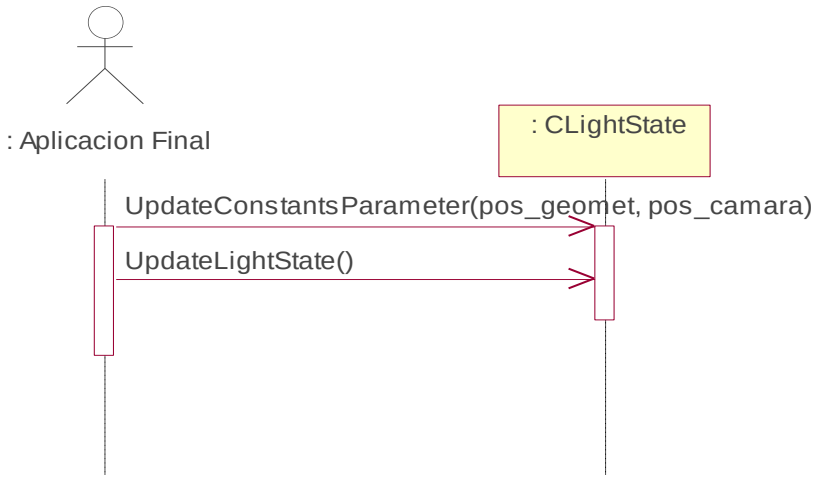

Figura 34: Sección Actualizar Luz

Sección “Habilitar Luz”

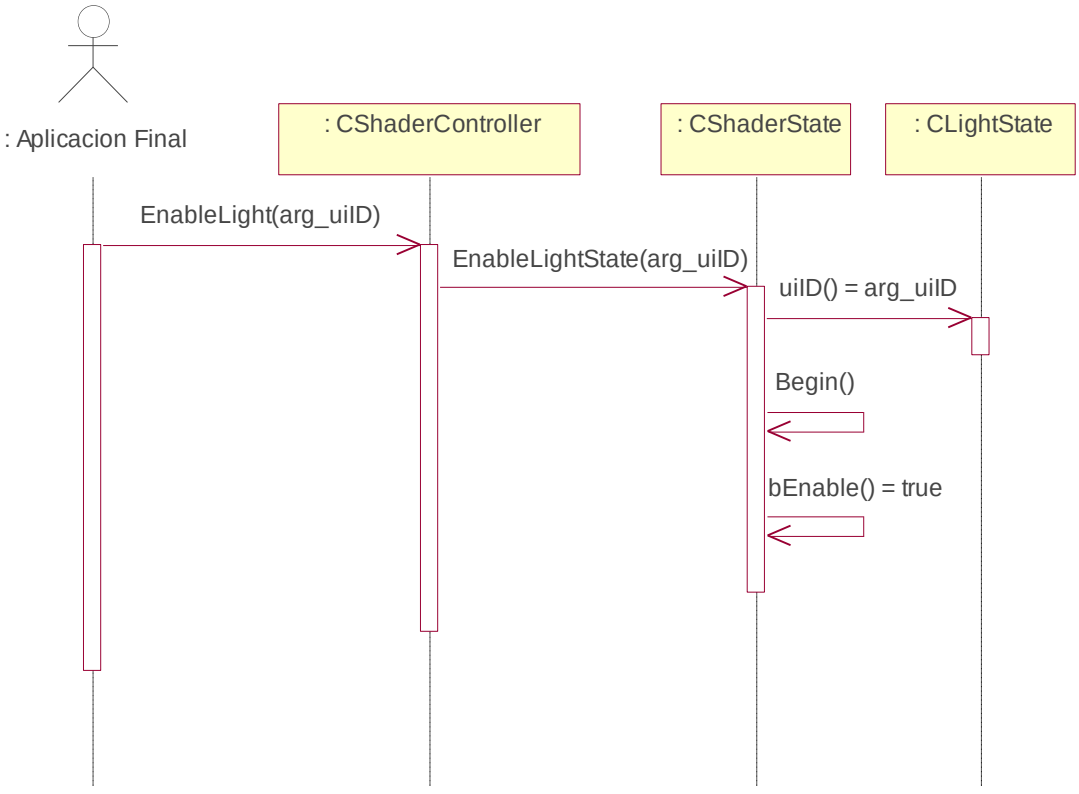

Figura 35: Sección Habilitar Luz

Sección “Deshabilitar Luz”

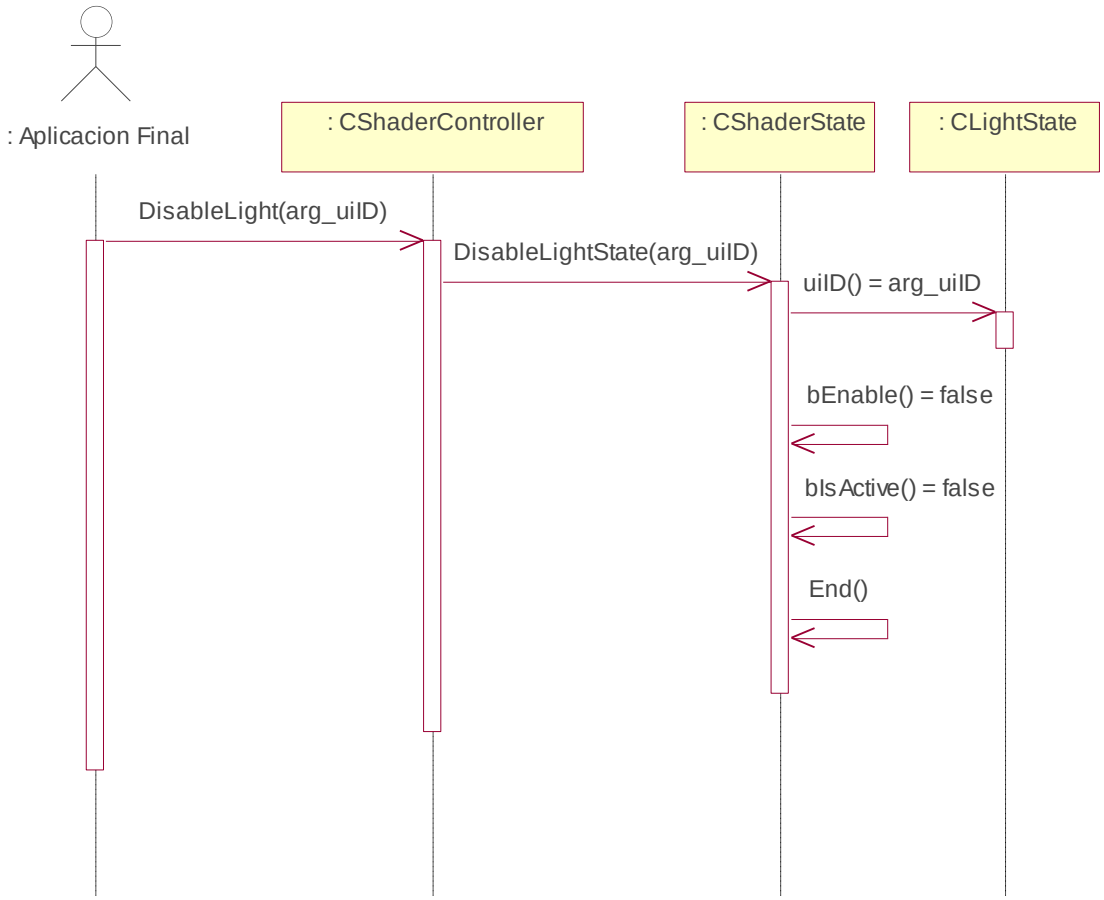

Figura 36: Sección Deshabilitar Luz

4.3.2 Diagramas de Secuencia CU Gestionar Sombra

Sección “Crear Sombra”

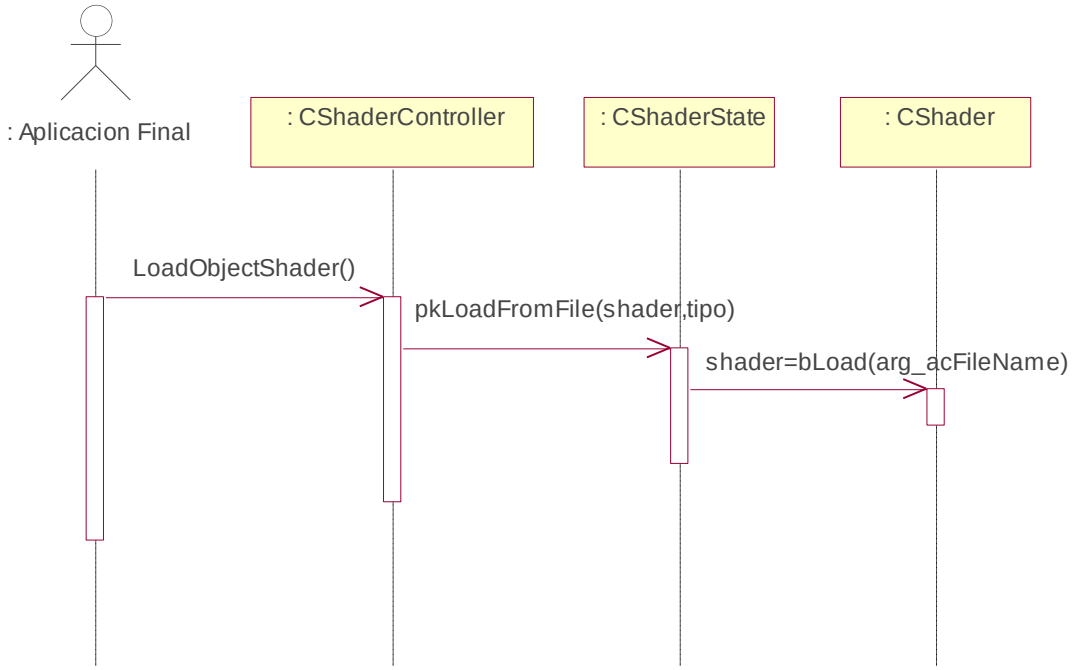

Figura 37: Sección Crear Sombra

Sección “Actualizar Sombra”

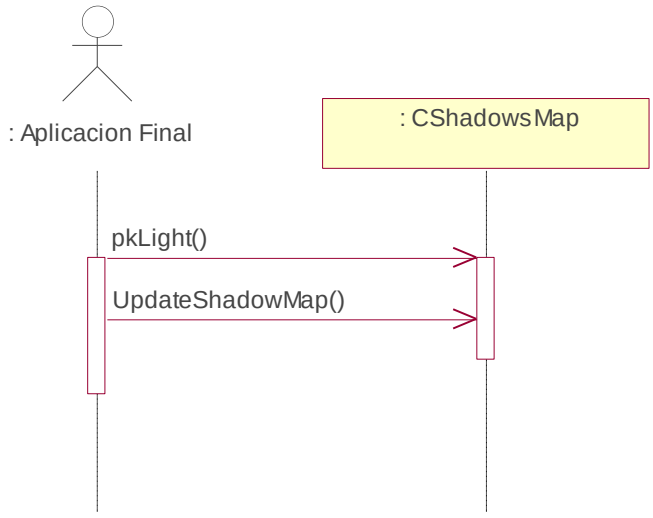

Figura 38: Sección Actualizar Sombra

Sección “Habilitar Sombra”

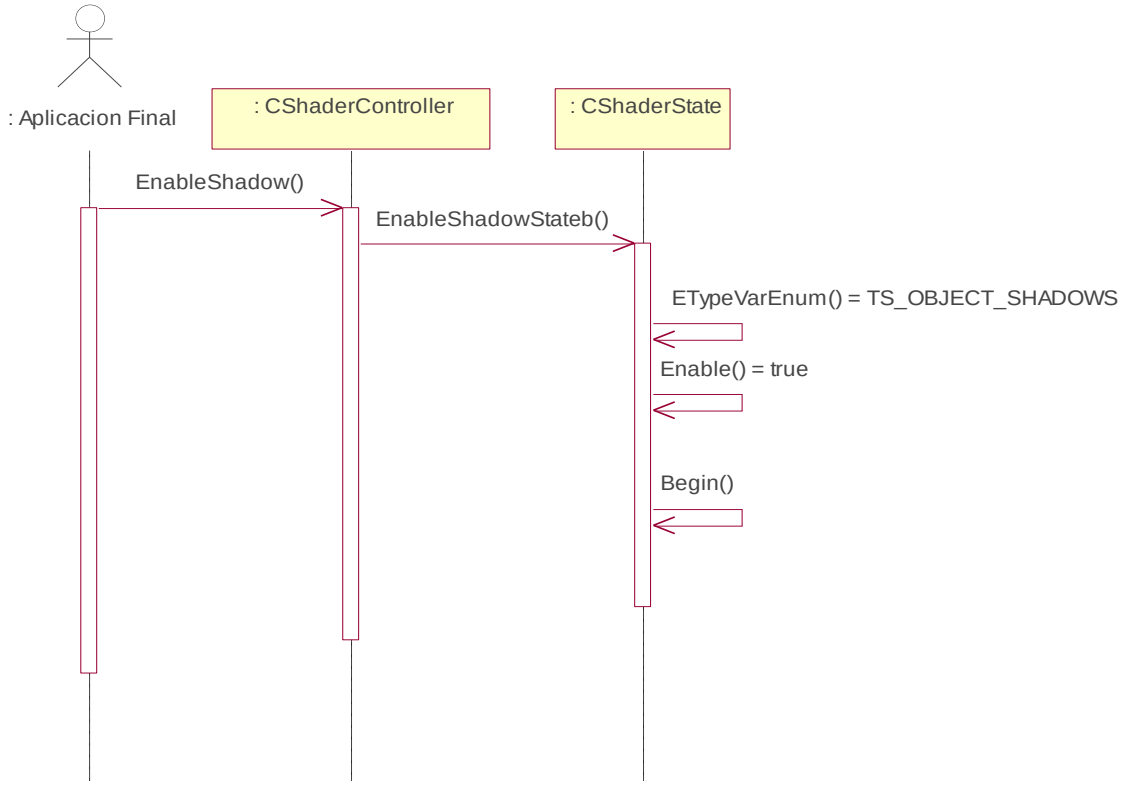

Figura 39: Sección Habilitar Sombra

Sección “Deshabilitar Sombra”

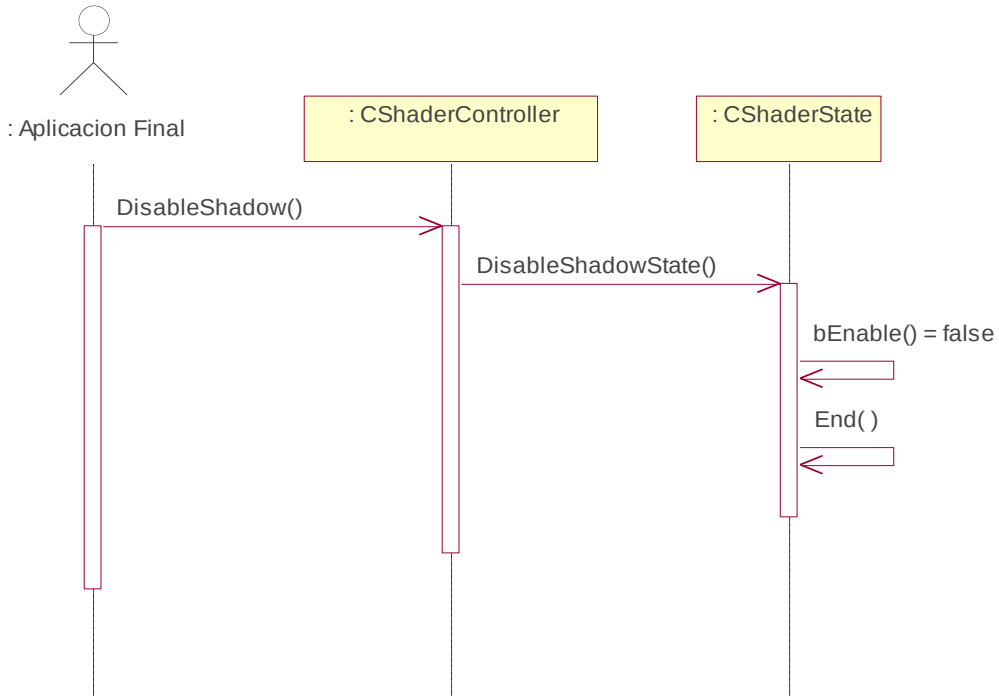

Figura 40: Sección Deshabilitar Sombra

4.3.3 Diagramas de Secuencia CU Cargar vertex-fragmet shader

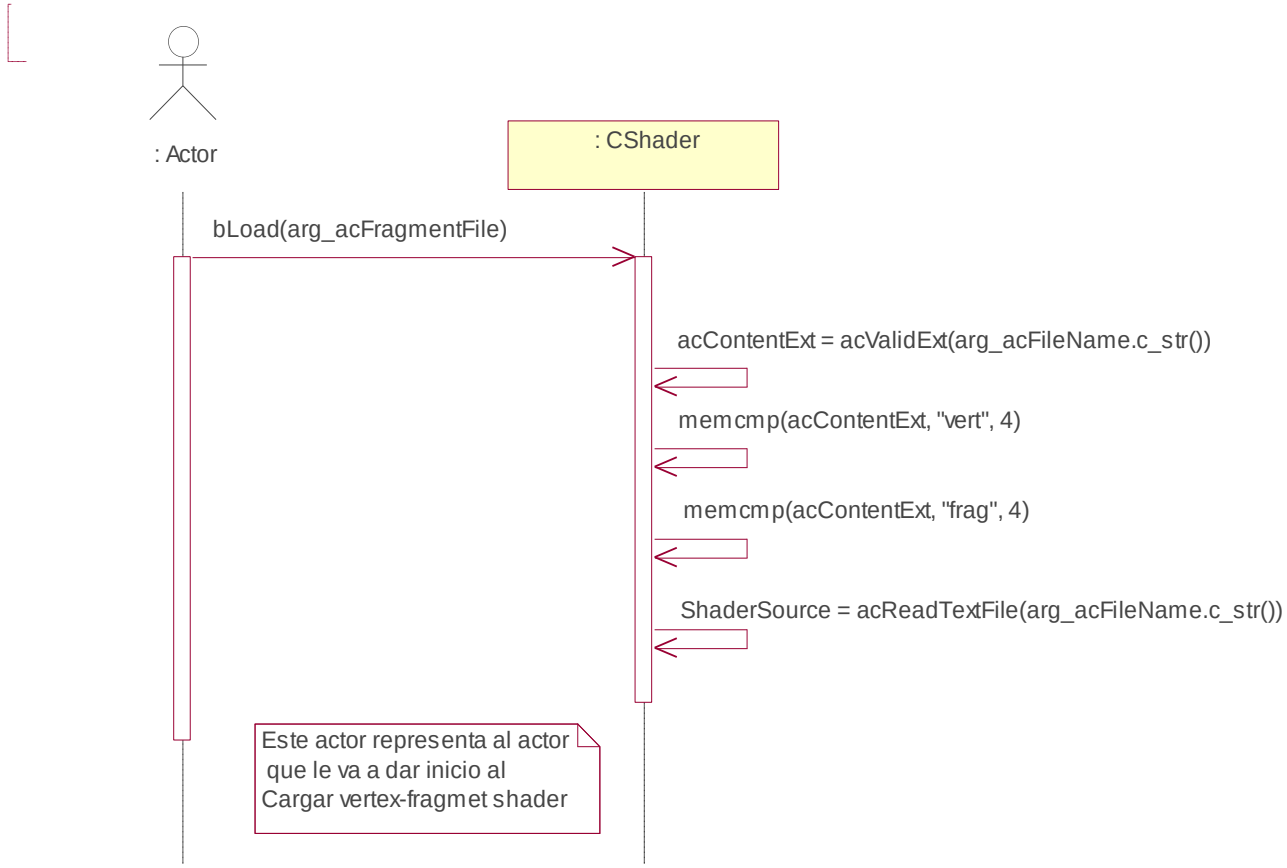

Figura 41: Diagramas de Secuencia CU Cargar vertex-fragmet shader

4.3.4 Diagramas de Secuencia CU Cargar Fichero

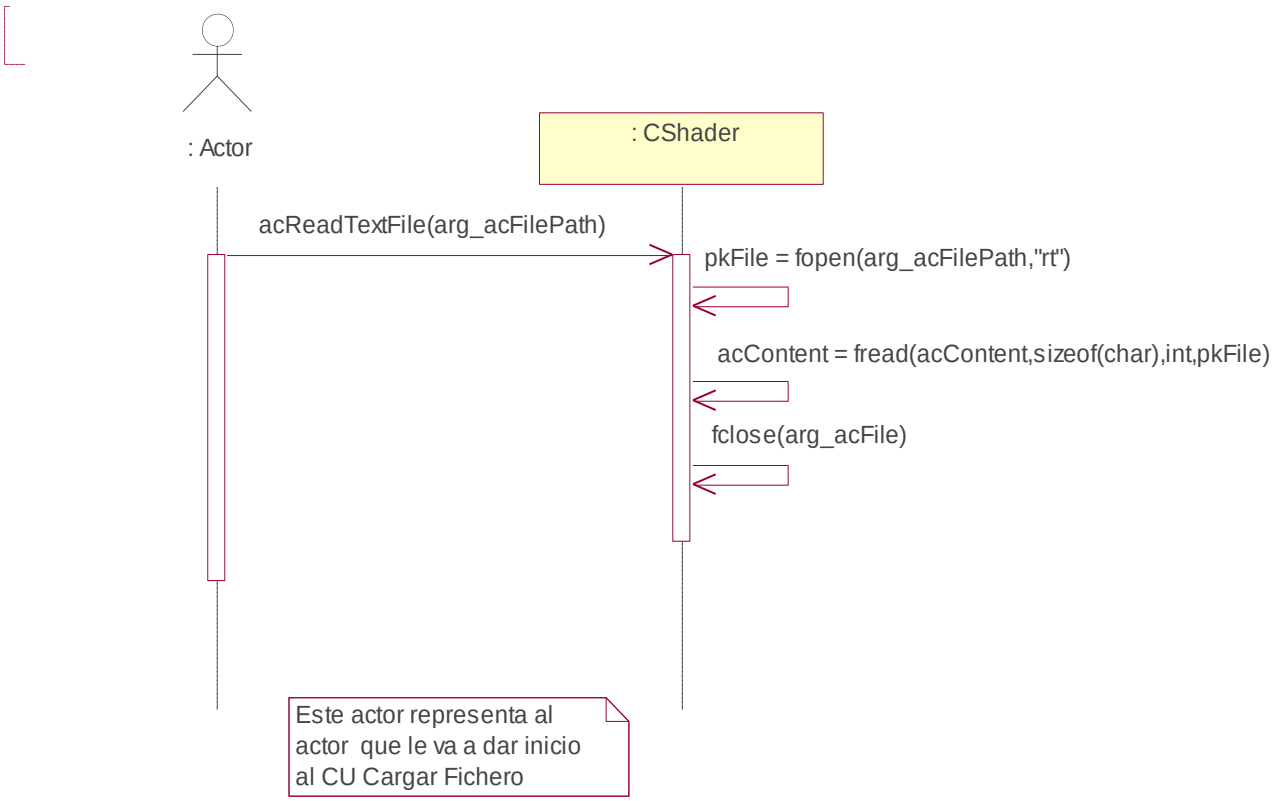

Figura 42: Diagramas de Secuencia CU Cargar fichero.

4.4 Diagrama de Componentes

En el diagrama de componentes que se muestra a continuación se describe un apartado del sistema, que es otra forma de representar una vista estática del sistema, que representa la organización y dependencia que haría entre los componentes físicos que se necesitan para ejecutar la aplicación

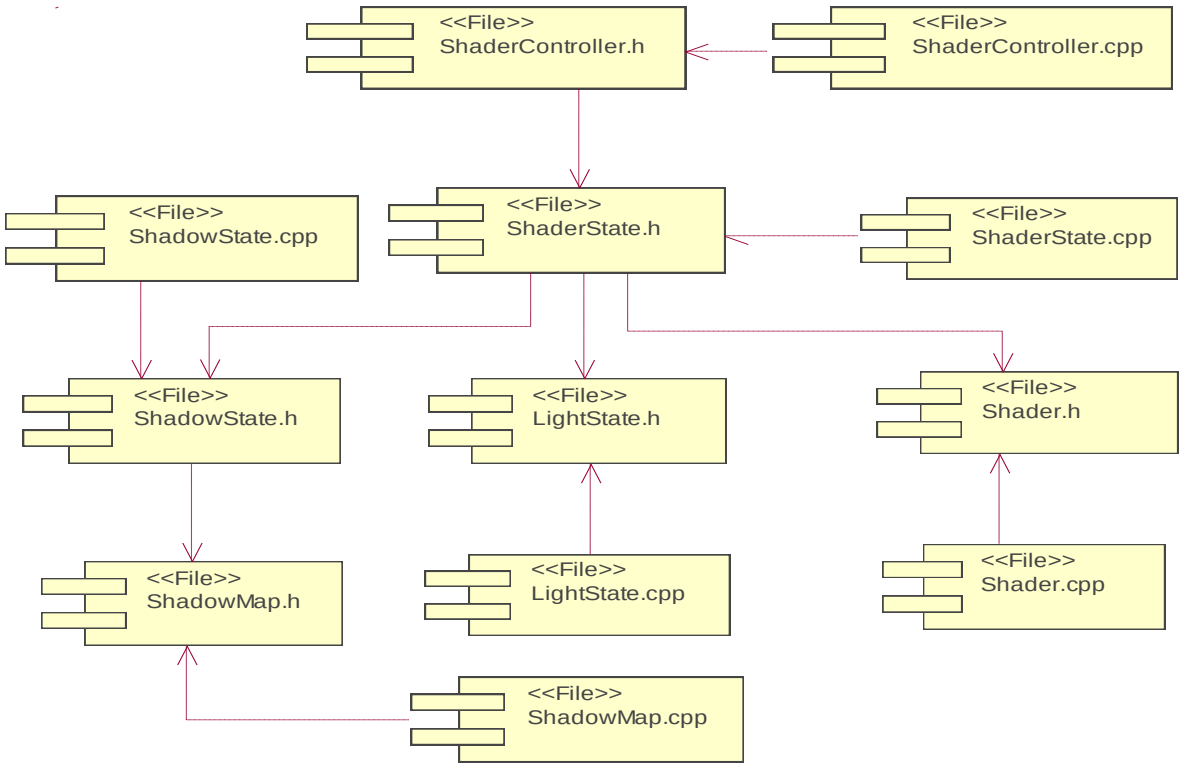

Figura 43: Diagrama de Componentes

4.5 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++ (indexado, 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:

ClassName.h

ClassName.cpp

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**M**ove**T**ype**L**ight {**MTL**_NONE, **MTL**_FREE_LIGHT,**MTL**_MANUAL_LIGHT };

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**lassName;

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_b**VarName, float **arg_f**VarName);

~**C**ClassName ();

Funciones:

bool **b**Function1 (…);

char* **ac**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 **i**MyVar; //variable

Obtención del valor:

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

Establecimiento del valor:

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

Obtención y establecimiento del valor:

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

Conclusiones

En este momento se encuentra todo preparado para pasar a la etapa de programación de los casos de uso desarrollados en el primer ciclo a partir de los estándares de programación definidos anteriormente. También se podría tener en cuenta la posibilidad adicional que brinda la herramienta Rational Rose, generar el código fuente de los componentes relacionados con los casos de uso a desarrollar en el primer ciclo.

Conclusiones Generales

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. A partir de esta investigación, se propone una solución factible.

Para la construcción del sistema como tal se realizó un proceso de Ingeniería utilizando el Proceso Unificado del Software (RUP) como metodología de desarrollo, a partir del cual se obtuvo un pequeño módulo funcional que permite el uso de los algoritmos que aquí se implementaron haciendo uso de los shader, y que permite al programador llevar a sus escenas efectos de iluminación y sombreado bastantes realistas.

Recomendaciones

1. Se recomienda la extensión del módulo a la biblioteca gráfica DirectX.
2. Profundizar un poco más en las características y particularidades del lenguaje HLSL anteriormente expuesto.
3. Fusionar Phong con Shadow Map y SH con Shadow Map.

Referencia Bibliográfica

FUENTES ELECTRÓNICAS

[1]. **Matzka, Gerald.** *"Shadow Algorithms".* Computer Science Seminar, Institute of Computer- Graphics, Vienna University of Technology, 2002.

[2]. **Annala, Rami.** "Shadow Maps". [Online] 2002.
[http://www.tml.hut.fi/Opinnot/Tik-111.500/2002/paperit/rami_annala.pdf].

[3]. Shader. [Online] 2006. [http://en.wikipedia.org/wiki/Shader].

[4]. **Green, Robin.** *"Spherical Harmonic Lighting: The Gritty Details."*. Sony Computer Entertainment America, 2003.

[5]. **Dempski, Kelly y Viale, Emmanuel.** *" Advanced Lighting and Materials with Shaders "*. 2005.

[6] **Vázquez, Pau y Susín,Dani.** *" Generación de penumbras con hardware gráfico "*. Dept. LSI – Universidad Politécnica de Catalunya, 2003.

[7]. **Wales, Jimmy y Sanger, Larry.** GPU. [Online] 2007.
[http://es.wikipedia.org/wiki/Graphics_Processing_Unit].

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

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

[10]. **Emilio Pomares Porras. "***Generación Eficiente de Sombras Mediante Algoritmos Continuos*". Dep. LCC Universitat de Málaga, 2003

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

[12]. **J. Rost, Randi**. *"OpenGL Shading Language"*, Second Edition, 2006.

TESIS

[13]. **Muguercia Torres, Lien y Nodarse Valdés Yaíma.** *Luces y Sombras Dinámicas* . Cuidad de la Habana : s.n., 2007.

[14]. **Guzman, Mario E.** *" Uso de tecnologías de hardware gráfico en el apoyo al realismo en los entornos virtuales arquitectónicos"*. Universidad de Colina , 2004.

[15].**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.

[16]. **Carlos Javier Ogayar Anguita.** *"Optimización de Algoritmos Geométricos Básicos Mediante el Uso de Recubrimientos Simpliciales".* Universidad de Granada, 2006.

[17]. **David Wong Aitken, Jorge Alvarado Valderrama.** *"Algoritmos de sombreado y ocultamiento de líneas visibles".* Facultad de ciencias físicas y matemáticas, Universidad Nacional de Trujillo. 2006

LIBROS

[18]. **Luna, Frank D**. *"Introduction to 3D Game Programming with DirectX"*. USA, Word Ware Publishing. 2003.

[19]. **Astle**, **Dave y Hawking, Kevin.** *"OpenGL Game Programming"*. USA, Prima Tech Lublishing. 2001.

[20]. **Birn, Jeremy.** "*Digital lightning and rendering".* USA, New Riders Publishing. 2000.

[21]. **NVIDIA Corporation.** *" NVIDIA GPU Programming Guide"*. 2004.

[22]. **Ivar Jacobson, Grady Booch, James Rumbaugh.** *"Proceso Unificado de Desarrollo de Software".* Person Education.S.A.2000.

Índice de figuras, ecuaciones y tablas

ÍNDICE DE FIGURAS

ÍNDICE DE ECUACIONES

ÍNDICE DE TABLAS

Glosario de Abreviaturas

2D: Dos dimensiones.

3D: Tres dimensiones.

CPU: Unidad Central de Procesamiento.

SH: Siglas en ingles de *Spherical Harmonics que en español quiere decir Esferas Harmónicas* .Técnica utilizada para la generación de iluminación de forma eficiente.

GPU: Unidad de Procesamiento Gráfico.

SM: *Mapeo de Sombras* .Técnica utilizada para la generación de sombras dinámicas.

RV: Realidad Virtual.

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.

FFP: *Fixed – Function Pepeline* (Pepeline de Función Fija).

CU: Caso de uso.

CAD: El diseño asistido por computador remoto (o computadora u ordenador), abreviado como DAO (diseño asistido por computador) pero más conocido por sus siglas inglesas CAD (*Computer Aided Design remote*), es el uso de un amplio rango de herramientas computacionales que asisten a ingenieros, arquitectos y a otros profesionales del diseño en sus respectivas actividades. También se llega a encontrar denotado con una adicional "Dc=0" en las siglas CADD, diseño y bosquejo asistido por computadora (*Computer Aided Drafting and Design*).

CG: *C for Graphics*. Lenguaje de programación de alto nivel para realizar aplicaciones gráficas y lograr mayor eficiencia y realismo.

HDSRV: *Herramientas de Desarrollo para Sistemas de Realidad Virtual* .Biblioteca a la cuál se le acoplara el modulo de luces y sombras dinámicas

T&L: *Iluminación y Trasformación*.

FFP: *Fixed-Function* Pipeline (Pipeline de Función Fija).

HLSL: High Level Shading Language. Lenguaje de Programación de Alto Nivel para realizar aplicaciones gráficas y lograr mayor eficiencia y realismo.

GLSL: Es el acrónimo de OpenGL Shading Language, una tecnología parte del API estándar OpenGL, que permite especificar segmentos de programas gráficos que serán ejecutados sobre el GPU. Su contrapartida en DirectX es el HLSL.

API: *Application Programmer's Interface*, en español Interfaces para programadores de aplicaciones.

SRV: Sistemas de Realidad Virtual.

HDRI: Siglas en Ingles que significa *High Dynamics Range Image*, rango dinámico de una imagen que tiene que ver con el contraste entre las regiones más brillantes y las más oscuras. En la mayoría de las imágenes, el rango dinámico existe entre los valores 0 y 255. Esto es ideal pata la mayoría de los usos de las fotografías.

Glosario de Términos

A:

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

Aliasing: Efecto que se produce en las líneas, especialmente las que están casi horizontales o verticales, que aparecen dentadas o irregulares debido a su representación por píxeles.

D:

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

C:

Color difuso: Color que refleja un objeto cuando recibe una luz directa.

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:

Fragment Shader (FS): Es lo mismo que un PS. El nombre de *fragment* viene de que una escena 3D es proyectada en el plano x-y (2D) donde los puntos son llamados fragmentos. Un fragmento contiene no sólo información de color del punto sino también otra información tal como la posición y las

coordenadas de la textura por ejemplo. Diversos fragmentos pueden ser unidos para mostrar finalmente un píxel en la pantalla.

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

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

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).

Ley de Lambert: La componente difusa de la luz reflejada por una superficie es proporcional al coseno del ángulo de incidencia.

M:

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

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.

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:

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 (PS): 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.

Penumbra: Punto que no está completamente en sombra ni completamente iluminado.

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.

Rasterización: La rasterización es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital, como una pantalla de computadora o una impresora electrónica.

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).

Stencil Buffer**:** el stencil buffer es un buffer que se tiene para manejar el valor de los píxeles. Esta técnica generalmente es aplicada para generar Sombras y Reflexiones en aplicaciones en 3D.

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.

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.

Templates: Plantillas……

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.