UNIVERSIDAD DE LAS CIENCIAS INFORMATICAS

FACULTAD 5 REALIDAD VIRTUAL

Desarrollo de técnicas para la aplicación de efectos de textura en entornos virtuales generados con la herramienta de desarrollo "SceneToolkit"

TRABAJO DE DIPLOMA PARA OPTAR POR EL TÍTULO DE INGENIERO EN CIENCIAS INFORMÁTICAS

Autores:

Robin de la Paz Acosta

Dany Jesús Valiente Prieto

Tutor: Ing. Fernando Jiménez López

Co-Tutor: Ing. Yanoski Camacho Román

Ciudad de la Habana

Junio, 2008

OPINIÓN DEL TUTOR DEL TRABAJO DE DIPLOMA

Título: Desarrollo de técnicas para la aplicación de efectos de textura en entornos virtuales generados con la herramienta de desarrollo “SceneToolkit”

Autores: Robin de la Paz Acosta, Dany Jesús Valiente Prieto

El tutor del presente Trabajo de Diploma considera que durante su ejecución los estudiantes mostraron las cualidades que a continuación se detallan.

Por todo lo anteriormente expresado considero que los estudiantes están aptos para ejercer como Ingeniero en Ciencias Informáticas y propongo que se le otorgue al Trabajo de Diploma la calificación de:

_____

Ing. Fernando Jiménez López

Firma _____

Fecha

DATOS DE CONTACTO

Ing. Fernando Jiménez López.

Graduado de Ingeniería Informática en la Ciudad Universitaria José Antonio Echeverría. Actualmente es Profesor Instructor de la Universidad de Ciencias Informáticas. Imparte la asignatura Gráficos por Computadora y se desempeña como Jefe del Polo de Realidad Virtual de la Facultad 5.

E-Mail: fjimenez@uci.cu

Ing. Yanoski Camacho Román

Graduado de Ingeniería Informática en la Ciudad Universitaria José Antonio Echeverría. Actualmente es Profesor Instructor de la Universidad de Ciencias Informáticas. Imparte la asignatura Sistemas Operativos y es Jefe del Proyecto "Herramienta de Desarrollo de Sistemas de Realidad Virtual", perteneciente al Polo de Realidad Virtual de la Facultad 5.

E-Mail: rcamacho@uci.cu

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:

Robin de la Paz Acosta

_____

Dany Jesús Valiente Prieto

_____

Tutor:

Ing. Fernando

_____

DEDICATORIA

A Nini, Lela, mi papá. A Carmen.

A mi familia y amigos.

Dany

A mi papa, mi mama y mi abuela Ada.

Robin

AGRADECIMIENTOS

Nuestros más sinceros agradecimientos a todos aquellos que han contribuido en la realización de este proyecto, en especial a Leticia, Fernando y Yanoski. A nuestro Comandante en Jefe, Fidel Castro, por darnos la posibilidad de crecer y educarnos en esta Isla, y tener la gran visión de crear la Universidad de las Ciencias Informáticas, a la cual agradecemos además por dejarnos formar parte de esa tropa del futuro y formarnos como los profesionales que seremos.

De Dany

A mi mamá, a mi abuela y a toda mi familia por su amor y apoyo incondicional en cada momento de mi vida, por forjarme además como hombre de bien y estar pendientes de cada paso que doy. Agradecerles a los maestros y profesores que me hayan impartido clases en todos los niveles de enseñanzas, por brindarme la luz del conocimiento. A mis amigos, de la universidad, que me han acompañado en estos 5 años de la carrera y me han apoyado en cada momento de ella.

De Robin

A todas las personas que de una forma u otra pusieron su granito de arena en la elaboración de esta tesis. A toda mi familia, especialmente a mi papá, mi mamá y mi abuela que me han apoyado siempre y han sabido guiarme hasta aquí.

RESUMEN

Entre los avances más novedosos de la actualidad informática se destacan los Sistemas de Realidad Virtual (SRV). Esta tecnología ofrece prestaciones de una manera más segura y económica, puede encontrarse en aplicaciones de la defensa, la medicina, la educación y el entrenamiento.

Lograr el nivel de realismo deseado en los entornos virtuales, requiere de un alto consumo de recursos y a veces no se cumplen las expectativas. Por esta razón, el uso de texturas ha cobrado gran importancia en el mundo de los gráficos por computadora si se quieren lograr efectos visuales de calidad que no sean muy costosos computacionalmente.

Este trabajo propone una solución para incorporar mayor realismo a las escenas virtuales, mediante efectos visuales, usando texturas. Partiendo del estudio de varias técnicas para aplicar estos efectos, se plantean ventajas y desventajas de las mismas, y se propone diseñar e implementar un conjunto de clases acoplables a la herramienta, que permitan incorporar efectos de reflejo a través de la técnica de "Mapeo de Entorno", y de relieve o rugosidad en la superficie de los objetos a través de la técnica "Bump Mapping". Para lograr un mejor rendimiento y aprovechar las potencialidades del GPU, se usan los shaders.

Como resultado se obtienen efectos bastantes realistas mediante un conjunto de clases listas para ser acopladas a la STK.

ÍNDICE

INTRODUCCIÓN

La Realidad Virtual es una simulación tridimensional del mundo real o ficticio, generada o asistida por computadoras mediante la cual se puede lograr que el usuario tenga la sensación de pertenecer a ese mundo virtual o al menos de interactuar con él. Por ejemplo, puede realizar acciones dentro del mundo tales como desplazarse, tocar y levantar los objetos del entorno, escuchar sonidos y experimentar muchas más situaciones que se asemejan al mundo real.

Actualmente la Realidad Virtual es ampliamente utilizada en el mundo, pues tiene un sinnúmero de aplicaciones que van desde el área del entretenimiento hasta la capacitación del personal antes de enfrentarse a las situaciones de la vida real. Sin embargo, lograr que las simulaciones asemejen lo mejor posible la realidad es una tarea bastante compleja que requiere que los programadores tengan un dominio amplio de las técnicas de Programación Gráfica desarrolladas hasta la actualidad y usen herramientas que faciliten su trabajo haciéndolo más transparente, rápido y eficiente.

En el año 1974, con la tesis doctoral Ed Catmul, surgió la técnica de mapeo de texturas la cual llevó al mundo de los gráficos por computadoras a un nuevo nivel de realismo. El mapeo de texturas consiste en aplicar una serie de dibujos o imágenes a las caras de un objeto tridimensional, consiguiendo de esta forma elevar su realismo. Esta técnica proporciona muchas ventajas, ya que permite aumentar enormemente la complejidad visual de un modelo sin necesidad de aumentar su complejidad geométrica. Además, permite no sólo elevar la velocidad de procesamiento al disminuir el número de polígonos de la escena, sino también que los desarrolladores ahorren mucho tiempo y esfuerzo que se emplearía en la modelación de objetos complejos. Esta técnica ha abierto nuevas vías de explotación para multitud de disciplinas como la simulación civil y militar, la arquitectura, el diseño industrial, la aeronáutica, entre muchas otras, siendo por esto ampliamente usada en todo el mundo.

Actualmente *“En cualquier escena que se precie, la iluminación y las texturas suelen tener mayor importancia que la propia geometría que sustenta los modelos”*. [1]

En la Universidad de las Ciencias Informáticas (UCI), en el Polo de Realidad Virtual, existen varios proyectos productivos vinculados a esta rama que trabajan con una herramienta básica de desarrollo denominada "Scene Toolkit"(STK). La misma está compuesta por varios módulos que proporcionan un conjunto de funcionalidades, permitiendo a los programadores desarrollar aplicaciones gráficas en menor tiempo y con menos esfuerzo. Sin embargo, “SceneToolkit” aún está en construcción pues todavía no provee algunas funcionalidades importantes.

El uso de las texturas en la herramienta actualmente es muy básico, por lo que en las escenas creadas se presentan problemas con el realismo, ya sea por la aparición de efectos indeseables en las texturas (como el llamado efecto de interferencia o nieve en texturas en movimiento) o por la ausencia de efectos que vemos cotidianamente en el mundo real como los reflejos o el relieve en los objetos.

Imagine la naturaleza sin reflejos en el agua, en los metales o en cualquier otra superficie brillante. Estos efectos que parecen pequeños detalles suponen una gran diferencia en escenas virtuales tridimensionales generadas por ordenador pues contribuyen a un mayor realismo en las mismas y pueden lograrse aplicando técnicas de mapeo de texturas.

Precisamente este fenómeno constituye el **problema** que se plantea resolver en este trabajo:

- La carencia de técnicas para la aplicación de efectos visuales a través de texturas en la STK.

Para darle solución a este problema, se plantea como **objeto de estudio**:

- Aplicación de texturas en entornos de realidad virtual.

Como **campo de acción**:

- Efectos visuales a través de texturas en entornos de realidad virtual.

Como **objetivo general**:

- Implementar técnicas que permitan incorporarle a la “Scene Toolkit” efectos visuales a través de texturas, e incrementar el realismo de las escenas creadas.

Para dar cumplimiento al objetivo propuesto se realizarán las siguientes **tareas de investigación**:

- Determinar los tipos de textura existentes así como sus características fundamentales y los usos que se le dan.
- Analizar los tipos de filtrado de textura y modos de aplicación.
- Identificar y analizar las bases teóricas del uso de la técnica avanzada de

mapeo de texturas “Mipmapping”, así como las ventajas de su aplicación.

- Especificar algunas de las técnicas que existen para aplicar efectos visuales a través de texturas.
- Seleccionar los efectos que se aplicarán y las técnicas más apropiadas para generarlos.
- Estudiar los algoritmos matemáticos usados en las técnicas seleccionadas.
- Analizar cómo incorporar los efectos de textura, a las funcionalidades de la “Scene Toolkit”.
- Diseñar las clases necesarias para dar solución al problema propuesto.
- Implementar dichas clases y las técnicas seleccionadas para aplicar efectos de textura.

Como resultado de este trabajo, se pretende obtener un conjunto de clases actualizables y acoplables a la herramienta de desarrollo “Scene ToolKit”.

Capítulo 1: Fundamentación Teórica.

Introducción

El mapeo de texturas es una técnica gráfica que consiste en aplicar una serie de dibujos o plantillas a las caras de un objeto tridimensional. Por ejemplo, si pegamos la foto del rostro de una persona en una esfera plana, la habremos representado como una cabeza. Lo mismo ocurre si aplicamos la textura de la madera a un cilindro: obtendremos un tronco bastante real. [1]

Los principales factores a la hora de implementar el mapeo de texturas son la velocidad y la precisión en la aplicación de los mapas. En ciertos casos, prima un factor sobre el otro. En los juegos de consola es mucho más importante la velocidad de representación (para lograr una gran fluidez de juego), que la precisión en los cálculos. En algunos casos suele bajarse la profundidad de bits de las texturas incluso a 16 o 8 bits para ganar velocidad.

En cambio, en simulaciones como las arquitectónicas, es fundamental tener una gran calidad de texturas y de aplicación en la geometría. [1]

Los mapas de textura están formados por arreglos bidimensionales de datos. Cada porción de dato se denomina texel (pixel de textura). Aunque estos mapas son rectangulares, las texturas pueden ser mapeadas sobre cualquier objeto (sea este rectangular o no) como por ejemplo esferas, cilindros, conos y otros objetos tridimensionales.

1.1 Tipos de Textura

Según la dimensión del espacio de textura, se pueden distinguir tres tipos de textura:

Texturas en 1D (Unidimensional): Son imágenes con anchura pero sin altura, o viceversa; tienen un único texel de ancho o de alto. [2]

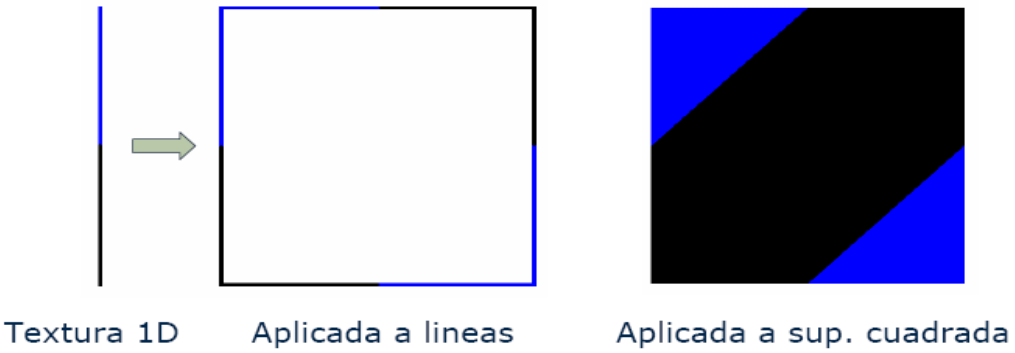


Figura 1: Texturas Unidimensionales

Texturas en 2D (Bidimensional): Son imágenes con más de un texel de alto y ancho. Idealmente se podría pensar que se le está pegando una foto plana a un cuerpo en 3D.

Es el tipo de textura más utilizado.

Al igual que la textura en 1D, se compone de valores de color RGB y pueden incluir valores alfa (de transparencia). [2]

Figura 2: Texturas Bidimensionales

Texturas en 3D (Tridimensional): Las texturas 3d además de cubrir un espacio horizontal o plano, también tienen una profundidad. No se pega una foto, sino que hay un volumen con información de color, el cual es asignado al objeto en 3D. El resultado visual es equivalente al que aparecería si 'esculpiéramos' el objeto.

Se elimina la necesidad de definir nuevas texturas para el interior de la figura.

Necesita mucha memoria. Se utiliza en juegos, visualización médica, etc. [2]

Figura 3: Texturas Tridimensionales

Usualmente los desarrolladores usan texturas 2D (bidimensionales) en sus gráficos. Sin embargo aún no están desechadas las texturas 1D (unidimensionales) y 3D (tridimensionales). [3]

1.2 Funciones de Texturización

Estas funciones tienen como misión determinar el valor RGBA final de cada pixel a visualizar a partir del color de la superficie del objeto a texturizar y del color de la textura. OpenGL permite al desarrollador elegir entre 4 métodos:

Modo Decal: En este modo se diferencian dos casos. Si trabajamos con formato RGB el color utilizado para el objeto es el de la textura. En cambio, con formato RGBA, el color a utilizar es una mezcla del color de la superficie y el de la textura, predominando más uno y otro dependiendo del valor alpha. [1]

Modo de Sustitución (Replace): Similar al anterior. En modo RGB es idéntico. Este método o bien deja el color de la superficie o bien lo sustituye por el color de la textura. [1]

Modo de Modulación (Modulation): Nos permite ir escalando el color final entre el color original de la superficie y el negro dependiendo del grado de luminosidad de la textura (1 – color de la superficie; 0 – negro). Podemos también, en vez de lo anterior, tomar el valor original y modularlo con el valor de la textura. [1]

Modo Mezcla (Blend): Se utiliza para mezclar los valores del color y de transparencia de la superficie con los valores de color y transparencia de la textura. De forma añadida podemos mezclar todo a su vez con un tercer color que determinemos. [1]

1.3 Repetición de la textura

Las dimensiones del espacio de textura suelen estar normalizadas, es decir que todas sus coordenadas están entre 0 y 1. Se realiza una asignación punto a punto. Es

decir, a cada primitiva se le asignan coordenadas uv entre 0 y 1 a cada uno de sus puntos, definiendo un área de correspondencia entre el modelo y la textura.

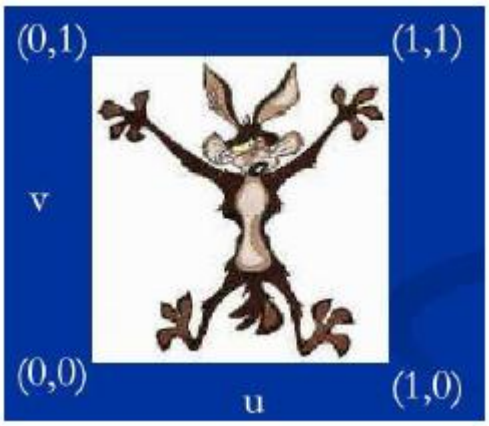


Figura 4: Normalización de la Textura

Por lo tanto cualquier valor de la coordenada x o y mayor que 1 o menor que 0 estaría fuera del rango del mapa de textura.

Cabe preguntarse entonces qué ocurre cuando uno referencia un valor fuera de rango de la imagen. Hay dos posibilidades:

- La primera nos permite repetir los pixels de los bordes de la textura cuando se referencia fuera de ella (**clamping**), pero esto no parece tener mucha utilidad. Podemos optar también porque la textura sólo tenga efecto dentro de sus límites, poniendo el valor alpha como 0 y no sucedería nada en este caso.

Clamping asigna las coordenadas de forma que:

u = min (u, 1); v = min (v, 1) [2]

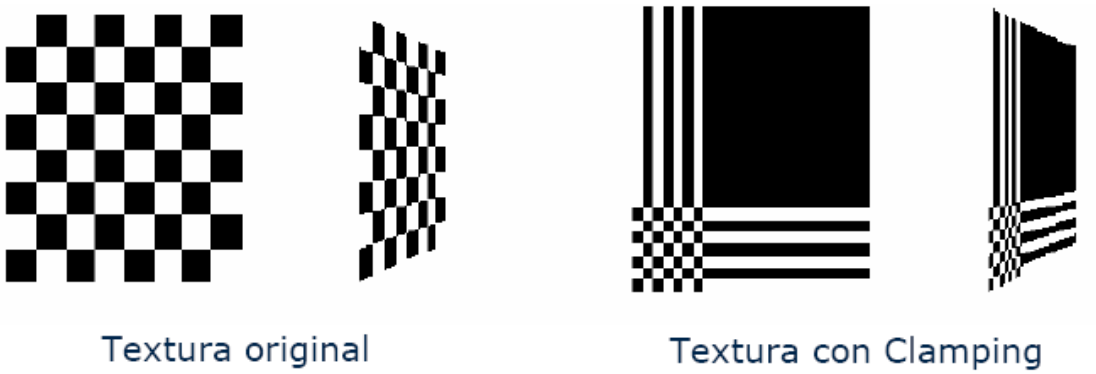


Figura 5: Repetición de la textura, Textura con Clamping

- La otra posibilidad consiste en repetir la textura completa. Es decir que en lugar de tener un mapa con solo una imagen, se tiene un mapa donde la imagen de la textura está repetida varias veces, unas contiguas a las otras (**wrapping**). Permite trabajar con texturas más chicas y ahorrar memoria. Imaginemos por ejemplo el suelo de una habitación que está formado por 20x20 baldosas.

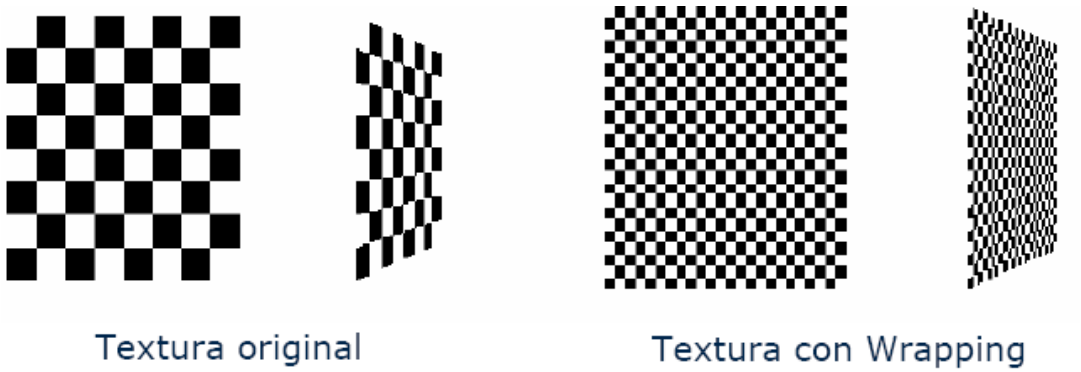


Figura 6: Repetición de la textura, Textura con Wrapping

Por tanto para aplicar una textura que aparezca repetida en el espacio del objeto se pueden asignar coordenadas fuera de rango, que internamente serán recortadas tomando módulo 1.

1.4 Filtrado

Al realizar el mapeo de texturas muchas veces nos encontramos con efectos visuales que disminuyen la calidad del resultado. Estos efectos pueden aparecer por dos razones fundamentales:

1. Los texels al ser proyectados sobre los objetos aparecen mucho mayores que los pixels de la imagen lo cual provoca que la textura se vea como una cuadrícula de texels.
2. Los texels proyectados son mucho más pequeños que los pixels de la imagen por lo que un pixel de la imagen debe representar a muchos pixels del espacio de texturas, provocando que se pierdan texels no representados.

Estos problemas surgen al no cumplirse una relación de semejanza entre el tamaño de los pixels y el tamaño aparente de los texels proyectados sobre el objeto. Para poder suavizar los efectos de esta disparidad y adaptar las texturas, si es posible, se aplican una serie de operaciones o filtros. [4]

1.4.1 Aplicación de Filtros a las imágenes

Existen diferentes filtros que se aplican de acuerdo a cada situación. Estos pueden ser de Magnificación o Minimización (Contracción) y se aplican para especificar que estos pixels y texels serán calculados para la imagen final como se muestra a continuación.

- **Magnificación**: Cuando los texels al ser proyectados sobre los objetos aparecen mucho mayores que los pixels de la imagen, y la textura se ve como una 'cuadrícula'. [2]
- **Minimización**: Cuando los texels proyectados son mucho más pequeños que los de la imagen. [2]

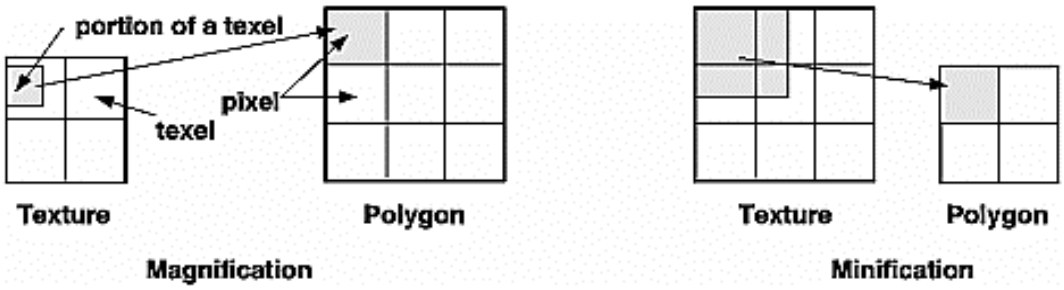


Figura 7: Esquema de los filtros de magnificación y minimización.

Para poder suavizar los efectos de esta disparidad y adaptar las texturas, se aplican los filtros.

1.4.1.1 Filtros de magnificación:

Estos filtros van a indicar cómo calcular el color de un pixel de la imagen cuando los texels ocupan varios pixels. Existen dos tipos los cuales son definidos por la librería gráfica OpenGL:

1. GL_NEAREST (o puntual): Se elige el punto de la textura más cercano a las coordenadas del pixel. Es el filtro por defecto y el menos costoso pero produce el efecto de dentado. [2]

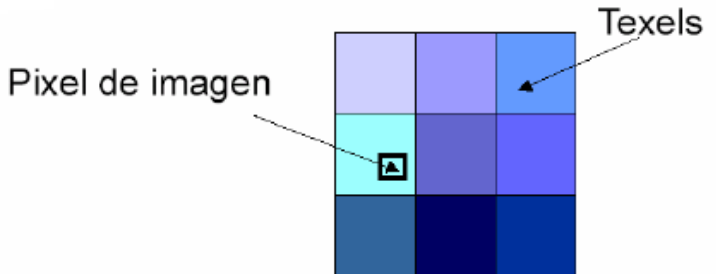


Figura 8: Representación del filtrado puntual Magnificación (GL_NEAREST en OpenGL)

2. GL_LINEAR: se realiza una interpolación lineal (ó promedio ponderado) entre los colores de una matriz de texels de 2×2 que están más cercanos al centro del pixel evaluado, de forma que aparece la misma imagen general pero suavizada. Produce un resultado borroso, pero que elimina el dentado. [2]

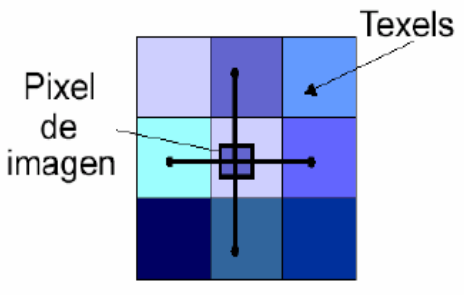


Figura 9: Representación del filtrado mediante interpolación de colores Magnificación (GL_LINEAR en OpenGL)

1.4.1.2 Filtros de Minimización

Estos indican cómo calcular el color de un pixel de la imagen cuando los texels son más pequeños que los pixels. OpenGL también define dos filtros:

1. GL_NEAREST (o puntual): Se elige el texel situado en el centro del pixel. Este filtro tiene el problema de que si la posición cambia ligeramente se producen los efectos de dentado y fluctuación de la textura. [2]

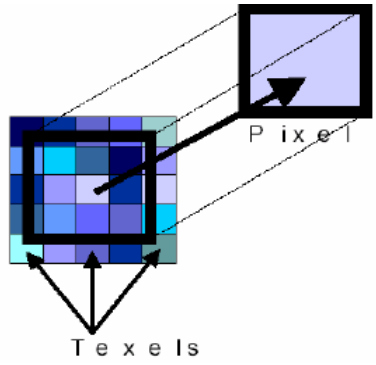


Figura 10: Representación del filtrado puntual Minimización (GL_NEAREST en OpenGL)

2. GL_LINEAR: Es similar al filtro de magnificación linear. Se realiza una interpolación lineal entre los colores de una matriz de texel de 2×2 que están más cercanos al centro del pixel evaluado. [2]

Esta variante es más efectiva pues elimina el dentado pero sólo funciona bien si las diferencias de tamaño no son demasiado grandes.

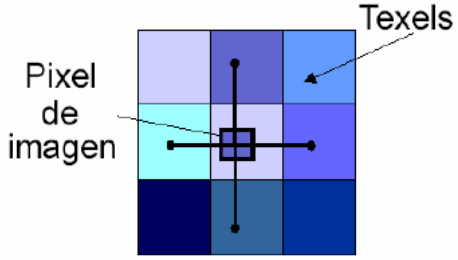


Figura 11: Representación del filtrado mediante interpolación de colores Minimización (GL_LINEAR en OpenGL)

Estos filtros pueden considerarse también como Bilinear y Trilinear según la cantidad de coordenadas que tengan en cuenta para realizar el filtrado.

1.4.1.3 Filtro Bilinear y Filtro Trilinear

Filtro Bilinear: La interpolación se realiza en 2 dimensiones (horizontal y vertical). El filtrado bilinear, define la textura de un pixel como la media de las imágenes de los pixels que lo rodean en el eje X e Y. Sin esta técnica, cada pixel tendría la misma textura que los de alrededor. [1]

Filtro Trilinear: Produce imágenes más realistas pero a la vez es más costoso computacionalmente pues requiere más cálculos. Básicamente se realizan 2 interpolaciones bilineares para cada pixel, correspondientes a los 2 niveles de mipmap más cercanos a la resolución actual con la que se está visualizando el objeto y se promedia el resultado asignándole este al pixel de pantalla.

Por ejemplo si el objeto en este momento se está visualizando a una resolución de 40x40 pixels, los niveles de mipmap que se utilizarían para realizar la interpolación serían los que poseen las resoluciones 64x64 y 32x32.

1.5 Mipmap.

Es una técnica de contracción muy potente y que evita todos los problemas de dentado y oscilación de los colores, pero supone incrementar ligeramente el espacio de almacenamiento de las texturas. Se trata de utilizar una función lineal para interpolar entre los texels más cercanos, pero haciendo que el tamaño de éstos siempre sea comparable al de los pixels. Para ello se guardan diferentes versiones de la misma textura, con diferentes resoluciones. Normalmente el número de texels en cada dimensión (s, t…) es una potencia de dos, y varía también en un factor 2 de un nivel a otro, desde la imagen de mayor resolución hasta llegar al tamaño 1x1.[4]

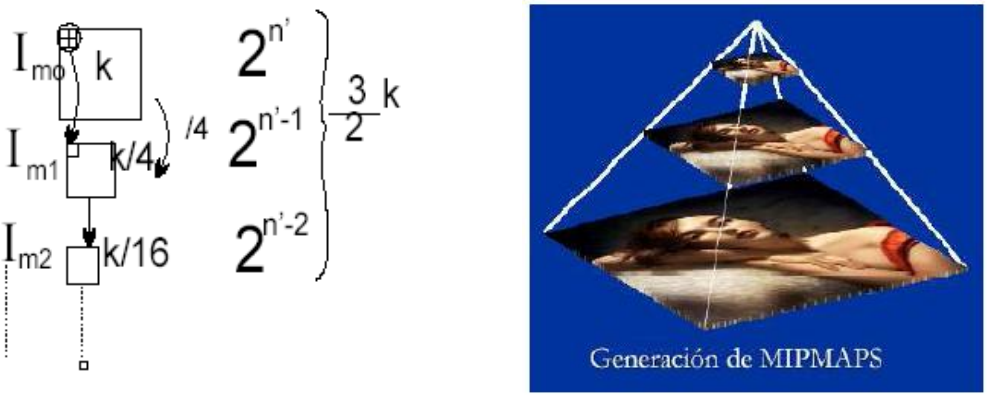


Figura 12: 1-Fórmula de la técnica Mipmap, 2- Generación de Mipmap

1.6 Tipos de Mapeo

En general, el proceso involucra el mapeo matemático de un dominio (el de los pixels de la pantalla) a otro (los pixels de la textura). Dicho proceso matemático se basa principalmente en operaciones con matrices. [1]

Las coordenadas de la textura son asignadas a los vértices del objeto en cuestión, según diversos procedimientos, de tal forma que se identifique cada uno de los pixels de dicho objeto durante el render con uno de los texels de la textura para sustituir, o alterar, alguna de las características de superficie del pixel original, como el color, el brillo o la transparencia, por aquellas que indique la propia textura. En función del tamaño visible del objeto, la resolución de render y la de la propia textura, podrá resultar que un pixel del objeto se corresponda con varios texels, promediando los valores de estos, o que un texel haga lo propio con varios pixels, aplicando entonces sus valores a todos los pixels afectados. [1]

1.6.1 Mapeo sobre el objeto:

Consiste en aplicar la textura directamente sobre el objeto. Es el ideal para suelos, paredes y demás superficies lisas. No suelen obtener resultados apropiados para superficies que no sean planas. [2]

1.6.2 Mapeo en dos partes:

Para solventar los problemas que presenta el sistema anterior se mapea primero la textura como si fuese una superficie 3D (S-mapping), ya sea plana, cilíndrica, cúbica o esférica para posteriormente aplicar ésta sobre la propia superficie tridimensional (O-mapping). [2]

1.6.2.1 Mapeo plano.

Lo único que realiza este tipo de mapeo es colocar la imagen sobre la superficie. En objetos con relieve este tipo de mapeo se suele emplear la variante de envoltura, en el que la textura se acomoda a la forma del objeto sobre el que se aplica. Como resultado, la textura se deformará cuanto más agudizada sea la distorsión del objeto. [1]

El mapeo plano sólo es realmente adecuado para superficies con caras planas, como pueden ser los objetos cúbicos o planos (proyectores de cine, TV, suelos y paredes). Es desaconsejable en situaciones donde las protuberancias o ángulos en los distintos polígonos son muy acusados, a no ser que se utilice una textura especialmente diseñada para ello. [1]

1.6.2.2 Mapeo cúbico.

Es una variante del mapeo plano, ya que continúa proyectando la imagen de forma plana, pero aplicando la textura desde los tres ejes de coordenadas. [2]

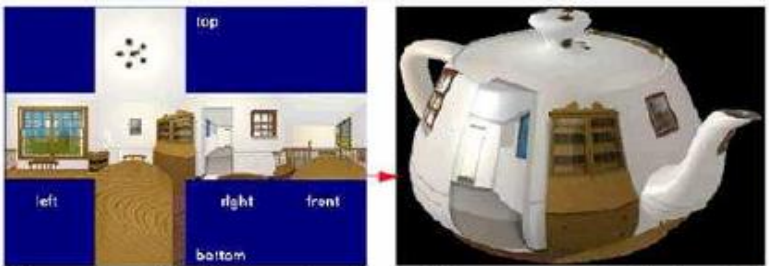


Figura 13: Mapeo Cúbico – Aplicación a Superficies 3D

1.6.2.3 Mapeo esférico.

La textura se contrae en los polos de la imagen y se expande por el centro, por lo que las imágenes se distorsionan para adaptarse a la forma esférica. Es el ideal para objetos esféricos de superficie lisa, como balones, globos, etc. [2]

Figura 14: Mapeo Esférico

1.6.2.4 Mapeo cilíndrico.

Es el más conveniente para objetos cilíndricos como tubos, troncos, etc. No se producen distorsiones porque simplemente envuelve al objeto siguiendo uno de sus ejes. [2]

Un buen modo de obtener una imagen de mayor calidad sobre una superficie cilíndrica es aplicar una fórmula matemática para saber la longitud de circunferencia del objeto en cuestión, L=2.š.r (la longitud de la circunferencia del cilindro, donde r es el radio del cilindro). Lo más sencillo es calcular el radio. Prácticamente cualquier programa permite, como mínimo, analizar la posición de cada objeto y/o punto en el espacio tridimensional, y con la longitud calculada se puede diseñar con facilidad una textura con las mismas dimensiones en un programa de tratamiento de imágenes. [1]

Figura 15: Mapeo Cilíndrico

1.6.2.5 Mapeo UV

Posiblemente se acerca más a la realidad que cualquier otro, puesto que la textura está “anclada” a las coordenadas UV correspondientes (las coordenadas UV son como la longitud y la latitud en una esfera, un valor determinado por dos números que indica la posición exacta de un punto en el espacio). Sólo se tiene que colocar la textura, y retorcer, estirar o doblar la figura, para que la textura aplicada se acomode a los nuevos valores. Aunque sus ventajas son numerosas en cuanto a texturizar objetos deformados o moldeados, donde verdaderamente se aprecia el poder del mapeo UV es en la animación de personajes. [1]

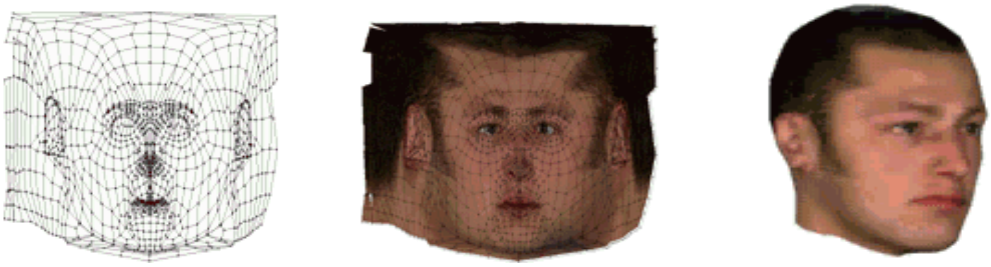


Figura 16: Mapeo UV

1.7 Técnica de Mipmapping

En un entorno virtual las texturas pueden verse desde muchas distancias, y como consecuencia, en múltiples ocasiones estas deben aparecer en la pantalla más pequeña de lo que realmente son. Es decir, con un nivel de detalle menor del que realmente tienen. Cuando sucede esto, si no se aplica ningún algoritmo especial, se obtiene un resultado desagradable en la imagen pues esta se dibuja como si tuviera un efecto de interferencias que se hace más notable si dicha imagen de textura está en movimiento.

Para dar solución a este problema y evitar así este efecto visual tan desagradable -que le resta realismo a las escenas virtuales-, se usa técnica denominada *Mipmapping que* fue introducida en 1983 por Williams L. Mediante esta una imagen de textura original es escalada y filtrada en múltiples resoluciones, de modo tal que cada imagen subsiguiente (nivel de mipmap) tiene la mitad de resolución de la anterior y ¼ del tamaño de la misma.

Las texturas aplicadas son típicamente cuadradas y tienen una amplitud de lado (número de pixels) que sea potencia de 2.

Por ejemplo, si la textura original es un Bipmap de 256x256 pixels, el mipmap asociado tendría un total de 8 imágenes, cada una con ¼ del tamaño de la anterior: 128×128 pixels, 64×64, 32×32, 16×16, 8×8, 4×4, 2×2 y 1×1, esta última representando un simple pixel. [3]

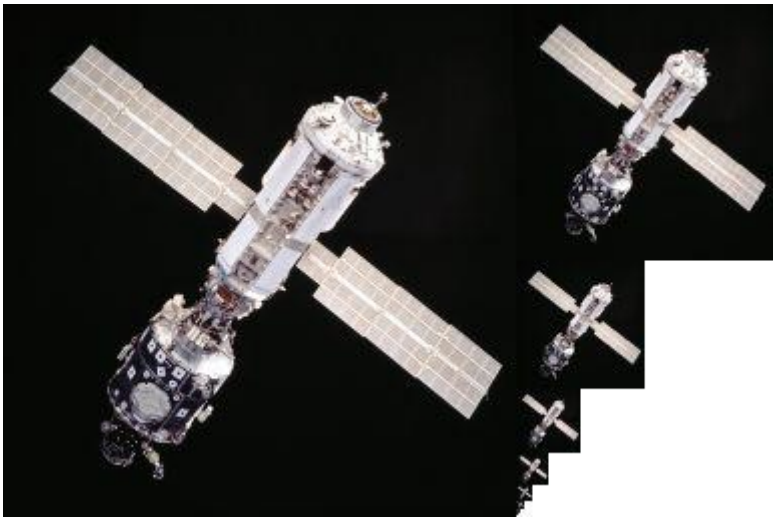


Figura 17: Ejemplo de Mipmap. A la izquierda se muestra la imagen principal acompañada de copias filtradas con un tamaño reducido.

Cada una de estas texturas escaladas representa un nivel de mipmap que se debe aplicar al objeto a una determinada distancia. Esto permite elegir la textura que

se adapte mejor a la escala de dibujado en pantalla, según la distancia a la que se encuentra el objeto del punto de visión. De esta forma se logra que se vea mejor la textura desde todas las distancias, incrementando significativamente su calidad visual.

Esta técnica requiere de un modesto incremento del uso del espacio de almacenamiento, equivalente a 1/3 del tamaño de la textura original, pues la suma de las áreas: 1/4 + 1/16 + 1/64 + 1/256 +… converge a 1/3). [3]

Aunque la textura original va a ser usada en ocasiones mostrando el máximo nivel de detalle, el render pondrá la textura adecuada a medida que el objeto se aleje o se acerque del punto de visión. Esto permite, además, aumentar la velocidad del render, debido a que el número de pixels de texturas que están siendo procesados, es mucho menor que el que se procesa cuando no se usa la técnica. [3]

En OpenGL para que tenga efecto el algoritmo de mipmap se necesita elegir uno de los métodos de filtrado descritos a continuación:

GL_NEAREST_MIPMAP_NEAREST Utiliza la imagen más cercana a la resolución de la pantalla. Usaremos el filtro GL_NEAREST cuando usemos este mapa. [1]

GL_NEAREST_MIPMAP_LINEAR Utiliza la imagen más cercana a la resolución de la pantalla. Usaremos el filtro GL_LINEAR cuando usemos este mapa. [1]

GL_LINEAR_MIPMAP_NEAREST Interpola linealmente entre las dos imágenes más cercanas a la resolución de la pantalla. Usaremos el filtro GL_NEAREST cuando utilicemos este mapa. [1]

GL_LINEAR_MIPMAP_LINEAR Interpola linealmente entre las dos imágenes más cercanas a la resolución de la pantalla. Usaremos el filtro GL_LINEAR cuando utilicemos este mapa. [1]

Los filtros GL_LINEAR_MIPMAP_NEAREST y GL_LINEAR_MIPMAP_LINEAR son los más costosos porque necesitan mucho tiempo de cálculo. [1]

La librería gráfica OpenGl proporciona 2 variantes para aplicar esta técnica. La primera de ellas consiste en definir mipmaps igual a como se definen normalmente las texturas con la función *glTexImage2D( ),* teniendo en cuenta que con cada llamada a esta función se define un único nivel de mipmap para determinada resolución de la textura. El nivel 0 representa a la textura original de mayor resolución, el nivel 1 la mitad de la resolución del nivel 0 y así sucesivamente. En el siguiente ejemplo la textura de resolución 64x64 representa el nivel 0.[3]

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST_MIPMAP_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 64,64,0, GL_RGB, GL_UNSIGNED_BYTE, texImage0);
glTexImage2D(GL_TEXTURE_2D, 1, GL_RGB, 32,32,0, GL_RGB, GL_UNSIGNED_BYTE, texImage1);
glTexImage2D(GL_TEXTURE_2D, 2, GL_RGB, 16,16,0, GL_RGB, GL_UNSIGNED_BYTE, texImage2);
glTexImage2D(GL_TEXTURE_2D, 3, GL_RGB, 8, 8, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage3);
glTexImage2D(GL_TEXTURE_2D, 4, GL_RGB, 4, 4, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage4);
glTexImage2D(GL_TEXTURE_2D, 5, GL_RGB, 2, 2, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage5);
glTexImage2D(GL_TEXTURE_2D, 6, GL_RGB, 1, 1, 0, GL_RGB, GL_UNSIGNED_BYTE, texImage6);
```

La segunda variante es construir mipmaps automáticamente, usando la librería GLU. Para texturas bidimensionales la función *gluBuild2DMipmaps ()* sustituye las llamadas que normalmente hacíamos a *glTexImage2D ()* y se define como: [3]

```
int gluBuild2DMipmaps(GLenum target, GLint internalFormat, GLint width, GLint height,
                      GLenum format, GLenum type, void *texels);
```

1.8 La técnica de "Mapeo del Entorno" (Environment Mapping)

Imagine la naturaleza sin reflejos en el agua, los metales u otras superficies brillantes. Estos pequeños detalles suponen una gran diferencia para las escenas 3D generadas por ordenador. [17]

Tal vez en un juego por ejemplo esto no tenga mucha importancia porque su principal objetivo es lograr el entretenimiento sin tomar demasiado en serio estos detalles, pero en el caso de los simuladores cobra mayor importancia pues sin estos efectos el entorno carece de realismo.

Lograr representar reflejos con gran realismo cobra gran importancia en el caso de los simuladores. Este efecto puede lograrse utilizando texturas, de forma que al mapearlas, simulen que la superficie del objeto está reflejando su entorno. Esta técnica se conoce como "Mapeo del Entorno" (Environment Mapping).

El Mapeo de Entorno permite a los desarrolladores generar reflejos espectaculares, con un alto grado de precisión e incluso en tiempo real, y efectos de iluminación especular, lográndose escenas tridimensionales (3D) de gran realismo. En la misma se define una función de mapeo que depende de las coordenadas del vector normal a la superficie en cada punto. Esto permite calcular los reflejos sin necesidad de seguir numerosos rayos secundarios, por lo que se reducen en gran medida el número de cálculos necesarios para la simulación.

Obtención de Reflejos

Cuando usted mira fijamente algún punto en una superficie muy reflexiva, la superficie en ese punto refleja el rayo que viaja de su ojo al punto de la superficie dentro del entorno. Las características del rayo reflejado dependen del rayo de visión original y de la superficie normal en el punto donde el rayo de visión alcanza la superficie. [9]

Calculando el vector reflejado

El vector I es el llamado **vector incidente** que va desde el punto de visión hasta la superficie del objeto. Cuando **I** alcanza la superficie este es reflejado en la dirección **R** basado en el vector normal a la superficie. Entonces **R** sería el rayo reflejado. Es posible calcular el vector R en términos del vector I y N con la siguiente ecuación: [9]

$$R = I - 2N(N \bullet I)$$

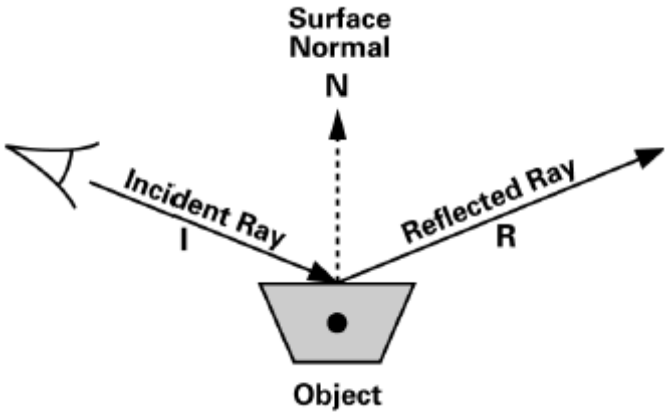


Figura 18: Vector Reflejado

Calcular el vector reflejado es una operación común en los gráficos por computadora por lo que muchas librerías proveen funciones que toman el vector incidente y el vector normal a la superficie y retornan el vector reflejado. [9]

Hay objetos en los que no debe aplicarse esta técnica porque los resultados no serán satisfactorios. Por ejemplo, no producirá resultados correctos para rayos que tengan que rebotar en el objeto varias veces antes de escapar al entorno. [5]

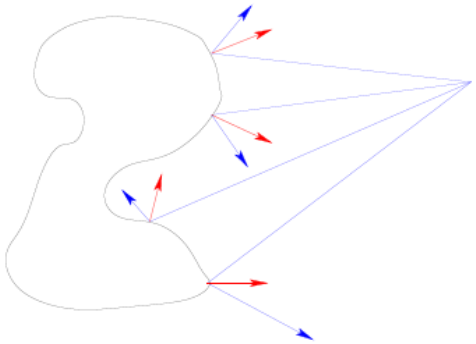


Figura 19: Rayos Incidentes y normales en una imagen que no es plana.

En la imagen anterior las flechas azules representan los rayos reflejados y las rojas las normales a la superficie. Se puede observar como algunos de los rayos reflejados rebotan varias veces en la superficie del objeto antes de escapar al ambiente.

Existen distintas variantes de Environment Mapping las cuales se enumeran a continuación:

1.Mapeo Esférico (*Spherical Environment Mapping*).

2.Mapeo Cúbico (*Cube Environment Mapping*).

3.Mapeo Parabólico (*Parabolic Mapping*).

De estas las más usadas son el mapeo esférico y el cúbico. En la primera se usa una única imagen mientras que en la segunda se usan las 6 imágenes que corresponden a las 6 caras del cubo.

1.8.1 Mapeo Esférico del Entorno

El Spherical Environment Mapping (SEM) o Mapeo Esférico del Entorno es la primera y más simple de las técnicas que permiten simular reflexiones del entorno en la superficie de los objetos. Esta usa una sola textura para la reflexión la cual puede ser cualquiera pero generalmente se usan texturas esféricas listas.[10]

Estas texturas son llamadas mapas esféricos, y pueden ser creadas con photoshop u otros programas, usando el modificador **Spherize**. [10]

El algoritmo para realizar el mapeo esférico es el siguiente:

- 1 - **u** es el vector unitario que va desde la posición de la cámara hasta el vértice actual: esta es la posición del vértice en el *espacio de visión* y es también el vector de visión.
- 2 - **n** es la normal del vértice.
- 3 - **r** es el vector reflejado dada la normal:

r = reflect(u, n)

r = 2 * (n cos u) * n - u

- 4 - **m** es un valor intermedio:

m = sqrt($r.x^2 + r.y^2 + (r.z + 1.0)^2$)

- 5 - **s** y **t** son las coordenadas finales de la textura.

s = r.x / m + 0.5

t = r.y / m + 0.5

Es bastante simple de usar sin embargo tiene sus inconvenientes. La reflexión se desplaza en el objeto y sigue a la cámara, es decir que es dependiente del punto de visión. De esta forma tendrá que si la cámara se mueve alrededor del objeto la reflexión mostrada no será realista porque se verá siempre la misma imagen reflejada. [10]

Desafortunadamente, para los desarrolladores el **SEM** es usado para una sola orientación y un solo punto de visión, limitando su uso a condiciones muy específicas que no pueden ser violadas. Es decir, que aunque puede producir reflexiones satisfactorias, en las condiciones antes descritas prácticamente queda en desuso debido a que se combinan varias limitaciones como la deformación de la imagen (es una consecuencia natural del mapeo esférico), dependencia de punto de visión y la incapacidad reflejar entorno en tiempo real. [18]

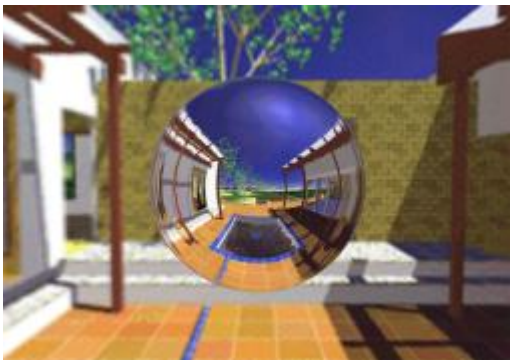


Figura 20: Reflexión Correcta usando Spherical Environment Mapping.

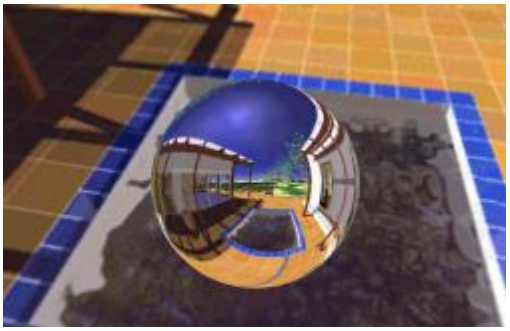


Figura 21: Reflexión incorrecta usando Spherical Environment Mapping

Cambio del Punto de Visión:

El mismo mapa esférico de la imagen es aplicado cambiando el **punto de visión**. La reflexión de la *figura 21* obviamente es incorrecta. No hay forma de que el observador pueda ver la piscina y el reflejo de la piscina al mismo tiempo. [18]

1.8.2 Mapeo Cúbico del Entorno

El Cube Environment Mapping (CEM) o Mapeo Cúbico del Entorno surge como una solución a las limitaciones del mapeo esférico. Esta técnica proporciona los mejores resultados, permitiendo a los desarrolladores crear reflexiones muy precisas y en tiempo real. Requiere de 6 texturas cargadas en memoria al mismo tiempo. Un **cubemap** es un tipo real de textura (igual que las 2D), pero tienen 6 caras correspondientes a los 3 ejes de coordenadas en sentido positivo y negativo y estas se unen para formar un cubo con las aristas alineadas con los ejes.

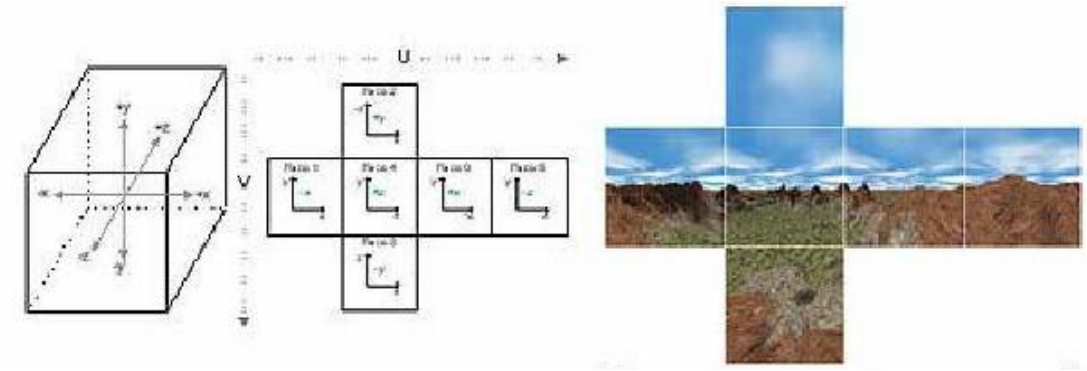


Figura 22: Mapeo Cúbico del Entorno.

En la librería gráfica OpenGL cada una de las seis texturas tiene un nombre que representa determinado eje de coordenadas en sentido positivo o negativo, como se muestra a continuación:

GL_TEXTURE_CUBE_MAP_POSITIVE_X
GL_TEXTURE_CUBE_MAP_NEGATIVE_X
GL_TEXTURE_CUBE_MAP_POSITIVE_Y
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
GL_TEXTURE_CUBE_MAP_POSITIVE_Z
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z [15]

Al cambiar la forma del mapa de textura a una forma cúbica proporciona un camino simple para la creación de imágenes reflectantes. Este cubo engloba el objeto que va a reflejar el entorno. Al ser texturas estándar, planas y alineadas con los ejes son relativamente sencillas de crear.

A diferencia del SEM el CEM es independiente del punto de visión y no se presentan los problemas de deformación y Warping asociado al mapeo esférico.

1.8.2.1 Generación de Texturas en Tiempo Real

Para crear un mapeado cúbico de un objeto, se pueden realizar desde el centro del objeto 6 renderizados orientando la cámara con los distintos ejes principales en direcciones positiva y negativa y usando una perspectiva de 90º. De cada renderizado se puede capturar la imagen generada y guardarla como textura para luego usarla en el mapeado cúbico. Para esta última función existen diversas llamadas OpenGL que dan lugar a distintos rendimientos (glCopyTexSubImage2D, glCopyTexImage2D, glGetTexImage, glCopyTexImage,…). [15]

Mapeo de coordenadas cúbicas

Una vez que tengan el vector reflejado(R) en determinado punto, la coordenada de R con la magnitud más grande determina la cara. Las restantes dos coordenadas son divididas entre el valor absoluto de la coordenada más grande y es trazado un mapa en el rango [0.0 - 1.0]. [10]

Ejemplo:

En el vector R = {0.287,-0.944, 0.164} se selecciona la cara de NEG_Y. Las coordenadas de textura {s, t} son calculadas como sigue:

s = (0.287/0.944*0.5) + 0.5 y t = (0.164/0.944*0.5) + 0.5 entonces {s, t} = {0.65, 0.58}. El vector R es el mismo en cuanto al DPEM. [10]

1.8.3 Mapeo Parabólico:

Se trata de un método que utiliza la parábola como herramienta matemática para realizar el render de manera que se obtienen mejores resultados que con el 'mapeo affine' con el inconveniente de reducir algo la velocidad. [1]

Para explicar este método vamos a recurrir a un ejemplo. Supongamos que se quiere mapear una textura yendo desde una coordenada x = 0 hasta otra con coordenada x = 10. La ecuación cuadrática tanto para u como para v (a partir de ahora nos centraremos en las ecuaciones de u ya que las de v son exactamente iguales $ax^2 + bx + c$. [1]

Calcularemos los valores de los coeficientes usando el punto inicial, el final y otro a nuestra elección, por ejemplo el punto intermedio, con lo que tenemos que $x_0 = 0$, x_2

= 10 y $x_1 = x_2/2 = 5$, y podemos construir el siguiente sistema de ecuaciones cuya resolución nos dará el resultado que buscábamos: [1]

$u_1 = ax_1^2 + bx_1 + c$

$u_2 = ax_2^2 + bx_2 + c$

$u_3 = ax_3^2 + bx_3 + c$

Si resolvemos el sistema de ecuaciones obtenemos que:

$a= -u_2 + 2u_1 - u_0 / x^2$

$b= u_2 - u_0 - ax_2 / x$

$c= u_0$

Una vez que tenemos esto, pasamos a conseguir, a partir de las derivadas primera y segunda de la ecuación, los valores que nos van a permitir realizar el mapeo. [1]

du = ax + b

ddu = a

De esta manera nuestro bucle interno tendrá la siguiente forma:

Para cada pixel <screenX, screenY> correspondiente a la línea de barrido, hacer col = texel de la textura en la posición <u, v>

Dibujar el pixel en la posición <screenX, screenY> con el color 'col' [1]

u += du

du += ddu

v += dv

dv += ddv

1.9 Bump Mapping

1.9.1 ¿Qué es Bump Mapping?

Bump Mapping es una técnica de gráficos computacionales 3D que consiste en dar un aspecto rugoso a las superficies de los objetos. Esto puede ser claramente aplicado cuando se desea dar un efecto de relieve en el objeto. Esta técnica modifica las normales de la superficie, sin modificar su geometría. Lógicamente, las normales originales de la superficie serán perpendiculares a la misma. Bump Mapping se encarga de eliminar esa perpendicularidad y modificar estas normales para lograr el efecto deseado, todo ello sin modificar en ningún momento la topología ni la geometría del objeto. El resultado es bastante rico y detallado, y pueden lograrse grandes parecidos a elementos naturales (como puede ser la textura de una naranja). La diferencia entre mapeado de desplazamiento (displacement mapping) y bump mapping

es evidente: en bump mapping no se perturba la geometría (sino que se perturba la dirección del campo normal). [7]

1.9.2 Ecuaciones de Iluminación

El color final del pixel que aparece en la pantalla está dado por la siguiente ecuación:

If = Ia + Id + Is

Donde **If** es la intensidad del color final del pixel, **Ia** es la intensidad del color ambiente, **Id** es la intensidad del color difuso y **Is** es la intensidad del color especular.

Ia, Id y Is son vectores 4d RGBA. [16]

El término **Ia** es el componente ambiental, Ia es el resultado de la multiplicación entre la componente ambiental de la luz y del material que compone la superficie del objeto 3d:

Ia = Al * Am.

Donde **Al** es la componente ambiental de la luz y **Am** la del material. Ia es generalmente una vector RGBA constante y su valor es el mismo independientemente del pixel. [16]

El término **Id** expresa la componente difusa final. Esta componente viene dada por la siguiente ecuación:

Id = Dl * Dm * LambertTerm

Donde **Dl** es la componente difusa de la luz y **Dm** la del material. El factor **LambertTerm** es la piedra angular dentro de la ecuación de iluminación. De hecho el valor de este factor hará posible la creación de sombra en un objeto 3d (auto-sombreado). Este coeficiente de Lambert es calculado con el siguiente producto escalar:

LambertTerm = max (N dot L, 0.0) [16]

Donde **N** es el vector normal al pixel y **L** el vector de la luz al mismo pixel. Esta simple relación, pero fundamental, nos dice que el valor del coeficiente será máximo (1.0) si el ángulo entre los dos vectores (L y N) es igual a cero, por ejemplo si el pixel esta directamente al frente de la luz. Para todos los demás casos, el coeficiente Lambert estará entre 0.0 y 1.0, lo cual generará su propia sombra. [16]

La función **max ()** previene que se pase algún valor negativo por el término Lambert. [16]

El término **Is** expresa la componente especular final. Esta componente es obtenida por:

Is = Sm x Sl x pow (max (R dot E, 0.0.), f)

El término **Is** es el más complicado de calcular por ser el responsable de la famosa reflexión especular en la superficie de los objetos. **Sl** es la componente especular de la luz y **Sm** la del material. E es el vector visión o el vector cámara y R es el vector reflejado de la luz **L** en relación con la normal **N**. **R** es obtenido con:

R = reflect (-L, N) [16]

Donde N es el vector normal del pixel analizado, L es el vector de luz y la función reflect() (disponible en GLSL), con la cual es posible calcular el vector reflejado de L en relación a N. la función **pow()** es la función que permite expresar la potencia de N a la p, pow (N, p). f es el factor exponencial especular (el famoso shininess en OpenGL) que representa la dureza y la precisión de la reflexión especular. [16]

La técnica Bump mapping precisamente consiste en darle mas vida y sparkling al pobre vector N (Normal), dibujando por cada pixel un vector normal en el mapa normal. [16]

1.10 Shader

1.10.1 Unidad de Procesamiento Gráficos (GPU)

GPU es un acrónimo utilizado para abreviar **Graphics Processing Unit**, que significa "Unidad de Procesado de Gráficos". [8]

Una GPU es un procesador dedicado exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y o 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 otro tipo de cálculos (como la inteligencia artificial o los cálculos mecánicos en el caso de los videojuegos). [8]

Una GPU implementa ciertas operaciones gráficas llamadas primitivas optimizadas para el procesamiento gráfico. Una de las primitivas más comunes para el procesamiento gráfico en 3D es el *antialiasing*, que suaviza los bordes de las figuras para darles un aspecto más realista. Adicionalmente existen primitivas para dibujar rectángulos, triángulos, círculos y arcos. Las GPU actualmente disponen de gran cantidad de primitivas, buscando mayor realismo en los efectos. [8]

1.10.2 ¿Qué son los Shader?

Shader -en informática gráfica-, es el término usado para referirse a un conjunto de instrucciones capaces de ser ejecutadas por el procesador gráfico (GPU). En otras

palabras, es un programa para la tarjeta gráfica. Su uso está vinculado comúnmente a aplicar efectos a la representación y usar la tarjeta gráfica para cálculos pesados como por ejemplo la detección de colisiones y otras aplicaciones.

Según el momento en que se aplica el código a la representación a lo largo del Pipeline de dibujado, se diferencian 3 tipos de shaders:

- Vertex Shader: Actúa sobre las coordenadas, color, textura, etc. de un vértice.
- Geometry Shader: Es capaz de generar nuevas primitivas dinámicamente.
- Pixel Shader: Actúa sobre el color de cada pixel del objeto.[6]

La utilización del Vertex Shader y del Pixel Shader ha permitido a los programadores una mayor libertad a la hora de diseñar gráficos en tres dimensiones, ya que puede tratarse a cada pixel y cada vértice por separado. De esta manera, los efectos especiales y de iluminación pueden crearse mucho más detalladamente, sucediendo lo mismo con la geometría de los objetos. Recursos como las operaciones condicionales o los saltos se utilizan de forma similar que en los lenguajes más conocidos. Sin los shaders, muchos de los efectos eran realizados en conjunto con la unidad de procesamiento central, disminuyendo en gran medida el rendimiento y limitando el avance a nivel gráfico de los mismos.

La siguiente figura es un diagrama simplificado de las etapas del pipeline y los datos que viajan a través de estas. Aunque extremadamente simplificado, es lo suficiente como para presentar importantes conceptos de la programación Shader. En esta subdivisión la funcionalidad fija de pipeline es presentada. Note que este pipeline es una abstracción y no necesariamente encuentra alguna implementación en particular en todos sus pasos. [19]

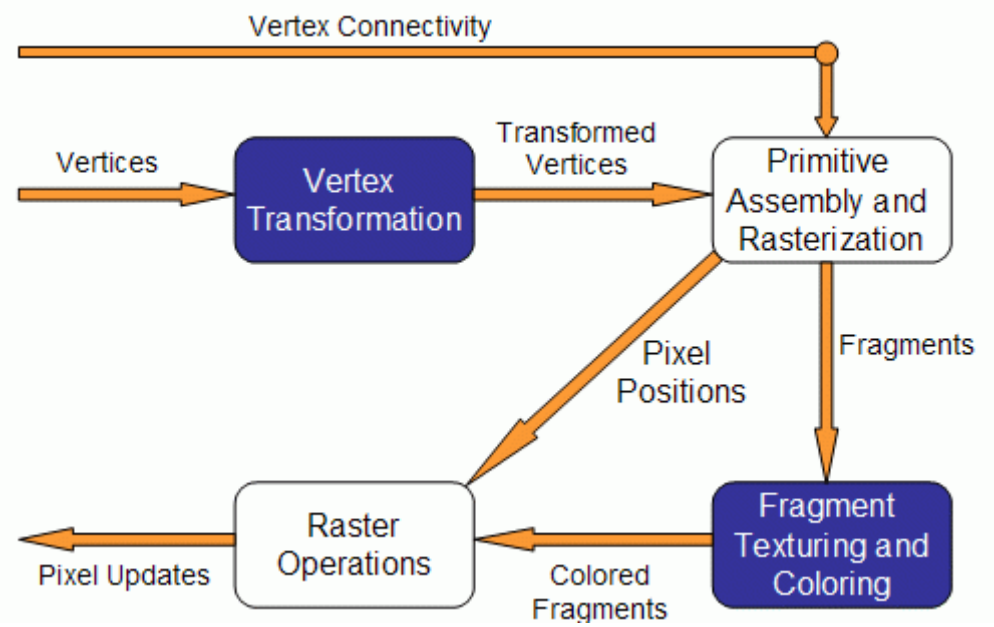


Figura 23: Diagrama simplificado de las etapas del Pipeline

1.11 Lenguajes de Programación

1.11.1 C++

El **C++** (pronunciado *"ce más más"*) es un lenguaje de programación, diseñado a mediados de los años 1980, por Bjarne Stroustrup, como extensión del lenguaje de programación C. [12]

Se puede decir que C++ es un lenguaje que abarca tres paradigmas de la programación: la programación estructurada, la programación genérica y la programación orientada a objetos. [12]

Las principales características del C++ son las facilidades que proporciona para la programación orientada a objetos y para el uso de plantillas o programación genérica (*templates*). [12]

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. [12]

1.11.2 Principales Librerías Gráficas.

OpenGL es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. Fue desarrollada por Silicon Graphics Inc. (SGI) en 1992. Su nombre viene del inglés Open Graphics Library, cuya traducción es librería de gráficos abierta (o mejor, libre, teniendo en cuenta su política de licencias). [20]

Se utiliza en campos como CAD, realidad virtual, representación científica y de información, simulación de vuelo o desarrollo de videojuegos, en el que su principal competidor es Direct3D de Microsoft Windows. [20]

En la actualidad OpenGL tiene dos propósitos principales: el primero, ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, presentando al programador una API única y uniforme; y la segunda, ocultar las diferentes

capacidades de las diversas plataformas hardware, requiriendo que todas las implementaciones soporten el conjunto completo de características OpenGL.[21]

OpenGL es una API basada en procedimientos de bajo nivel que requiere que el programador dicte los pasos exactos necesarios para renderizar una escena. Esto contrasta con las APIs descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para renderizarla. El diseño de bajo nivel de OpenGL requiere que los programadores conozcan en profundidad el pipeline gráfico, a cambio de la libertad ofrecida en la implementación de algoritmos novedosos de renderizado. [22]

DirectX es una colección de APIs creadas y recreadas para facilitar las complejas tareas relacionadas con multimedia, especialmente programación de juegos y vídeo en la plataforma Microsoft Windows. El kit de desarrollo de DirectX es distribuido gratuitamente por Microsoft. [11]

- DirectX incluye las siguientes APIs:
- DirectX incluye las siguientes APIs:
- DirectDraw: para dibujado de imágenes en dos dimensiones (planas).
- Direct3D (D3D)]]: para representación de imágenes en tres dimensiones.
- DirectInput: utilizado para procesar datos del teclado, ratón, joystick y otros controles para juegos.
- DirectPlay: para comunicaciones en red.
- DirectSound: para la reproducción y grabación de sonidos de ondas.
- DirectMusic: para la reproducción de pistas musicales compuestas con DirectMusic Producer.
- DirectShow: para reproducir audio y video con transparencia de red.
- DirectSetup: para la instalación de componentes DirectX.

A pesar de ser desarrollado exclusivamente para la plataforma Windows, una implementación de su API se encuentra en progreso para sistemas Unix (en particular Linux) y X Window System conocida como WineX, desarrollada por la empresa de software Transgaming y orientada a la ejecución de juegos desarrollados para Windows bajo sistemas Unix. [11]

1.11.3 Lenguajes de Shader

Los lenguajes para escribir shaders están muy relacionados con la API gráfica en uso y los más comunes son:

- **HLSL** usado con DirectX y propiedad de Microsoft.

- **GLSL** usado con OpenGL y libre.
- **CG** propiedad de la empresa Nvidia.

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. [14]

Este utiliza muchas funciones y estructuras de control similares al lenguaje de programación C y apoya lazos y bifurcación, incluyendo if, else, if/else, for, do-while, break, continue, etc. [14]

HLSL es el acrónimo de High Level Shader Language o High Level Shading Language. Es un lenguaje de alto nivel desarrollado por Microsoft para ser usado con Microsoft Direct3D API. Es análogo al lenguaje GLSL usado por el estándar OpenGL y al CG desarrollado para tarjetas gráficas NVIDIA Cg. [13]

Conclusiones

La texturización es una forma de incrementar drásticamente el detalle y el realismo de las imágenes sintéticas sin necesidad de utilizar una representación muy complicada de los objetos.

El mapeo de texturas es probablemente el avance más significativo en los gráficos por ordenador de los últimos 10 años.

El presente capítulo hemos partido de las bases del mapeo de texturas para acabar finalizando con técnicas avanzadas que permiten plasmar efectos visuales en la vida real.

Cuánto más avance las técnicas de mapeo de texturas se obtendrá un mayor realismo en las aplicaciones tales como juegos, aplicaciones médicas etc...

Capítulo 2: Descripción de las Soluciones Propuestas

Introducción

En este capítulo se proponen, de las técnicas mencionadas en el capítulo 1, las más apropiadas para dar solución al problema científico planteado, así como variantes de las mismas y los algoritmos a utilizar para implementarlas.

También se dan a conocer el o los tipos de texturas a utilizar, los tipos de mapeo de textura, funciones de texturización, tipos de filtros que se aplicarán a las texturas, lenguajes de programación, librerías gráficas, herramientas a utilizar, etc.

2.1 Tipos de texturas a utilizar

Se utilizarán texturas bidimensionales (2D) que son las más usadas en los gráficos por computadora. Estas permiten usando las técnicas adecuadas alcanzar un nivel elevado de realismo sin un costo computacional alto a diferencia de las texturas 3D que consumen mucho más memoria.

La resolución de las mismas debe ser potencia de 2, de modo que permitan aplicar técnicas de filtrado de manera satisfactorio. Las imágenes deben tener formato BMP o TGA sin compactar. Este último va a permitir utilizar texturas con transparencia. Ambos formatos de textura son los utilizados actualmente en la “Scene Toolkit”.

También serán usadas las texturas cúbicas llamadas **Cubemap** que ocupan mayor espacio de memoria al ser la combinación de 6 texturas 2D pero que serán de gran utilidad.

2.2 Tipos de Filtrado de Textura

Serán utilizados los tipos NEAREST y LINEAR, además según el número coordenadas que se tengan en cuenta para realizar la interpolación, los filtrados **bilinear** y **trilinear**. En el caso del trilinear se usará solo como filtro de minimización (*cuando los texels sean más pequeños que los pixels*) conjuntamente con la técnica de Mipmapping que es la que provee la tercera coordenada donde se interpolan los niveles de mipmap más cercanos a la resolución con la que se esté visualizando el objeto.

Se debe tener en cuenta que el filtrado trilinear es mucho más costoso computacionalmente que bilinear sin embargo provee los mejores resultados por lo que la selección debe ser de acuerdo al nivel de realismo que se necesite. Además, por lo cual se le dará la posibilidad al usuario de seleccionar la variante más apropiada según su conveniencia.

La utilización de la técnica de Mipmapping permitirá que a la hora de aplicar los filtros NEAREST o LINEAR la diferencia de los pixels de la textura y los de la pantalla no sea demasiado grande, pues de ocurrir esto los resultados del filtrado no serán satisfactorios.

Además del Mipmapping automático que provee la herramienta STK actualmente, este proyecto propone implementar una variante manual mediante la cual el usuario pueda entrar la textura que desee para cada nivel de mipmap.

2.3 Tipos de Mapeo de texturas

Será usado el mapeo en 2 partes que será de gran utilidad para texturizar cualquier objeto, no solamente los que sean planos. De sus variantes se usarán el mapeo plano sobre todo para fijar texturas de materiales o de color base a los objetos.

También el mapeo esférico va a ser ideal para objetos esféricos como valones pero que puede aplicarse sobre cualquier objeto y tiene aplicación en la obtención de reflejos básicos y otros efectos básicos. Será aplicado además el mapeo cúbico que continúa proyectando la imagen de forma plana pero aplica la textura desde los 3 ejes de coordenadas y resulta de gran utilidad para lograr reflejos precisos del entorno, entre otras aplicaciones.

Tanto el mapeado esférico como el cúbico serán necesarios para crear objetos reflectantes a través de la técnica de Mapeo de Entorno.

2.4 Efectos de Textura

Los efectos de textura que se proponen son:

La generación de reflejos de entorno y la obtención relieve en la superficie de los objetos sin aumentar su complejidad geométrica, ambos mediante técnicas avanzadas con el uso de texturas.

También se implementará el mipmapping de modo que el usuario pueda usar imágenes predeterminadas para cada nivel de mipmap y estas se intercambien de acuerdo a la resolución actual con la que se esté visualizando el objeto o determinada porción de este.

Los reflejos podrán tener mayor o menor realismo de acuerdo a la técnica de mapeo de entorno usada para generarlos, la cual podrá ser escogida de acuerdo a los requerimientos de la aplicación que se esté desarrollando y al hardware que se posea. Estos podrán ser estáticos (usados sobre todo en objetos inmóviles y entornos que no cambien) o podrán ser dinámicos, los cuales son mucho más costosos computacionalmente pero mucho más reales. Se dará la posibilidad además de regular la opacidad de los reflejos, así como la intensidad con que este va a reflejar la luz.

El relieve de superficie será implementado aprovechando las potencialidades de las tarjetas gráficas modernas y liberando de esta forma al CPU de cálculos tan costosos.

2.5 Técnicas Avanzadas.

Para la obtención de reflejos será aplicada la técnica la ya mencionada técnica de "Environment Mapping" o "Mapeo de Entorno". Serán usadas las variantes: Mapeo Esférico del Entorno (SEM) y Mapeo Cúbico del Entorno (CEM). El SEM se propone como la variante más simple ya que tiene muchas desventajas (ver epígrafe 1.8.1), pero en algunas condiciones donde no se requiera mucha precisión puede ser útil.

Por otra parte, como variante avanzada será desarrollado el CEM, que soluciona las limitaciones del SEM y es -de todas las variantes del mapeado de entorno- la que proporciona los mejores resultados al no depender del punto de visión, eliminarse los efectos de deformación y obtener una perspectiva completa del entorno.

Además va a permitir obtener reflejos dinámicos al poder actualizar las texturas de entorno en tiempo de ejecución a través del render de la escena. Cómo única desventaja tiene que es más costoso que el SEM, pero si se quiere que los entornos sean realistas es necesario contar con un hardware que lo soporte.

Para el caso de las superficies rugosas de propone la técnica "Bump Mapping" la cual no perturba la geometría del objeto y permite incrementar el realismo de determinadas superficies que por naturaleza presentan protuberancias.

2.5.1 Algoritmos de Mapeo

Se utilizarán algoritmos correspondientes a las técnicas seleccionadas. Para implementar el Spherical Environment Mapping se usará el algoritmo de mapeo de

coordenadas esféricas y para el cúbico el correspondiente mapeo de coordenadas cúbicas.

Para el caso del bump mapping se usará un algoritmo para hallar el color final de cada pixel, este algoritmo debe incluir la transformación de las normales usando una textura de normales, así como la ecuación general de iluminación (ver epígrafe 1.9.2) basada en las normales transformadas y en la posición de la luz y el punto de visión.

2.6 Lenguajes de Programación y Herramientas

Para la implementación de las técnicas se utilizará el paradigma de programación Orientado a Objetos y el lenguaje de programación de alto nivel C++, con el cual está implementada la herramienta "Scene Toolkit" y que tiene grandes potencialidades para la programación gráfica.

Como librería gráfica se usará OpenGL la cual es libre. Para aprovechar las potencialidades del hardware gráfico moderno disponible y liberar de cálculos complejos al CPU, se implementarán variantes de shader para las técnicas mencionadas con anterioridad, en el caso del Bump Mapping será solo con Shader.

Como lenguaje de shader será utilizado el GLSL siglas de "*OpenGL Shading Language"*, el cual es parte del API estándar OpenGL.

La utilización de shader permitirá aprovechar las potencialidades del hardware gráfico moderno al permitir el acceso a sus características programables, lo que posibilita liberar de carga al CPU y lograr efectos más realistas.

Como herramientas de Desarrollo se utilizará Microsoft Visual Studio 2003. Como herramienta case, el Rational Rose 2003 para la construcción de modelos y diagramas, y el Render Monkey para probar los Shader.

Conclusiones.

En este capítulo han quedado plasmadas las soluciones técnicas para resolver el problema planteado en esta tesis. Se define qué algoritmos implementar, las técnicas avanzadas de mapeo de texturas a utilizar, y los lenguajes de programación que utilizaremos para lograr los objetivos propuestos.

De esta forma quedan sentadas las bases técnicas para implementar los algoritmos y diseñar un conjunto de clases para la aplicación de efectos de texturas en Entornos Virtuales generados con la Scene Toolkit.

Capítulo 3 Construcción de la Solución Propuesta.

Introducción

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

3.1 Reglas del Negocio.

Las Reglas del Negocio propuestas son:

1. Las texturas cargadas deben ser de formato BMP (24 bit) o TGA (32 y 24 bit) sin compactar.
2. Las resoluciones de las texturas cargadas deben ser potencia de 2.
3. Los ficheros de Shader cargados deberán tener extensión .VERT o .FRAG, de no existir el fichero especificado el programa no detiene su ejecución sin embargo no ejecuta el Shader.
4. Para aplicar el Mipmapping manual, es necesario que todas las imágenes que representan los niveles a partir del uno, posean el mismo nombre que la original (la de nivel 0), seguido del número del nivel que definen.
5. Las texturas usadas para el Mapeo Cúbico del entorno, es decir cada cara del cubemap, debe tener una resolución máxima de 256x256 para lograr un mejor rendimiento.

3.2 Modelo del Dominio

Para el modelamiento del sistema, se seleccionó lo que se conoce como Modelo del Dominio, que viene a ser un modelo conceptual del sistema a desarrollar como se muestra en la siguiente figura:

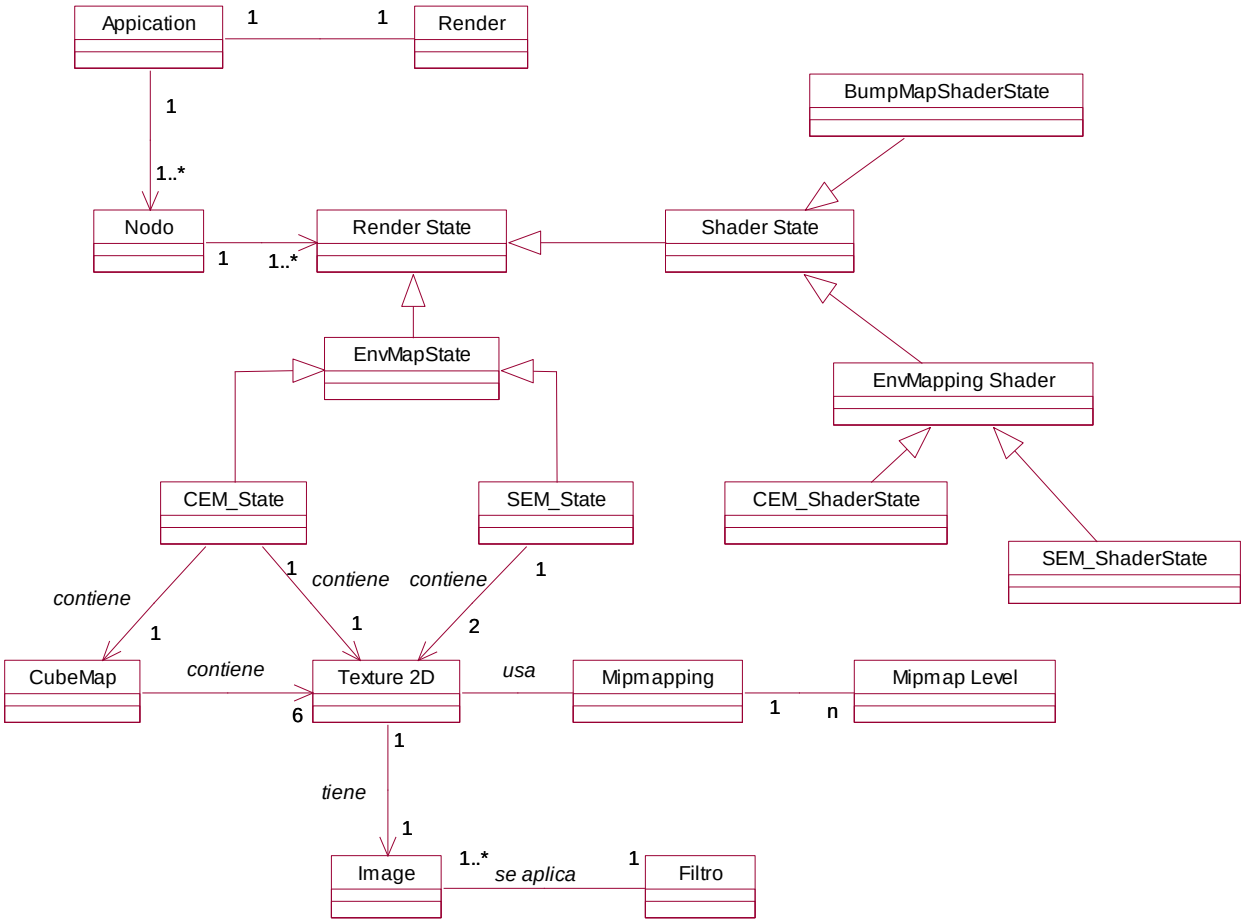


Figura 24: Modelo del Dominio.

3.2.1 Glosario de términos del dominio.

EnvMapState: Estado Mapeo de Entorno.

SEM_State: Estado Mapeo Esférico del Entorno.

CEM_State: Estado del Mapeo Cúbico del Entorno.

CubeMap: Textura que a su vez está constituida por seis texturas 2D.

Mipmapping: Técnica de mapeo que se le aplica a la Textura, cambio del nivel de mipmap de la textura aplicada sobre el objeto teniendo en cuenta la distancia de

este objeto con respecto a la cámara, usando imágenes predeterminadas para cada uno de estos niveles de mipmap.

Mipmap Level: depende de la distancia a la que se encuentra un objeto con respecto a la cámara.

BumpMapShaderState: Estado Shader del Bumpmapping

EnvMappingShader: Estado Shader del Mapeo de Entorno.

SEM_ShaderState: Estado Shader del Mapeo Esférico del Entorno.

CEM_ShaderState: Estado Shader del Mapeo Cúbico del Entorno.

Texture 2D: imagen que sirve de "piel" a los modelos en un mundo virtual.

Image: Imagen.

Los demás términos que se encuentran en el modelo del dominio, y que no son descritos aquí, son términos de la STK.

3.3 Captura de Requisitos.

En este epígrafe se define qué es lo que el sistema debe hacer, para lo cual se identifican las funcionalidades requeridas y las restricciones que se imponen, que esto no es más que los requerimientos del sistema, tanto los funcionales como los no funcionales.

3.3.1 Requisitos Funcionales.

1. Cargar Imagen con formato TGA desde fichero.
2. Cargar Imagen con Formato BMP desde fichero.
3. Crear una imagen.
4. Crear una textura 2D.
5. Crear el Estado SEM.
6. Modificar Estado SEM.
7. Crear el estado CEM.
8. Modificar estado CEM.
9. Gestionar estado Environment Mapping.
   - 9.1 Habilitar estado.
   - 9.2 Deshabilitar estado.
10. Eliminar el estado.

    Cargar el código del Vertex Shader y el Fragment Shader especificando la ruta de los ficheros.
11. Crear variables de Shader.

12. Actualizar el valor de las variables de Shader.
13. Configurar Estado de Shader Environment Mapping.
    - 13.1 Crear Estado.
    - 13.2 Modificar estado.
14. Configurar Estado de Shader Bump Mapping.
    - 14.1 Crear Estado.
    - 14.2 Modificar estado.
15. Gestionar Estado de Shader.
    - 15.1 Habilitar estado.
    - 15.2 Deshabilitar estado.
    - 15.3 Eliminar estado.
16. Gestionar Mipmapping
    - 16.1 Habilitar mipmapping.
      - 16.1.1- Establecer modo y tipo de mipmapping.
    - 16.2 Modificar tipo y modo de mipmapping.
    - 16.3 Deshabilitar mipmapping.
    - 16.4 Crear mipmapping manual.
      - 16.4.1 Calcular la cantidad de mipmaps necesarios dado el tamaño de la textura.
      - 16.4.2 Cambiar cantidad de mipmaps a utilizar por la textura.
      - 16.4.3 Generar rutas de los mipmaps dada la ruta del fichero original y la cantidad de mipmaps.
      - 16.4.4 Crear mipmaps.
    - 16.5 Activar Filtro de magnificación NEAREST.
    - 16.6 Activar Filtro de magnificación LINEAR.
    - 16.7 Desactivar filtro de magnificación.
17. Establecer niveles de multitextura con OpenGL.
18. Aplicar opacidad a texturas 2D y cubemap con OpenGl.
19. Activar modo de aplicación de la textura con la librería gráfica.
20. Generar coordenadas de textura automáticamente con OpenGL usando planos.
21. Generar automáticamente coordenadas esféricas de textura con OpenGL.
22. Generar automáticamente coordenadas cúbicas de textura con OpenGL.
23. Calcular el vector reflejado para cada vértice de un objeto.
24. Generar reflejos del entorno utilizando mapeo esférico.
    - 24.1 Calcular coordenadas esféricas de textura para cada vértice del objeto.

24.2 Calcular coordenadas esféricas de textura para cada pixel del objeto.

24.3 Fijar la textura al objeto.

24.4 Cambiar las coordenadas de la textura en tiempo de ejecución.

25. Generar reflejos del entorno utilizando mapeo cúbico

25.1 Crear un cubemap a partir de 6 texturas 2D.

25.2 Calcular la inversa de una matriz.

25.3 Calcular las coordenadas cúbicas de textura para cada vértice del objeto.

25.4 Calcular las coordenadas cúbicas de textura para cada pixel del objeto.

25.5 Fijar el cubemap sobre el objeto.

25.6 Cambiar las coordenadas del cubemap en tiempo de ejecución.

26. Generar Bumpmapping

26.1 Calcular el vector de Posición de la Cámara.

26.2 Calcular el vector de Posición de la luz.

26.3 Transformar los vectores de Posición de la cámara y de posición de la luz.

26.4 Calcular la normal según el color de la textura bumpmapping en la coordenada 0.

26.5 Calcular el color final del pixel.

27. Aplicar mipmapping a una textura.

27.1 Establecer niveles de mipmaps usando la librería grafica.

27.2 Establecer modo de mipmapping a aplicar usando la librería gráfica.

27.3 Escoger nivel de mipmap apropiado según la distancia a la que esté el objeto de cámara.

27.4 Cambiar mipmap en tiempo de ejecución según la distancia entre el objeto y la cámara.

28. Aplicar filtro bilinear a una textura.

28.1 Calcular el color final de los texel de una textura interpolando los colores de los texel adyacentes al mismo.

29. Aplicar filtro trilinear a una textura.

29.1 Calcular el color final de un texel interpolando los colores de 2 niveles de mipmap.

30. Crear Textura en OpenGL.

31. Crear Shader usando la librería gráfica.

32. Compilar un Shader.

33. Adicionar variables al Shader.

3.3.2 Requisitos no Funcionales.

- **Software:** Sistema operativo Windows y Linux.

- **Hardware:** Se debe contar con un hardware que soporte OpenGl 1.4 para versiones sin shader y para el caso de los shader tarjeta de video de 128 megabyte de RAM como mínimo que pixel shader 1.0 y vertex shader 1.0.

- **Restricciones de Diseño e implementación**:

Debe usarse el paradigma de programación Orientado a objetos. Se usará el mismo estándar de codificación que la STK de SIMPRO en su primera versión

Las clases a desarrollar deben ser acoplables a la STK, por lo que su arquitectura está restringida por la arquitectura de esta herramienta.

El conjunto de clases debe programarse en el lenguaje C++, se debe utilizar de forma transparente la librería gráfica OpenGl y GLSL como lenguaje de shader para aprovechar las potencialidades del GPU moderno.

Como herramientas de desarrollo de debe usar CodeBlock para el caso de que se trabaje en Linux y Microsoft Visual Studio 2003 para Windows.

- **Usabilidad:** Los futuros usuarios del sistema serán programadores con conocimientos básicos de programación gráfica y de la terminología afín. El producto debe estar concebido para que el usuario piense qué desea hacer y no cómo hacerlo, por lo que éste requerimiento debe estar presente en alto grado en el producto final.

- **Rendimiento:** Como aplicación en tiempo real debe tener alta velocidad de procesamiento de los cálculos, tiempo de respuesta y recuperación del sistema.

- **Soporte:**

Portabilidad: En una versión inicial deberá correr sobre la plataforma Windows, pero debe estar preparado para que con rápidas modificaciones pueda migrar para Linux.

Debe ser fácil de mantener, extensible y adaptable de forma que pueda trabajar no solo OpenGl sino con otras librerías gráficas.

3.4 Modelo de Casos de Usos del Sistema.

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

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

3.4.1- Actor del sistema

Actores	**Justificación**
Programa Desarrollado con STK	Es quien se beneficia con las funcionalidades que brinda el conjunto de clases para la aplicación de efectos de texturas en entornos virtuales.

Tabla 1: Actor Del Sistema.

3.4.2- Casos de Usos del Sistema

Id CUS	**Casos de Usos del Sistema**
CUS1	Configurar Estado Mapeo del Entorno (EMapping)
CUS2	Configurar Estado Mapeo Cúbico del Entorno (CEM)
CUS3	Configurar Estado Mapeo Esférico del Entorno (SEM)
CUS4	Gestionar Estado Mapeo del Entorno (EMapping)
CUS5	Gestionar Mipmapping.
CUS6	Configurar Estado de Shader Mapeo de Entorno (**EMapShaderState)**
CUS7	Configurar Estado de Shader Bump mapping **(BumpmapShaderState).**
CUS8	Gestionar Estado de Shader
CUS9	Aplicar Mipmapping.
CUS10	Aplicar Mapeo de Entorno (EMapping)
CUS11	Ejecutar Estado de Shader
CUS12	Ejecutar Estado de Shader Mapeo Cúbico del Entorno (CEM).
CUS13	Ejecutar Estado de Shader Mapeo Esférico del Entorno (SEM).

CUS14	Ejecutar Estado de Shader de Bump Mapping.
CUS15	Generar Textura de OpenGL. (CUS de la Herramienta STK)
CUS16	Hacer Ciclo de Escena (CUS de la Herramienta STK)

Tabla 2: Identificador por cada Caso Uso del Sistema.

3.4.3- Diagramas de CUS

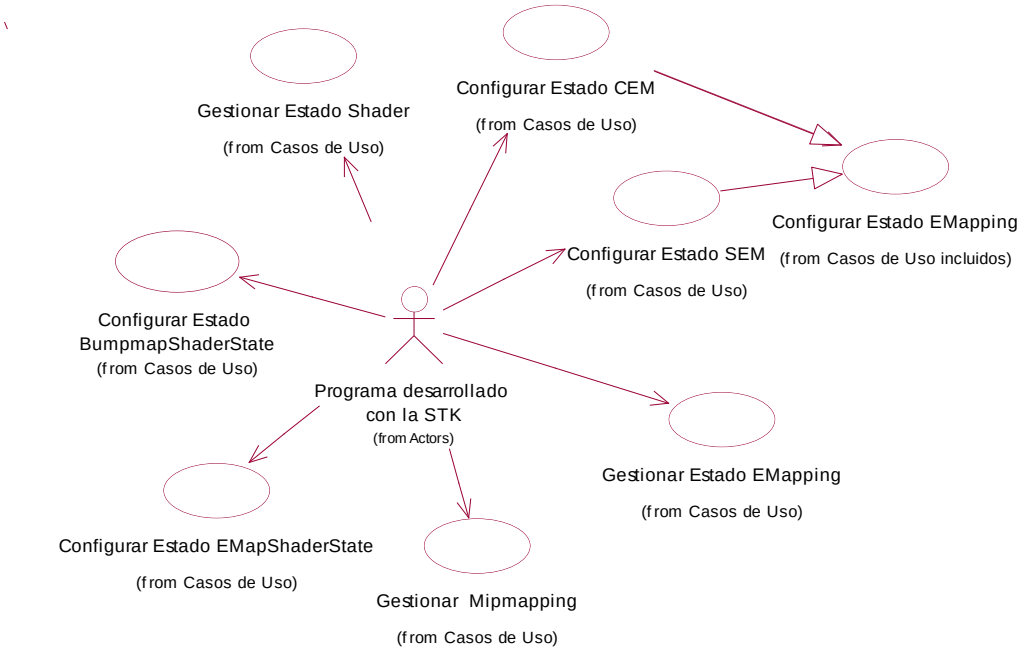


Figura 25: Diagrama de CUS de Gestión.

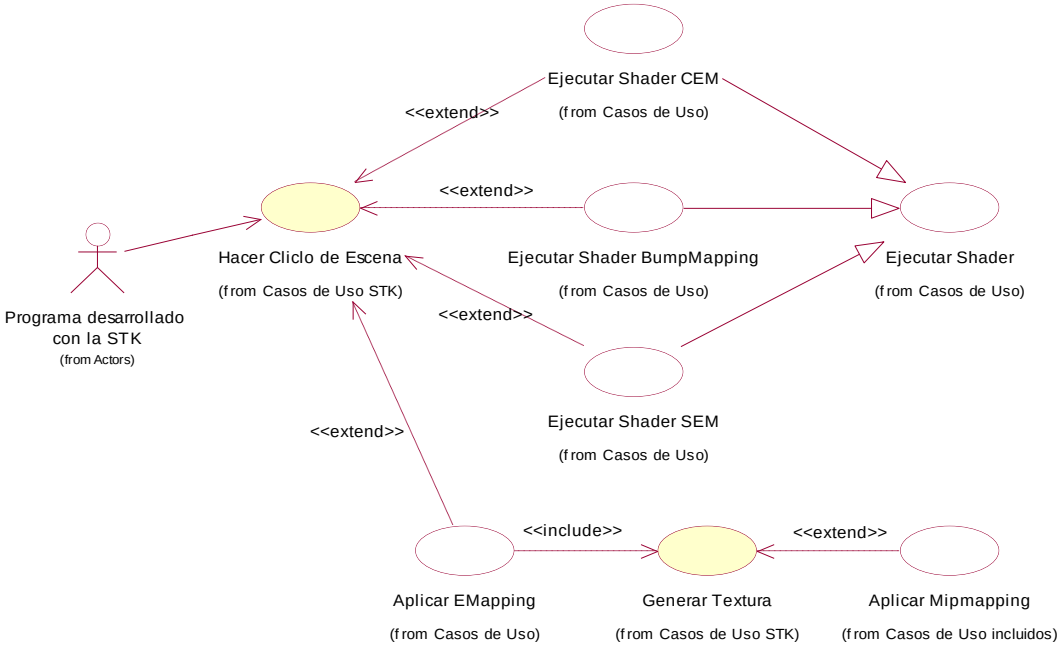


Figura 26: Diagrama de de CUS de Ejecución.

3.4.4- Especificación de los CUS en formato expandido

Caso de uso:	Configurar Estado Mapeo de Entorno (EMapping)
Actor (es):	
Propósito:	Crear y/o modificar estado de **EMapping**.
Resumen:	El CUS comienza cuando el actor crea y/o modifica el estado de EMapping, y culmina con la ejecución de una de estas acciones.
Referencias:	RF 1, 2, 3, 4
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
Sección: Crear Estado.	
1- Solicita crear un Estado EMapping especificando la ruta de la imagen de la textura base, el factor de opacidad (lodBias) y si va a usar shader o no.	1.1- Crea el estado de EMapping. 1.1.1- Crea una nueva textura en el estado EMapping como textura base. 1.1.2- Crea una imagen para la textura. 1.1.3- Carga el contenido de la imagen desde fichero con la ruta especificada. 1.1.4- Le asigna el valor entrado al factor de opacidad. 1.1.5- Le asigna el valor al modo shader.
Sección: Modificar Estado.	
1- Solicita modificar la textura base del estado especificando la ruta de la imagen.	1.1- Crea una textura. 1.2- Crea una imagen. 1.3- Carga el contenido de la imagen desde fichero con la ruta especificada. 1.4- Le asigna la textura creada a la textura base del estado.
2- Específica el factor de opacidad del estado.	2.1- Le asigna el factor de opacidad al estado
3- Especifica si va usar Shader o no.	3.1- Le asigna el valor al estado.

Curso Alternativo de los Eventos	
Sección: Crear Estado.	
	1.1.3- Si la ruta especificada es incorrecta no carga la imagen y lanza una excepción.
Sección: Modificar Estado.	
	1.4- Si la ruta especificada es incorrecta no carga la imagen y lanza una excepción.
Post-condiciones:	
Prioridad:	Crítico.

Tabla 3: Descripción del CUS "Configurar Estado de Mapeo de Entorno (EMapping)".

Caso de uso:	Configurar Estado Mapeo Cúbico del Entorno (CEM)
Actor (es):	Programa desarrollado con STK
Propósito:	Crear y/o modificar el estado de CEM.
Resumen:	El caso de uso comienza cuando el actor solicita crear y/o modificar el estado CEM y en dependencia de esto el sistema crea o modifica el estado con lo cual finaliza el caso de uso.
Referencias:	RF 7, 8, Hereda del CUS1
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. Solicita crear o modificar el estado.	1.1 Si escoge crear ir a sección **Crear Estado CEM.** 1.2 Si escoge modificar ir a sección **Modificar Estado CEM.**
Sección: Crear Estado CEM.	
1- Solicita crear el estado CEM pasándole la ruta de la imagen de la textura base, el factor de opacidad y si va a usar shader o no**,** si va a ser estático o no y *tamaño de las texturas a generar*.	1.1- Crea un Estado de Environment Mapping. **Ir a CUS** Configurar estado EMapping. 1.2- Crea el estado de CEM. 1.2.1- Crea el cubemap al estado CEM.

	1.2.2- Le asigna el modo estático o dinámico. 1.2.2- *Le asigna el tamaño de las texturas a generar*.
Sección: Modificar Estado CEM.	
1- Solicita cambiar la textura base, el factor de reflexión, el modo de shader o la posición del objeto.	1.1- **Ir a CU** *Gestionar Estado EMapping.*
2- Solicita crear un nuevo cubemap del estado CEM especificando para cada cara la ruta de la imagen.	2.1- Crea el Cubemap Para cada cara del cubemap: 2.2- Crea una nueva textura. 2.3- Crea una imagen para esa textura. 2.4- Carga el contenido de la imagen desde fichero con la ruta especificada.
3- Específica si va a ser estático o dinámico.	3.1- Le asigna el valor al estado.
4- Entra el tamaño de las texturas a generar en caso de que sea dinámico.	4.1- Le asigna el tamaño de textura especificado.
Curso Alternativo de los Eventos	
Post-condiciones:	
Prioridad:	Crítico.

Tabla 4: Descripción del CUS "Configurar Estado de Mapeo Cúbico del Entorno (CEM)".

Caso de uso:	Configurar Estado de Mapeo Esférico del Entorno (SEM)
Actor (es):	Programa desarrollado con STK
Propósito:	Crear y/o modificar el estado de **SEM**
Resumen:	El caso de uso comienza cuando el actor utiliza el sistema para crear y/o modificar el estado de Mapeo **Esférico del Entorno**, y finaliza con la ejecución de esas acciones.
Referencias:	RF 5, 6, Hereda del CUS1
Pre-condiciones:	

Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. Solicita crear o modificar el estado.	1.1 Si escoge crear ir a sección **Crear Estado SEM.** 1.2 Si escoge ir a sección **Modificar Estado SEM.**
Sección: Crear Estado SEM	
1- Solicita crear el estado SEM pasándole la ruta de la imagen de la textura base, el factor de opacidad, si va a usar shader o no y la ruta de la imagen de la textura de entorno**.**	1.1- Crea un Estado de Environment Mapping. **Ir a CU** Configurar *Estado EMapping.* 1.2- Crea el estado de SEM. 1.2.1- Crea la textura de entorno del estado. 1.2.2- Crea una imagen. 1.2.3- Carga el contenido de la imagen desde fichero con la ruta especificada. 1.2.4- Le asigna la imagen a la textura de entorno.
Sección: Modificar Estado SEM	
1- Solicita cambiar la textura base, el factor de opacidad o el modo de shader.	1.1- **Ir a CU** *Gestionar Estado EMapping*
2- Solicita crear la textura de entorno del estado SEM especificando la ruta de la imagen.	2.1- Crea una textura. 2.2- Crea una imagen. 2.3- Carga el contenido de la imagen desde fichero con la ruta especificada. 2.4- Le asigna la textura creada a la textura de entorno del estado.
Curso Alternativo de los Eventos	
Post-condiciones:	
Prioridad:	Crítico.

Tabla 5: Descripción del CUS "Configurar Estado de Mapeo Esférico del Entorno (SEM))".

Caso de uso:	Gestionar Estado Mapeo del Entorno (EMapping)
Actor (es):	Programa desarrollado con STK
Propósito:	Habilitar o deshabilitar y eliminar el estado de **SEM.**
Resumen:	El caso de uso comienza cuando el actor solicita habilitar o deshabilitar el estado de Mapeo del Entorno, y finaliza con la ejecución de esas acciones.
Referencias:	RF 9.1, 9.2, 9.3
Pre-condiciones:	

Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. Solicita habilitar o deshabilitar el estado.	1.1 Si escoge habilitar ir a sección **Habilitar Estado.** 1.2 Si escoge deshabilitar ir a sección **Deshabilitar Estado.**
Sección: Habilitar Estado.	
1- Solicita habilitar el estado (puede pasar como parámetro un: CEM o SEM).	1.1- Verifica si se pasó como parámetro el estado. 1.1.1- Si se pasó crea un nuevo estado asignándole a este el pasado como parámetro. 1.1.2- Habilita el estado.
Sección: Deshabilitar Estado.	
1- Solicita deshabilitar el estado CEM y especifica si va a eliminar el estado.	1.1- Deshabilita el estado. 1.2- Si desea eliminar el estado lo destruye borrándolo de la memoria.
Curso Alternativo de los Eventos	
Sección: Habilitar Estado.	
	1.1.1- Si no se pasó como parámetro verifica que ya esté creado el estado. 1.1.2- Si existe lo habilita.
Sección: Deshabilitar Estado.	
	1.2- Si no desea borrarlo, mantiene el estado cargado en memoria.

Post-condiciones:	
Prioridad:	

Tabla 6: Descripción del CUS "Gestionar Estado Mapeo del Entorno (EMapping)".

Caso de uso:	Gestionar Mipmapping.
Actor (es):	Programa desarrollado con STK
Propósito:	Activar y desactivar el estado de mipmapping, así como los filtrados bilinear y trilinear.
Resumen:	El actor habilita o deshabilita el estado de mipmapping de una textura especificando el tipo. El actor también puede escoger si el mipmapping será automático o manual, y concluye cuando habilitamos o deshabilitamos el mipmapping.
Referencias:	RF 16.1, 16.2, 16.3, 16.4, 16.5, 16.6, 16.7
Pre-condiciones:	1. Tiene que estar creada la textura. 2. Para el mipmapping manual los ficheros de los mipmaps tienen el mismo nombre de la textura original pero se le agrega el número del nivel al final.
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
Sección: Habilitar	
1- Solicita activar el mipmapping a una textura.	
2- Especifica el modo de mipmapping a activar y especifica el tipo del mipmapping (manual o automático)	2.1- Activa el modo de mipmapping especificado en el estado. Para especificar el tipo de mipmapping: 2.2- Le asigna al mipmapping de la textura el tipo especificado (por defecto este es automático). 2.3- Verifica el tipo entrado. 2.4- Si es manual comprueba que haya sido cargada la imagen de la textura. 2.5- Si ya fue cargada crea un estado de Mipmapping manual pasándole la ruta de la imagen, el ancho y el alto. 2.6- Determina la cantidad de mipmaps a generar según el tamaño de la imagen de la textura 2.7- Con la ruta de la imagen de la textura y la cantidad de mipmaps que se necesitan determina las rutas para los demás

	mipmaps. 2.8- Se crean tantas imágenes como mipmaps se requieran. 2.9- Se cargan las imágenes con las rutas de los mipmaps.
Sección: Deshabilitar	
1- Solicita desactivar el mipmapping de la textura.	1.1- Deshabilita el mipmapping.
Curso Alternativo de los Eventos	
Sección: Habilitar	
	2.5- Si no se ha cargado 2.5.1- Lanza una excepción de que no se puede crear el Mipmapping pues no ha sido creada la imagen original de la textura. 2.5.2 Se activa el tipo automático.
Post-condiciones:	
Prioridad:	Secundario.

Tabla 7: Descripción del CUS "Gestionar Estado de Mipmapping".

Caso de uso:	Configurar Estado de Shader Mapeo de Entorno (**EMapShaderState)**
Actor (es):	Programa desarrollado con STK
Propósito:	Crear y/o Modificar el estado de shader **EMapShaderState.**
Resumen:	El caso de uso comienza cuando el actor solicita crear y/o modificar el estado **EMapShaderState,** y culmina cuando alguna de estas funciones es ejecutada.
Referencias:	RF 10, 11, 12, 13.1, 13.2
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. Solicita crear y/o modificar el estado.	1.1 Si escoge crear ir a sección **Crear Estado.** 1.2 Si escoge Modificar ir a sección **Modificar Estado.**

Sección: Crear Estado	
1- Solicita crear el estado **EMapShaderState** pasándole el tipo de environment mapping.	1.1- Crea el estado. 1.1.1- Crea el Shader del estado pasándole una dirección del Vertex Shader y el Fragment Shader según el tipo especificado. 1.1.1.1- Carga el código del Vertex Shader y el Fragment Shader. 1.1.2- Crea una variable de tipo float para el factor de reflexión. 1.1.3- La adiciona a la lista de variables del estado. 1.1.4- Crea una variable de tipo float para el factor de opacidad. 1.1.5- La adiciona a la lista de variables del estado. 1.1.6- Crea una variable de shader de tipo Sampler2D para la textura base. 1.1.7- La adiciona a la lista de variables del estado. 1.1.8- Crea otra variable de Shader de tipo Sampler2D. 1.1.9- La adiciona a la lista de variables del estado. 1.1.10- Crea una variable de tipo Vector3 para la posición de la luz. 1.1.11- La adiciona a las lista de variables. 1.1.12- Crea una variable de tipo Vector3 para la posición de la cámara. 1.1.13- La adiciona a las lista de variables.
Sección: Modificar Estado	
2- Entra el valor del factor de reflexión	2.1- Le asigna el valor a la variable de shader correspondiente.
3- Entra el valor del factor de opacidad.	3.1- Le asigna el valor a la variable de shader correspondiente.
4- Entra el valor de la posición de la luz.	4.1- Le asigna el valor a la variable de shader correspondiente.
5- Entra el valor de la posición de la cámara.	5.1- Le asigna el valor a la variable de shader correspondiente.

Curso Alternativo de los Eventos	
Post-condiciones:	
Prioridad:	Secundario.

Tabla 8: Descripción del CUS "Configurar Estado de Shader Mapeo de Entorno (EMapShaderState)".

Caso de uso:	Configurar Estado de Shader Bump Mapping **(BumpmapShaderState)**
Actor (es):	Programa desarrollado con STK
Propósito:	Crear y/o Modificar el estado de shader **Bump Mapping.**
Resumen:	El caso de uso comienza cuando el actor utiliza el sistema para crear y/o modificar, el estado **Bump Mapping** y finaliza con la ejecución de una de esas acciones.
Referencias:	RF 10, 11, 12, 14.1, 14.2
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. Solicita crear y/o modificar el estado.	1.1 Si escoge crear ir a sección **Crear Estado.** 1.2 Si escoge Modificar ir a sección **Modificar Estado.**
Sección: Crear Estado	
1- Solicita crear el estado **Bumbmapping**.	1.1- Crea el estado. 1.1.1- Crea el Shader del estado pasándole una dirección del Vertex Shader y el Fragment Shader. 1.1.1.1- Carga el código del Vertex Shader y el Fragment Shader. 1.1.2- Crea una variable de tipo Vector4 para la posición de la luz ambiental. 1.1.3- La adiciona a la lista de variables del estado. 1.1.4- Crea una variable de tipo Vector4 para la posición de la luz especular.

	1.1.5- La adiciona a la lista de variables del estado. 1.1.6- Crea una variable de tipo Vector4 para la posición de la luz difusa. 1.1.7- La adiciona a la lista de variables del estado. 1.1.8- Crea una variable de tipo float para el poder especular (concentración de brillo en el objeto). 1.1.9- La adiciona a la lista de variables del estado. 1.1.10- Crea una variable de tipo Sampler2D para la Textura Base. 1.1.11 La adiciona a la lista de variables del estado. 1.1.12 Crea una variable de tipo Sampler2D para la textura Bumbmapping. 1.1.13 La adiciona a la lista de variables del estado.
Sección: Modificar Estado	
2- Entra el valor de la posición de la luz ambiental.	2.1- Le asigna el valor a la variable de shader correspondiente.
3- Entra el valor de la posición de la luz especular.	3.1- Le asigna el valor a la variable de shader correspondiente
4- Entra el valor de la posición de la luz difusa.	4.1- Le asigna el valor a la variable de shader correspondiente
5- Entra el valor del poder especular.	5.1- Le asigna el valor a la variable de shader correspondiente
Curso Alternativo de los Eventos	
Post-condiciones:	
Prioridad:	Secundario.

Tabla 9: Descripción del CUS "Configurar Estado de Shader Bump mapping (BumpmapShaderState)".

Caso de uso:	Gestionar Estado de Shader
Actor (es):	Programa desarrollado con la STK

Propósito:	Habilitar o deshabilitar el estado de Shader
Resumen:	El CUS comienza cuando el actor solicita habilitar o deshabilitar el estado de Shader, y culmina con la ejecución de alguna de estas acciones.
Referencias:	RF 15.1, 15.2, 15.3
Pre-condiciones:	

Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
1. Solicita Habilitar o deshabilitar el estado.	1.1 Si escoge crear ir a sección **Habilitar Estado.** 1.2 Si escoge Modificar ir a sección **Deshabilitar Estado.**
Sección: Habilitar Estado	
1- Solicita Habilitar el Estado de Shader.	1.1- Habilita el estado. 1.2- Verifica si se pasó como parámetro. 1.2.1- Si se pasó se le asigna al estado de Shader el pasado por parámetro.
Sección: Deshabilitar Estado	
1- Solicita deshabilitar el estado de shader y especifica si va a eliminarlo.	1.1- Desactiva el estado 1.2- Si especifica eliminarlo destruye el estado quitándolo de la memoria
Curso Alternativo de los Eventos	
Sección: Habilitar Estado	
	1.2.1- Si no se pasó por parámetro se verifica si no está creado el estado de Shader. 1.2.2 Si no está creado se crea uno nuevo.
Sección: Deshabilitar Estado	
	1.2- Si no especifica eliminarlo mantiene el estado cargado en memoria.

Post-condiciones:	
Prioridad:	

Tabla 10: Descripción del CUS "Gestionar Estado de Shader"

Caso de uso:	Aplicar Mipmapping.
Actor (es):	
Propósito:	Aplica o ejecuta un mipmapping que ya esté activado.
Resumen:	El CUS comienza cuando el CUS Gestionar Texturas en OpenGL necesita aplicar la técnica de mipmapping a alguna textura y activa el mismo de una forma manual o automática.
Referencias:	RF 27.1, 27.2, 27.3, 27.4, es <<extend>> del CUS16
Pre-condiciones:	Tiene que estar activado el mipmapping de la textura. La imagen original de la textura tiene que estar cargada.

Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
	1- Verifica si el tipo de Mipmapping activado es manual o automático.
	2- Si es manual 2.2- Obtiene el formato de la imagen, el ancho, alto y la información. 2.2.1 Crea el mipmap de nivel 0 con los datos obtenidos. 2.3- Obtiene el Mipmapping de la textura 2.4 Obtiene la cantidad de mipmaps del Mipmapping. 2.5 Desde 1 hasta la cantidad de mipmaps crea los restantes niveles de mipmap. 2.5.1 Obtiene la imagen del nivel de mipmap correspondiente. 2.5.2 Obtiene el formato de la imagen, el ancho, alto y la información. 2.5.3 Crea el nivel de mipmap correspondiente con los datos obtenidos. 2.6- Obtiene los mipmaps de la textura, crea los niveles de mipmap correspondientes.
Curso Alternativo de los Eventos	

	2- Si es automático 2.1- Obtiene el formato de la imagen original, el ancho, alto y la información. 2.2- Activa la generación automática de mipmaps para la imagen original con los datos externos.
Post-condiciones:	Será aplicado el mipmapping.
Prioridad:	Secundario.

Tabla 11: Descripción del CUS "Aplicar Mipmapping."

Caso de uso:	Aplicar Mapeo de Entornos (EMapping)
Actor (es):	
Propósito:	Habilitar o deshabilitar la generación de reflejos del entorno utilizando la técnica **SEM** o **CEM**.
Resumen:	El sistema utiliza la configuración del estado SEM o CEM creado y activado con anterioridad y habilita la generación de reflejos del entorno con la liberaría gráfica de OpenGL aplicando la técnica SEM o CEM.
Referencias:	RF 17,18,19, 20, 21, 22, 23, 24, 25.2, 25.3, 25.4, 25.5, 25.6, extensión del CUS15
Pre-condiciones:	El estado EM está habilitado.
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
	Recibe como entrada un estado de mapeo de entorno: Verifica el tipo de estado en caso de ser Esférico ir a la Sección 1: **Habilitar Estado de SEM**, en caso de ser cúbico ir a la Sección 2: **Habilitar Estado de CEM**. Sección 3: **Deshabilitar estado mapeo de entorno**.
Sección: Habilitar Estado de SEM	
	1.1- Verifica que se esté usando la textura base. 1.2- Si se está usando obtiene la **textura**. 1.2.1- Establece esta textura como la de nivel 0 de multitextura. 1.2.2 Habilita la textura 2D. 1.2.3 Genera dicha textura en OpenGL especificando como

	target GL_TEXTURE2D. 1.3 Si está desactivado el modo Shader del SEM: Genera automáticamente las coordenadas de la textura usando planos. 1.4- Obtiene la textura de Entorno del SEM. 1.5- Establece esta textura como nivel1 de multitextura. 1.6- Habilita la textura 2D 1.7- Genera dicha textura en OpenGL. 1.8- Si está desactivado el modo Shader del SEM: 1.8.1- Activa la generación automática de coordenadas de textura. 1.8.2- Activa el modo de Generación **Espherical Map.** 1.8.3- Obtiene el factor de opacidad del estado SEM. 1.8.4- Regula la opacidad de la reflexión del objeto.
Sección 2: Habilitar Estado de CEM	
	1.1- Verifica que se esté usando la textura base. 1.2- Si se está usando obtiene la **textura base**. 1.2.1- Establece esta textura como la de nivel 0 de multitextura. 1.2.2 Activa la textura 2D. 1.2.3 Genera dicha textura en OpenGl especificando como **target** GL_TEXTURE2D. 1.3- Si está desactivado el modo Shader del CEM: Activa el Mapeo automático usando planos. 1.4- Obtiene el Cubemap del estado CEM. 1.5- Establece esta textura como nivel1 de multitextura. 1.6- Verifica si ya está creado el cubemap en la memoria de video. 1.6.1- Si no está creado lo manda a crear*. Ir a CUS Generar CubeMap.* 1.7- Habilita la textura Cube_Map

	1.8- Si está desactivado el modo Shader del CEM: 1.8.1- Activa la generación automática de coordenadas de textura. 1.8.2- Activa el modo de Generación **Reflextion Map.** 1.8.3- Obtiene el factor de opacidad del estado CEM. 1.8.4- Regula la opacidad de la reflexión del objeto.
Sección 3: Deshabilitar Estado de Mapeo de Entorno.	
	1.1- Deshabilita la Textura 2D 1.2- Deshabilita la textura del cubemap. 1.3- Deshabilita la generación automática de coordenadas de textura.
Curso Alternativo de los Eventos	
Post-condiciones:	Estado SEM habilitado, estado CEM habilitado o estado EM deshabilitado.
Prioridad:	Crítico.

Tabla 12: Descripción del CUS "Ejecutar Mapeo de Entorno (EMapping)"

Caso de uso:	Ejecutar Estado de Shader
Actor (es):	
Propósito:	Habilitar el estado Shader del Shader pasado por parámetros.
Resumen:	El caso de uso comienza cuando el CUS externo Hacer Ciclo de Escena hace una llamada a este CU y le pide habilitar el estado del Shader pasado por parámetro en caso de este no estar creado se crea, se habilita, se accede a la lista de variables del Estado Shader.
Referencias:	RF 31, 32, 33.
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**

	1.1 Crear el programa de Shader. 1.2 Crea el Vertex Shader: 1.2.1 Le asocia el código al Vertex Shader. 1.2.2 Compila el Vertex Shader. 1.2.3 Asocia el Vertex Shader al programa. 1.3 Crea el Fragment Shader: 1.3.1 Le asocia el código al Fragment Shader. 1.3.2 Compila el Fragment Shader. 1.3.3 Asocia el Fragment Shader al programa. 1.4 Linkea el programa. 1.5 Si se compiló todo: 1.5.1 Accede a la lista de variables del Estado de Shader. 1.5.2 Adicionar las variables al programa de Shader.
Curso Alternativo de los Eventos	
	1.5 Si no se compilo deshabilito el estado Shader.
Post-condiciones:	Estado de Shader habilitado.
Prioridad:	Secundario.

Tabla 13: Descripción del CUS "Ejecutar Estado de Shader"

Caso de uso:	Ejecutar Estado de Shader Mapeo Cúbico del Entorno (CEM).
Actor (es):	
Propósito:	Generar reflejos realistas del entorno usando la técnica de Cube Environment Mapping CEM utilizando Shader.
Resumen:	El sistema calcula las coordenadas de mapeo para cada vértice y fija la textura en el objeto, si la cámara se mueve de lugar las coordenadas son generadas nuevamente. Determina la cara del cubemap y calcula las coordenadas de textura de esa cara que corresponde al vértice. Calcula el color final de cada pixel.
Referencias:	RF 23, 25.2, 25.3, 25.4, 25.5, 25.6, Hereda del CUS 11, extensión del CUS15
Pre-condiciones:	
Curso Normal de los Eventos	

Acción del Actor	**Respuesta del Sistema**
	1- Para Cada Vértice del Objeto: 1.1- Obtiene el vector normal N. 1.2- Obtiene la posición de la cámara. 1.3- A partir de la posición de la cámara y la posición del vértice calcula el vector incidente I. 1.4- Calcula el vector reflejado en dicho vértice, R a través de I y N.
	2- Determina la cara del cubemap y calcula las coordenadas de textura de esa cara que corresponde al vértice.
	3- Para cada Pixel del Objeto: 3.1- Calcula las coordenadas de texturas anteriores para cada pixel del objeto. 3.2- Calcula el color final de cada pixel. 3.2.1- Si se utiliza una textura simple utiliza el color de la textura y un factor de reflexión.
Curso Alternativo de los Eventos	
	3.2.1- Si son utilizadas varias texturas se mezclan los colores de las texturas según el factor de reflexión.
Post-condiciones:	
Prioridad:	Secundario.

Tabla 14: Descripción del CUS "Ejecutar Estado de Shader Mapeo Cúbico del Entorno (CEM)."

Caso de uso:	Ejecutar Estado de Shader Mapeo Esférico del Entorno (SEM).
Actor (es):	
Propósito:	Generar reflejos del entorno usando la técnica del Mapeo Esférico del Entorno.
Resumen:	El sistema calcula las coordenadas esféricas de mapeo para cada vértice y fija la textura en el objeto, si la cámara se mueve de lugar las coordenadas son generadas nuevamente. Determina la cara del cubemap y calcula las coordenadas de textura de esa cara que corresponde al vértice. Calcula el color final para cada pixel.
Referencias:	RF 23, 24.1, 24.2, 24.3, 24.4, Hereda CUS 11, extensión del CUS15
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
	1- Para Cada Vértice del Objeto: 1.1- Obtiene el vector normal N.

	1.2- Obtiene la posición de la cámara. 1.3- A partir de la posición de la cámara y la posición del vértice calcula el vector incidente I. 1.4- Calcula el vector reflejado en dicho vértice, R a través de I y N.
	2- Calcula las coordenadas esféricas de textura correspondientes a cada vértice.
	3- Para cada Pixel del Objeto: 3.1- Calcula las coordenadas de texturas anteriores para cada pixel del objeto. 3.2- Calcula el color final de cada pixel. 3.2.1- Si se utiliza una textura simple utiliza el color de la textura y un factor de reflexión.
Curso Alternativo de los Eventos	
	3.2.1- Si son utilizadas varias texturas se mezclan los colores de las texturas según el factor de reflexión.
Post-condiciones:	
Prioridad:	Secundario.

Tabla 15: Descripción del CUS "Ejecutar Estado de Shader Mapeo Esférico del Entorno (SEM)."

Caso de uso:	Ejecutar Estado de Shader de BumpMapping.
Actor (es):	
Propósito:	Generar sensación de relieve, utilizando la técnica de Bumpmapping.
Resumen:	El sistema calcula las coordenadas de textura para cada vértice, la posición de la cámara, la posición de la luz y transforma estos dos últimos. Para cada pixel calculo su color final.
Referencias:	RF 23, 26, Hereda del CUS 11, extensión del CUS15
Pre-condiciones:	
Curso Normal de los Eventos	
Acción del Actor	**Respuesta del Sistema**
	1- Para cada Vértice del Objeto: 1.1- Genera las coordenadas de la textura. 1.2- Calcula el vector de la Posición de la Cámara. 1.3- Calcula el vector de la Posición de la luz.

	1.4-Obtiene: 1.4.1- Vector Normal. 1.4.2- Vector Binormal. 1.4.3- Vector Tangente. 1.5- Transforma el vector de posición de la cámara usando la normal, binormal y la tangente. 1.6- Transforma el vector de posición de la luz usando la normal, binormal y la tangente.
	2- Para cada Pixel del Objeto: 2.1- Normaliza el vector de posición de la luz. 2.2- Calcula la normal según el color de la textura Bumpmapping en la coordenada 0. 2.3- Calcula la componente ambiental de la luz. 2.4- Calcula la componente difusa de la luz. 2.5- Calcula la componente especular de la luz. 2.6- Calcula el color final del pixel sumando las tres componentes de la luz anterior.
Curso Alternativo de los Eventos	
Post-condiciones:	
Prioridad:	Secundario.

Tabla 16: Descripción del CUS "Ejecutar Estado de Shader de BumpMapping"

Conclusiones

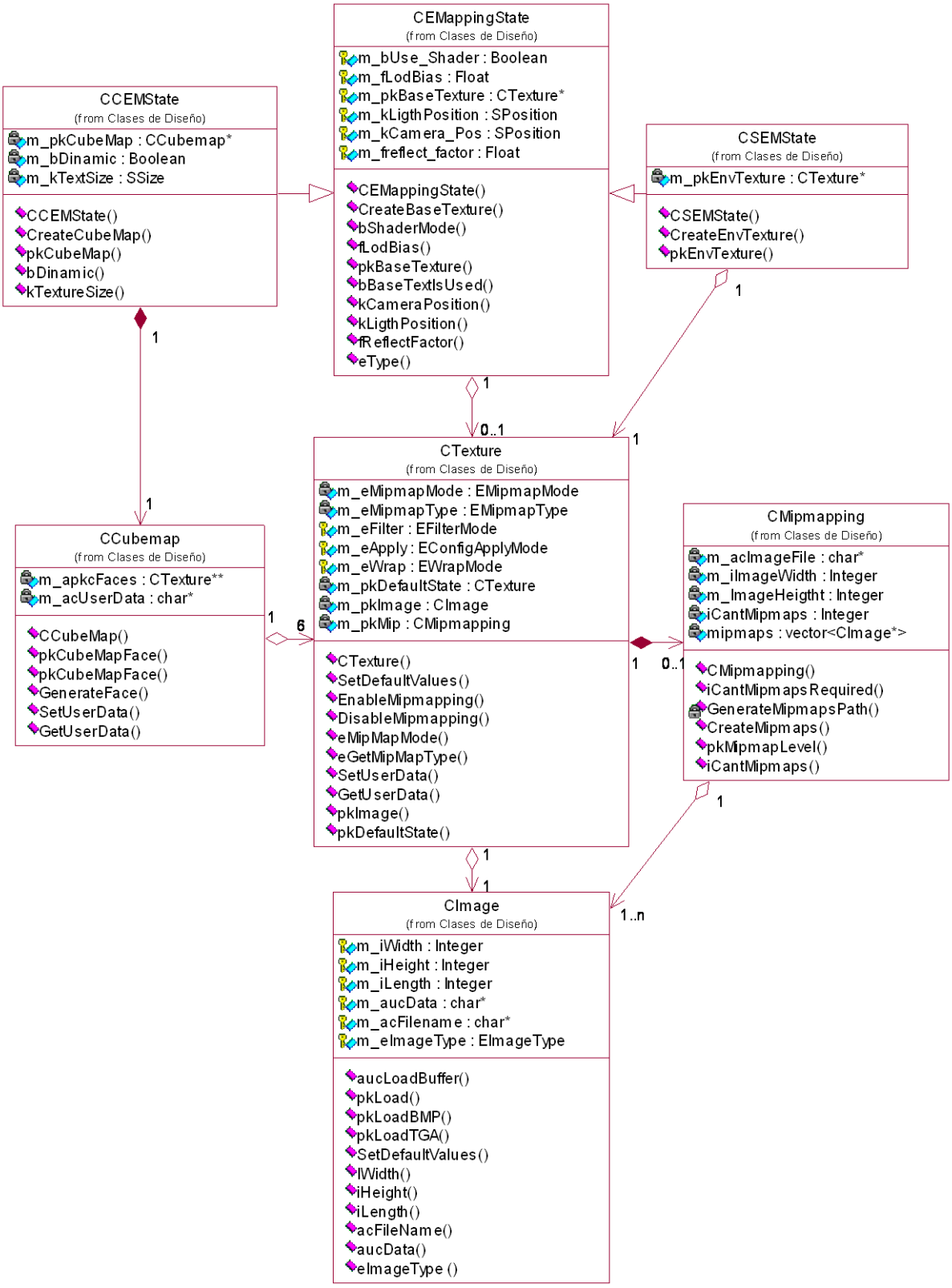
En este capítulo se definió las reglas del negocio, se le dio una primera visión de lo que se pretende obtener con este proyecto. Para ello quedaron establecidos los requisitos funcionales y los no funcionales del mismo, se confeccionaron los modelos de casos de usos del sistema, se muestra los diagramas de casos de usos del sistema y la descripción textual de los mismos, lo que le permitirá a sus futuros usuarios obtener los resultados esperados por el cliente.

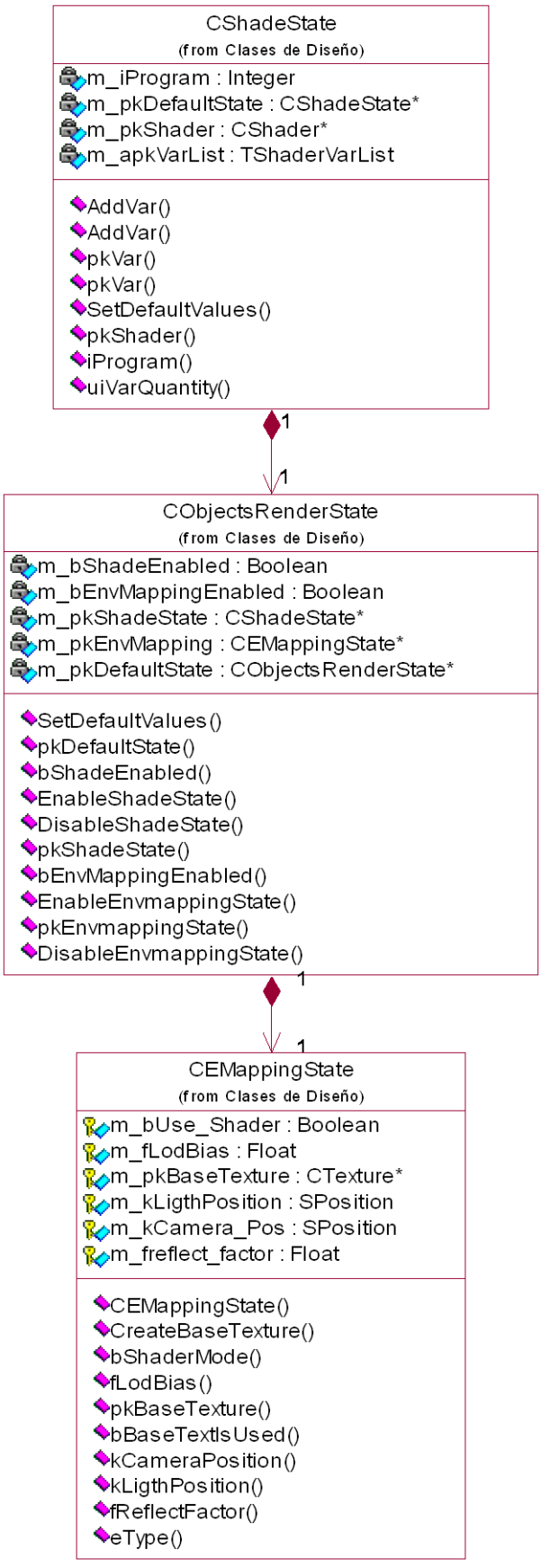
Capítulo 4: Diseño del Sistema

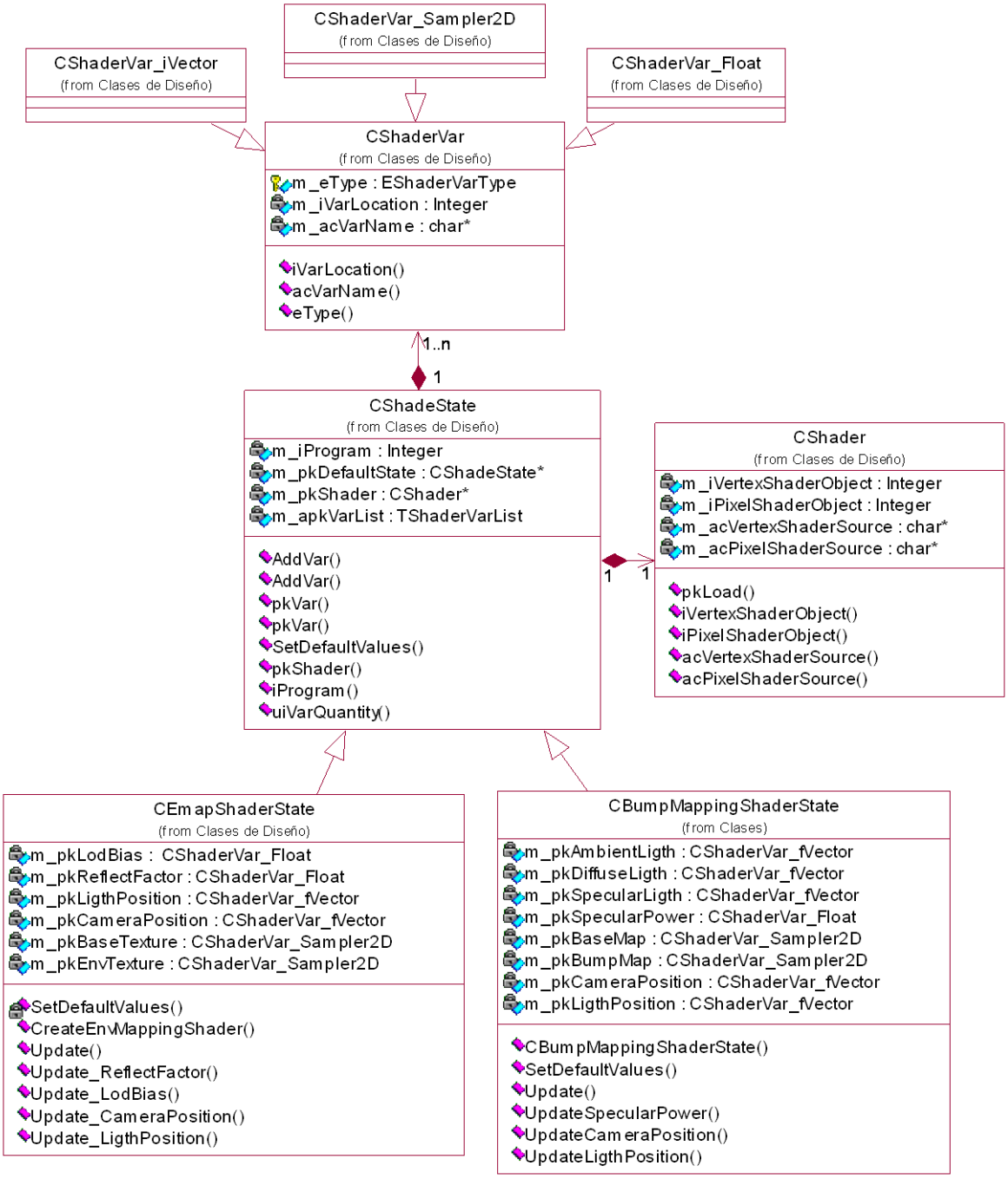
Introducción

En este capítulo se presentan los diagramas de clases del diseño como resultado del refinamiento de las etapas anteriores. Se brinda una breve explicación de las clases de diseño dejando plasmada los atributos y actividad fundamental que desarrollan sus métodos. Se presentan además los diagramas de interacción del diseño, en este caso los diagramas de secuencia de la realización de los casos de usos.

4.1- Diagramas de clases del Diseño.




CShadeState
(from Clases de Diseño)
m_iProgram : Integer
m_pkDefaultState : CShadeState*
m_pkShader : CShader*
m_apkVarList : TShaderVarList
AddVar()
AddVar()
pkVar()
pkVar()
SetDefaultValues()
pkShader()
iProgram()
uiVarQuantity()
1
1
CObjectsRenderState
(from Clases de Diseño)
m_bShadeEnabled : Boolean
m_bEnvMappingEnabled : Boolean
m_pkShadeState : CShadeState*
m_pkEnvMapping : CEMappingState*
m_pkDefaultState : CObjectsRenderState*
SetDefaultValues()
pkDefaultState()
bShadeEnabled()
EnableShadeState()
DisableShadeState()
pkShadeState()
bEnvMappingEnabled()
EnableEnvmappingState()
pkEnvmappingState()
DisableEnvmappingState()
1
1
CEMappingState
(from Clases de Diseño)
m_bUse_Shader : Boolean
m_fLodBias : Float
m_pkBaseTexture : CTexture*
m_kLigthPosition : SPosition
m_kCamera_Pos : SPosition
m_freflect_factor : Float
CEMappingState()
CreateBaseTexture()
bShaderMode()
fLodBias()
pkBaseTexture()
bBaseTextIsUsed()
kCameraPosition()
kLigthPosition()
fReflectFactor()
eType()


CShaderVar_Sampler2D
(from Clases de Diseño)
CShaderVar_iVector
(from Clases de Diseño)
CShaderVar_Float
(from Clases de Diseño)
CShaderVar
(from Clases de Diseño)
m_eType : EShaderVarType
m_iVarLocation : Integer
m_acVarName : char*
iVarLocation()
acVarName()
eType()
1..n
1
CShadeState
(from Clases de Diseño)
m_iProgram : Integer
m_pkDefaultState : CShadeState*
m_pkShader : CShader*
m_apkVarList : TShaderVarList
AddVar()
AddVar()
pkVar()
pkVar()
SetDefaultValues()
pkShader()
iProgram()
uiVarQuantity()
1
1
CShader
(from Clases de Diseño)
m_iVertexShaderObject : Integer
m_iPixelShaderObject : Integer
m_acVertexShaderSource : char*
m_acPixelShaderSource : char*
pkLoad()
iVertexShaderObject()
iPixelShaderObject()
acVertexShaderSource()
acPixelShaderSource()
CEmapShaderState
(from Clases de Diseño)
m_pkLodBias : CShaderVar_Float
m_pkReflectFactor : CShaderVar_Float
m_pkLigthPosition : CShaderVar_fVector
m_pkCameraPosition : CShaderVar_fVector
m_pkBaseTexture : CShaderVar_Sampler2D
m_pkEnvTexture : CShaderVar_Sampler2D
SetDefaultValues()
CreateEnvMappingShader()
Update()
Update_ReflectFactor()
Update_LodBias()
Update_CameraPosition()
Update_LigthPosition()
CBumpMappingShaderState
(from Clases)
m_pkAmbientLigth : CShaderVar_fVector
m_pkDiffuseLigth : CShaderVar_fVector
m_pkSpecularLigth : CShaderVar_fVector
m_pkSpecularPower : CShaderVar_Float
m_pkBaseMap : CShaderVar_Sampler2D
m_pkBumpMap : CShaderVar_Sampler2D
m_pkCameraPosition : CShaderVar_fVector
m_pkLigthPosition : CShaderVar_fVector
CBumpMappingShaderState()
SetDefaultValues()
Update()
UpdateSpecularPower()
UpdateCameraPosition()
UpdateLigthPosition()

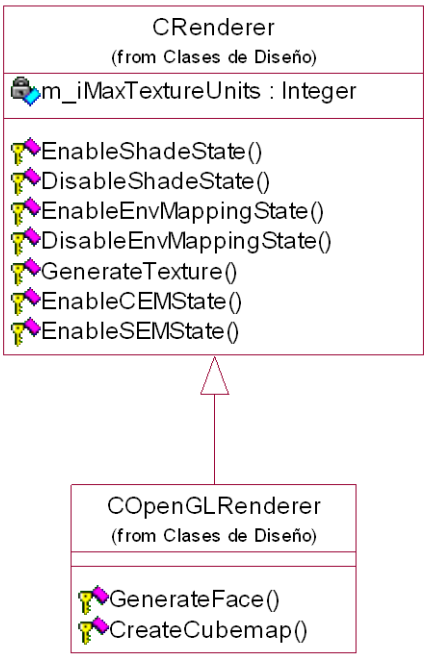


Figura 27: Diagrama de Clase del Diseño Parte I, II, III, IV.

Las clases anteriores están distribuidas en las 2 capas inferiores de la STK -es decir la capa Engine y la capa Renderer-, como se muestra a continuación.

Capa Engine	**Capa Renderer**
CImage, CTeture, CMipmapping, CCubeMap, CCEMState, CSEMState, CEMappingState, CObjectsRenderState, CShadeState, CShader, CShaderVar, CEmapShaderState, CBumpMapShaderState	CRenderer COpenGLRenderer

4.1.1 Descripción de las clases del diseño.

Las clases de diseño que se describen a continuación son las más importantes dentro del diseño de clases del proyecto. Los métodos de acceso a miembros de clases (set y get) no son especificados en las descripciones, ni tampoco los destructores, puesto que su funcionalidad es sencilla. Las clases de la STK que se utilizan no se describen, ya que solo son reutilizadas, en caso de redefinir o agregar algún método de alguna clase ya implementado por la STK, esta clase con sus métodos será referenciado en este epígrafe. Si la clase reutilizada de la STK es muy significativa entonces hacemos referencia a ella aquí también.

Nombre	CImage	
Tipo de clase	Entidad	
Atributo		**Tipo**
m_eImageType		EImageType
m_iWidth		int
m_iHeight		int
m_iLength		int
m_aucData		unsigned char*
m_acFilename		char*
Para cada responsabilidad		
Nombre	SetDefaultValues	
Descripción	Inicializa los valores por defectos de los atributos definidos en la clase.	
Nombre	bIsPowerOfTwo (int arg_iValue)	
Descripción	Devuelve un booleano especificando si es potencia de dos del valor pasado por parámetro.	
Nombre	aucLoadBuffer (const char* arg_acFilename)	
Descripción	Carga la imagen cuya dirección es pasada por parámetro y devuelve los datos de imagen cargada en el buffer.	
Nombre	pkLoad (char* arg_acFilename, bool arg_bRequirePowerOfTwo)	
Descripción	Carga la imagen cuya dirección es pasada por parámetros, verifica si es potencia de dos, en caso de ser una imagen de tipo BMP manda a cargar y devuelve esta imagen BMP, y en caso de ser TGA carga y devuelve esta imagen de tipo TGA.	
Nombre	pkLoadTGA (char* arg_acFilename, bool arg_bRequirePowerOfTwo = false)	
Descripción	Carga la imagen cuya dirección es pasada por parámetro, verifica si es potencia de dos, y devuelve esta imagen de tipo TGA.	
Nombre	pkLoadBMP (char* arg_acFilename, bool arg_bRequirePowerOfTwo)	
Descripción	Carga la imagen cuya dirección es pasada por parámetro, verifica si es potencia de dos, y devuelve esta imagen de tipo BMP.	
Nombre	eImageType ()	
Descripción	Retorna el tipo de imagen.	
Nombre	iWidth ()	
Descripción	Retorna m_iWidth.	
Nombre	iHeight ()	
Descripción	Retorna m_iHeight.	
Nombre	iLength ()	
Descripción	Retorna m_iLength.	
Nombre	aucData ()	
Descripción	Retorna m_aucData. Memoria video.	
Nombre	acFileName()	

Descripción	Retorna m_acFilename.

Tabla 17: Descripción de la clase del diseño "CImage"

Nombre	CTexture	
Tipo de clase	Entidad	
Atributo		**Tipo**
m_pkImage		CImage*
m_eApply		EConfigApplyMode
m_eWrap		EWrapMode
m_eFilter		EFilterMode
m_eMipmap		EMipmapMode
m_eMipmapType		EMipmapType
m_pkMip		CMipmapping *
m_pkDefaultState		CTexture
Para cada responsabilidad		
Nombre	CTexture()	
Descripción	Constructor vacío por defecto, inicializa valores.	
Nombre	CTexture(char*arg_imgPath)	
Descripción	Constructor sobrecargado que crea una imagen y carga el contenido de la misma cuya dirección es especificada.	
Nombre	SetDefaultValues ()	
Descripción	Inicializa los valores por defectos relacionados con la textura.	
Nombre	EnableMipmapping(EMipmapMode arg_emode, EMipmapType arg_etype)	
Descripción	Se le pasa el modo y tipo de Mipmap a aplicar y lo habilita.	
Nombre	DisableMipmapping()	
Descripción	Deshabilita el estado de Mipmapping.	
Nombre	eMipMapMode()	
Descripción	Retorna m_eMipmap.	
Nombre	eGetMipMapType()	
Descripción	Retona m_eMipmapType.	
Nombre	SetUserData(int arg_iSize, void* arg_pvData)	
Descripción	Adiciona el índice de la textura en la memoria de video.	
Nombre	GetUserData(int arg_iSize, void* arg_pvData)	
Descripción	Obtiene el índice de la textura en la memoria de video.	
Nombre	pkDefaultState()	
Descripción	Retorna el estado por defecto m_pkDefaultState	
Nombre	pkImage ()	
Descripción	Retorna la imagen m_pkImage	

Tabla 18: Descripción de la clase del diseño "CTexture"

Nombre	CMipmapping	
Tipo de clase	Entidad	
Atributo		**Tipo**
mipmaps		vector<CImage*>

m_acImageFile	const char*
m_iImageWidth	int
m_iImageHeigtht	int
m_iCantMipmaps	int
Para cada responsabilidad	
Nombre	CMipmapping ()
Descripción	Constructor vacío por defecto.
Nombre	CMipmapping(const char* arg_acImageFile, int arg_ImageWidth, int arg_iImageHeigtht)
Descripción	Constructor lleno.
Nombre	CreateMipmaps ()
Descripción	Verifica que estén cargadas todas la imágenes para cada nivel de mipmap devuelve un booleano.
Nombre	iCantMipmapsRequired ()
Descripción	Dada la imagen base o del nivel 0, calcula la cantidad de niveles de mipmaps dando el ancho y altura de la imagen. Se calcula a partir del valor máximo entre ancho y alto.
Nombre	GenerateMipmapsPath ()
Descripción	Teniendo la textura base, genero las demás texturas para cada nivel de mipmap.
Nombre	pkMipmapLevel(int arg_iLevel)
Descripción	Devuelve la imagen cargada según el nivel de mipmap especificado.
Nombre	iCantMipmaps ()
Descripción	Devuelve la cantidad de niveles de mipmap.

Tabla 19: Descripción de la clase del diseño "CMipmapping"

Nombre	CCubemap
Tipo de clase	Entidad
Atributo	**Tipo**
m_pkcFaces	CTexture**
m_acUserData;	char*
Para cada responsabilidad	
Nombre	CCubemap ()
Descripción	Constructor vacío por defecto.
Nombre	CCubemap(char*arg_acXPos, char*arg_acXNeg, char*arg_acZPos, char*arg_acZNeg, char*arg_acYPos, char*arg_acYNeg)
Descripción	Contructor lleno paso la direcciones de las seis texturas de cada una de las caras del cubo, para sus tres ejes.
Nombre	pkCubeMapFace(int arg_iFace)
Descripción	Devuelve la textura de la cara del cubo cuyo identificador de número de cara es especificado por parámetro.
Nombre	pkCubeMapFace(ECubeMapFace*arg_eFace)

Descripción	Devuelve la textura de la cara del cubo cuyo identificador de posición por eje de la cara es especificado por parámetro.
Nombre	GenerateFace(int arg_iFace, char* arg_acImageName, unsigned char *arg_ucImageData, int arg_iWidth, int arg_iHeight)
Descripción	Genera la textura de una cara del cubo, cuyo identificador, dirección, datos, ancho y alto de la textura son especificados por parámetros.
Nombre	SetUserData (int arg_iSize, void* arg_pvData);
Descripción	Adiciona el índice de la textura en la memoria de video.
Nombre	GetUserData (int arg_iSize, void* arg_pvData)
Descripción	Obtiene el índice de la textura en la memoria de video.

Tabla 20: Descripción de la clase del diseño "CCubemap"

Nombre	CEMappingState	
Tipo de clase	Entidad	
Atributo		**Tipo**
m_pkBaseTexture		CTexture*
m_bUse_Shader		bool
m_fLodBias		float
m_freflect_factor		float
m_kLigthPosition		SPosition
m_kCamera_Pos		SPosition
Para cada responsabilidad		
Nombre	CEMappingState ()	
Descripción	Constructor vacío por defecto.	
Nombre	CEMappingState(float arg_fOpacityFactor, bool arg_bUseShader, char*arg_acFilename)	
Descripción	Contructor lleno.	
Nombre	CreateBaseTexture(char*arg_acFilename)	
Descripción	Inicializa como textura base, a la textura cuya dirección es especificada por parámetros.	
Nombre	fLodBias ()	
Descripción	Retorna el valor de m_fLodBias, factor de opacidad.	
Nombre	bShaderMode ()	
Descripción	Retorna el valor de m_bUse_Shader, si usa shader o no.	
Nombre	kCameraPosition ()	
Descripción	Retorna el valor de m_kCamera_Pos, la posición de la cámara.	
Nombre	pkBaseTexture ()	
Descripción	Retorna el valor de m_pkBaseTexture, la textura definida como base.	
Nombre	bBaseTextIsUsed ()	
Descripción	Retorna un booleano especificando si es usada la textura base.	
Nombre	fReflectFactor ()	
Descripción	Retorna el valor de m_freflect_factor, factor de reflexión, solo se usa con shader.	

Nombre	kLigthPosition ()
Descripción	Retorna el valor de m_kLigthPosition, posición de la luz, solo se usa con shader.
Nombre	eType ()
Descripción	Método virtual que devuelve el tipo de mapeo de entorno usado: Esférico o Cúbico.

Tabla 21: Descripción de la clase del diseño "CEMappingState"

Nombre	CCEMState : CEMappingState	
Tipo de clase	Entidad	
Atributo		**Tipo**
m_pkCubeMap		CCubemap*
m_bDinamic		bool
m_kTextSize		SSize
Para cada responsabilidad		
Nombre	CCEMState(float arg_fOpacityFactor, bool arg_bUseShader, char*arg_acFilename, bool arg_bDinamic, CCubemap *arg_pkCubeMap)	
Descripción	Constructor lleno. Inicializa y asigna los valores pasados por parámetros.	
Nombre	CreateCubeMap(char*arg_acXPos, char*arg_acXNeg, char*arg_acZPos, char*arg_acZNeg, char*arg_acYPos, char*arg_acYNeg)	
Descripción	Creo el cubemap asignándole las direcciones de la imágenes a cada cara del mismo.	
Nombre	pkCubeMap()	
Descripción	Retorno el cubemap creado.	
Nombre	kTextureSize()	
Descripción	Retorna el tamaño de la textura, pueden ser modificadas por el usuario en caso de usar mapeo cúbico de forma dinámica.	
Nombre	bDinamic()	
Descripción	Retorna si será dinámico o no el mapeo cúbico de entornos.	

Tabla 22: Descripción de la clase del diseño "CCEMState"

Nombre	CSEMState : CEMappingState	
Tipo de clase	Entidad	
Atributo		**Tipo**
m_pkEnvTexture		CTexture*
Para cada responsabilidad		
Nombre	CSEMState()	
Descripción	Constructor vacío por defecto.	
Nombre	CSEMState(float arg_fOpacityFactor, bool arg_bUseShader, char*arg_acFilename, char* arg_acEnvFileName)	
Descripción	Constructor lleno. Inicializa y asigna los valores pasados por parámetros.	

Nombre	CreateEnvTexture(char*arg_acEnvFileName)
Descripción	Creo la textura de la envoltura de la esfera, cuya dirección es pasada por parámetro.
Nombre	pkEnvTexture()
Descripción	Retorna la textura de entorno creada.

Tabla 23: Descripción de la clase del diseño "CSEMState"

Nombre	CObjectsRenderState	
Tipo de clase	Control	
Atributo		**Tipo**
m_bShadeEnabled		bool
m_bEnvMappingEnabled		bool
m_pkShadeState		CShadeState*
m_pkEnvMapping		CEMappingState*
m_pkDefaultState		static CObjectsRenderState*
Para cada responsabilidad		
Nombre	CObjectsRenderState ()	
Descripción	Constructor vacío.	
Nombre	SetDefaultValues ()	
Descripción	Inicializa los valores por defecto de los atributos.	
Nombre	pkDefaultState ()	
Descripción	Retorna m_pkDefaultState, el estado por defecto.	
Nombre	bShadeEnabled ()	
Descripción	Retorna m_bShadeEnabled, si esta habilitado el modo shader.	
Nombre	EnableShadeState (CShadeState* arg_pkValue)	
Descripción	Habilita el estado de shader para el estado de shader especificado por parámetro.	
Nombre	DisableShadeState (bool arg_bClearState)	
Descripción	Deshabilito el estado shader y pregunto si quiero eliminar este estado en caso positivo lo elimino.	
Nombre	pkShadeState ()	
Descripción	Retorno m_pkShadeState, el estado de shader.	
Nombre	bEnvMappingEnabled()	
Descripción	Retorna m_bEnvMappingEnabled, el mapeo de entorno habilitado.	
Nombre	EnableEnvmappingState (CEMappingState* arg_pkEmap)	
Descripción	Habilita el estado del mapeo de entorno, para el estado de entono especificado por parámetros, en caso de no ser especificado se pone por defecto.	
Nombre	DisableEnvmappingState(bool arg_bClearState)	
Descripción	Deshabilito el estado del mapeo de entorno y pregunto si quiero eliminar este estado en caso positivo lo elimino.	
Nombre	pkEnvmappingState()	
Descripción	Retorna m_pkEnvMapping, estado de mapeo de entorno.	

Tabla 24: Descripción de la clase del diseño "CObjectsRenderState"

Nombre	CBumpMappingShaderState : CShadeState

Tipo de clase	Control
Atributo	**Tipo**
m_pkAmbientLigth;	CShaderVar_fVector *
m_pkDiffuseLigth	CShaderVar_fVector *
m_pkSpecularLigth	CShaderVar_fVector *
m_pkSpecularPower	CShaderVar_Float *
m_pkBaseMap	CShaderVar_Sampler2D *
m_pkBumpMap	CShaderVar_Sampler2D *
m_pkCameraPosition	CShaderVar_fVector *
m_pkLigthPosition	CShaderVar_fVector *
Para cada responsabilidad	
Nombre	CBumpMappingShaderState()
Descripción	Cargo el vertex y fragment shader correspondiente al bumbmapping, se crean las variables de shader y se adicionan al mismo.
Nombre	SetDefaultValues()
Descripción	Inicializa los valores por defectos.
Nombre	Update(SColor4f arg_kAmbient, SColor4f arg_kDiffuse, SColor4f arg_kSpecular, float arg_fSpecPower, SPosition arg_kViewPosition, SPosition arg_kLigthPosition)
Descripción	Actualiza o modifica los valores pasados por parámetros a el shader correspondiente.
Nombre	UpdateSpecularPower(float arg_fvalue)
Descripción	Actualiza o modifica el valor del poder especular.
Nombre	UpdateCameraPosition(SPosition arg_kViewPosition)
Descripción	Actualiza o modifica la posición de la cámara.
Nombre	UpdateLigthPosition(SPosition arg_kLigthPosition)
Descripción	Actualiza o modifica la posición de la luz.

Tabla 25: Descripción de la clase del diseño "CBumpMappingShaderState"

Nombre	CEMapShaderState : CShadeState
Tipo de clase	Control
Atributo	**Tipo**
m_eEMapType	EEMapType
m_pkReflectFactor	CShaderVar_Float *
m_pkLodBias	CShaderVar_Float *
m_pkLigthPosition	CShaderVar_fVector*
m_pkBaseTexture	CShaderVar_Sampler2D*
m_pkEnvTexture	CShaderVar_Sampler2D*
m_pkCameraPosition	CShaderVar_fVector*
Para cada responsabilidad	
Nombre	CEMapShaderState(EEMapType arg_eType)

Descripción	Verifica el tipo de mapeo de entorno (Cúbico o Esférico) en dependencia de ello carga el fragment y vertex sahder correspondiente, se crean las variables de shader y se adicionan al mismo.
Nombre	CreateEnvMappingShader()
Descripción	Crear un estado de Mapeo de Entornos.
Nombre	Update(float arg_fReflectFactor, float arg_flodBias, SPosition arg_kLigthPos, SPosition arg_kCameraPos)
Descripción	Actualizar los valores pasados por parámetros a el shader correspondiente.
Nombre	Update_ReflectFactor(float arg_fvalue)
Descripción	Actualizar el valor del factor de reflexión.
Nombre	Update_LodBias(float arg_fvalue)
Descripción	Actualiza el valor del factor de opacidad.
Nombre	Update_CameraPosition(SPosition arg_kCameraPos)
Descripción	Actualiza o modifica el valor de la posición de la cámara.
Nombre	Update_LigthPosition(SPosition arg_kLigthPos)
Descripción	Actualiza o modifica el valor de la posición de la luz.
Nombre	SetDefaultValues()
Descripción	Inicializa los valores por defectos del factor de reflexión, factor de opacidad, y la posición de la luz.

Tabla 26: Descripción de la clase del diseño "CEMapShaderState"

Nombre	CRenderer
Tipo de clase	Control
Atributo	**Tipo**
m_iMaxTextureUnits	int
Para cada responsabilidad	
Nombre	CRenderer ()
Descripción	Constructor vacío.
Nombre	GenerateTexture (CTexture* arg_pkTexture)
Descripción	Método virtual puro redefinido en COpenGLRenderer.
Nombre	EnableShadeState (CShadeState* arg_pkState)
Descripción	Método virtual puro redefinido en COpenGLRenderer.
Nombre	DisableShadeState ()
Descripción	Método virtual puro redefinido en COpenGLRenderer.
Nombre	EnableEnvMappingState(CEMappingState* arg_pkEnvMap)
Descripción	Método virtual puro redefinido en COpenGLRenderer.
Nombre	DisableEnvMappingState()
Descripción	Método virtual puro redefinido en COpenGLRenderer.
Nombre	EnableSEMState(CSEMState* arg_pkSEM)
Descripción	Método virtual puro redefinido en COpenGLRenderer.
Nombre	EnableCEMState(CCEMState* arg_pkCEM)
Descripción	Método virtual puro redefinido en COpenGLRenderer.

Tabla 27: Descripción de la clase del diseño "CRenderer"

Nombre	COpenGLRenderer : CRenderer	
Tipo de clase	Control	
Atributo		**Tipo**
Para cada responsabilidad		
Nombre	COpenGLRenderer ()	
Descripción	Constructor vacío.	
Nombre	GenerateTexture (CTexture* arg_pkTexture)	
Descripción	Verifica si la textura esta generada sino la genera, establece los parámetros especificos de la textura 2D: estableciendo la prioridad entre 0 y 1, modo de envoltura, modo de filtro, modo de mipmap y color de borde.	
Nombre	EnableEnvMappingState(CEMappingState* arg_pkEnvMap)	
Descripción	Verifica si el mapeo de entorno es Esférico o Cúbico, y en dependencia de ello manda a activar el estado correspondiente.	
Nombre	DisableEnvMappingState()	
Descripción	Deshabilita la generación automática de la textura. Deshabilita la textura 2D y cubemap.	
Nombre	EnableShadeState (CShadeState* arg_pkState)	
Descripción	Habilita el estado de Shader y compila el estado de Shader pasado por parámetros, inicializa todas las variables de Shader, en caso de no poder deshabilita el estado de Shader.	
Nombre	DisableShadeState ()	
Descripción	Deshabilita el estado de Shader.	
Nombre	EnableSEMState(CSEMState* arg_pkSEM)	
Descripción	Habilita el mapeado esférico recibido por parámetro.	
Nombre	EnableCEMState(CCEMState* arg_pkCEM)	
Descripción	Habilita el mapeado cúbico recibido por parámetro.	
Nombre	GenerateFace(CTexture* arg_pkTexture, GLenum arg_eTarget)	
Descripción	Verifica si la textura posee imagen en caso positivo verifica si esta activado el mipmapping, utiliza mipmapping manual o basico de opengl o ninguno. Establece los parámetros especificos de la textura 2D:estableciendo la prioridad entre 0 y 1, modo de filtro, modo de mipmap y color de borde.	
Nombre	CreateCubemap(CCubemap * arg_pkCubemap)	
Descripción	Genera las texturas para cada cara del cubemap.	

Tabla 28: Descripción de la clase del diseño "COpenGLRenderer"

Las clases CShaderState, CShader y todas las Clases CShaderVar pertenecen a la herramienta, por tal motivo aunque aparece en el Diagrama de clases del Diseño, no aparecen aquí la descripción de sus clases.

4.2 Diagramas de Secuencia

Dentro de los diagramas de interacción tomamos el diagrama de secuencia que muestra la interacción entre objetos, ordenadas en secuencia temporal durante un escenario concreto.

Cuando los CUS posean varios flujos o subflujos distintos, se muestra diagramas de secuencias para cada uno de ellos.

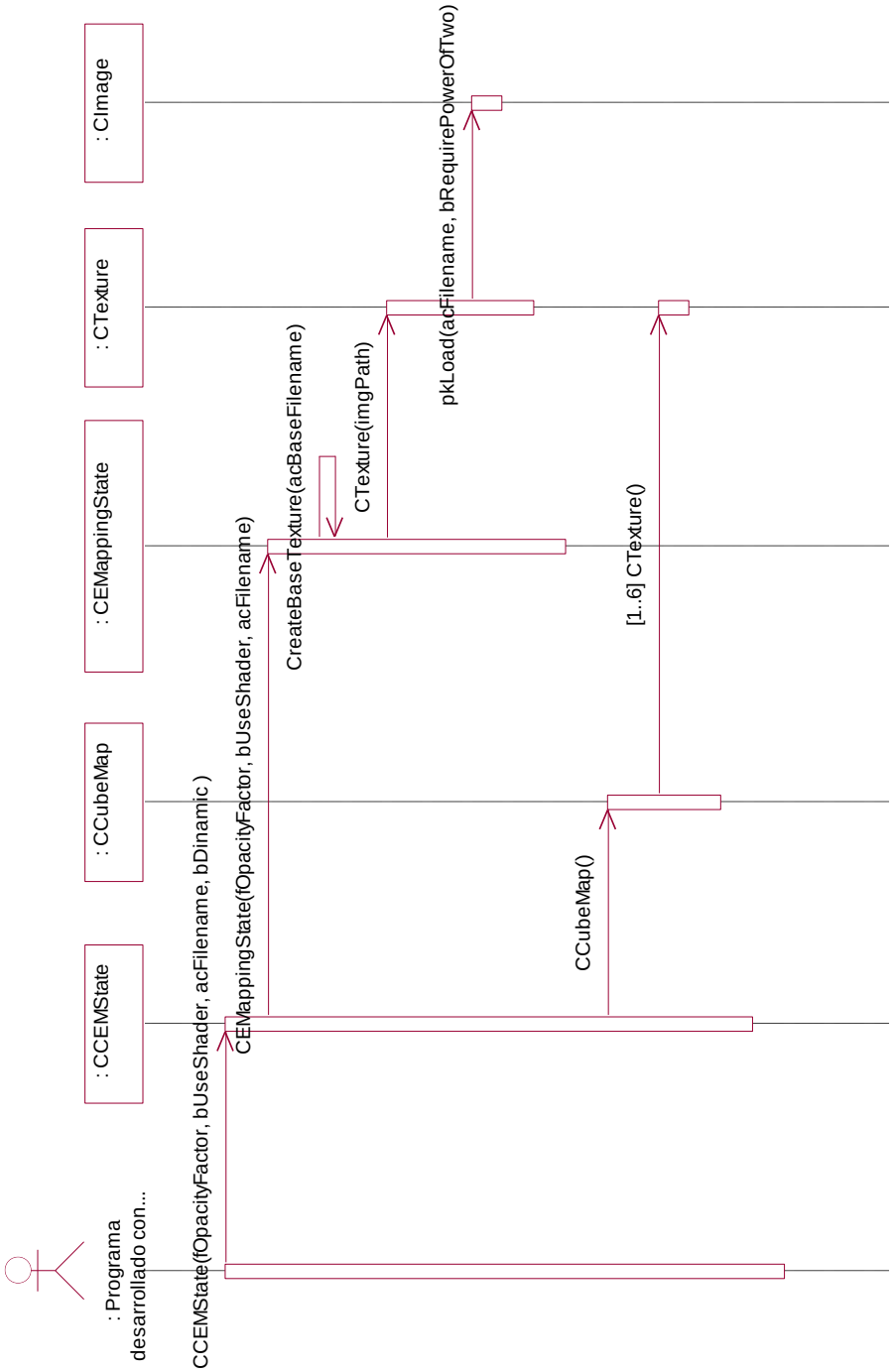


Figura 28: Diagrama de secuencia: Configurar Estado de Mapeo Cúbico del Entorno (CEM) – Sección: Crear Estado.

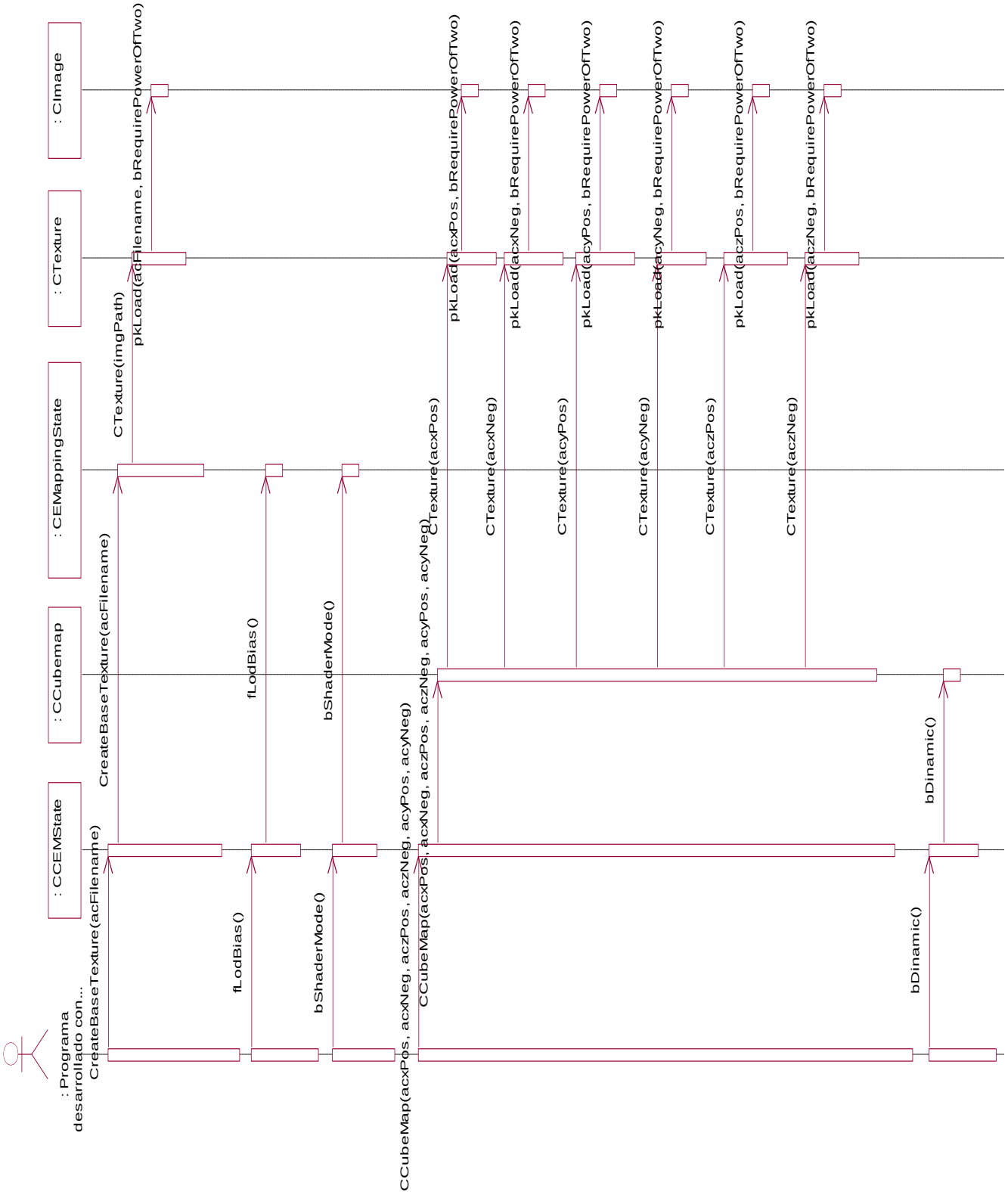


Figura 29: Diagrama de secuencia: Configurar Estado de Mapeo Cúbico del Entorno (CEM) – Sección: Modificar Estado.

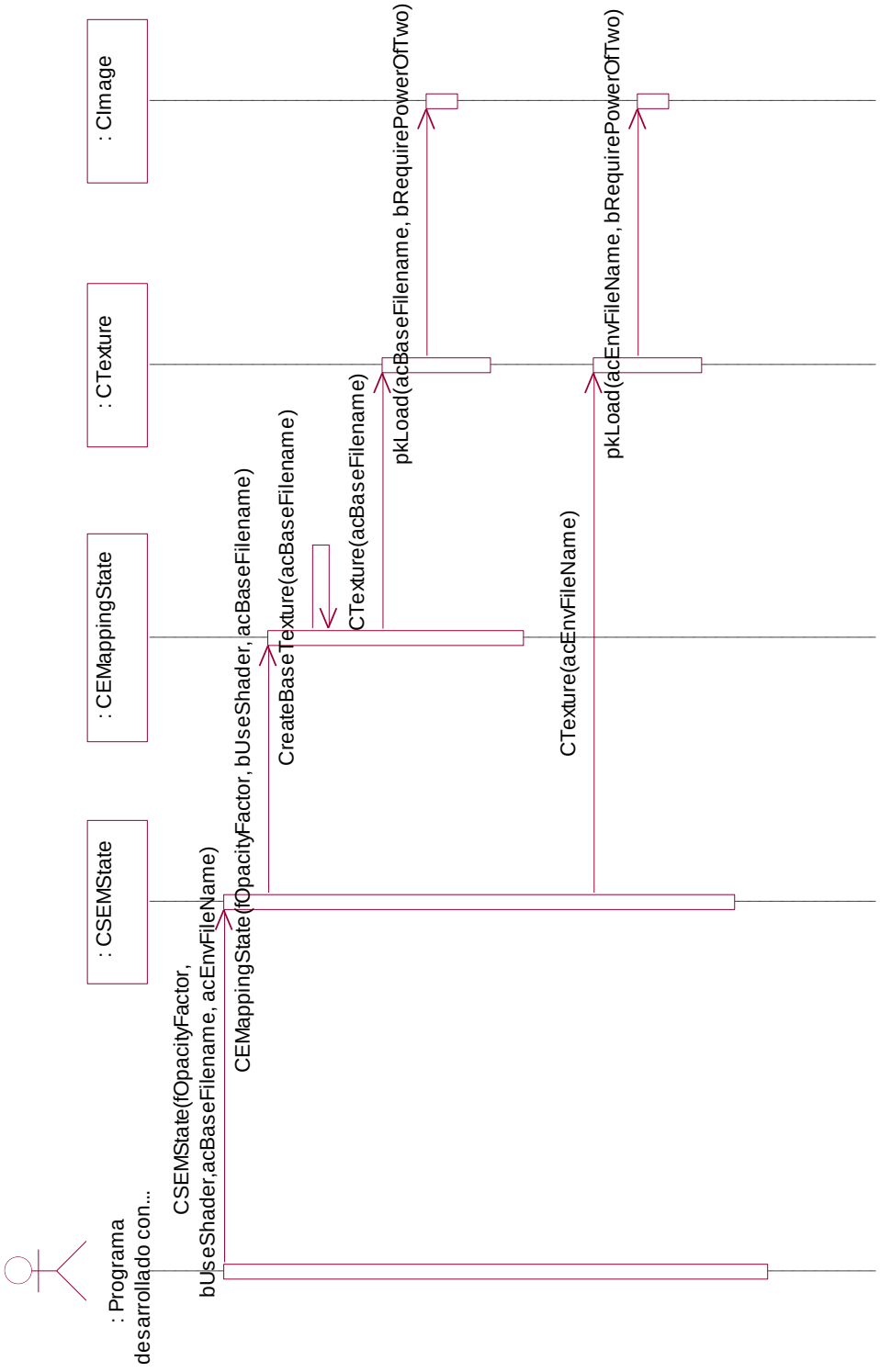


Figura 30: Diagrama de secuencia: Configurar Estado de Mapeo Esférico del Entorno (SEM) – Sección: Crear Estado.

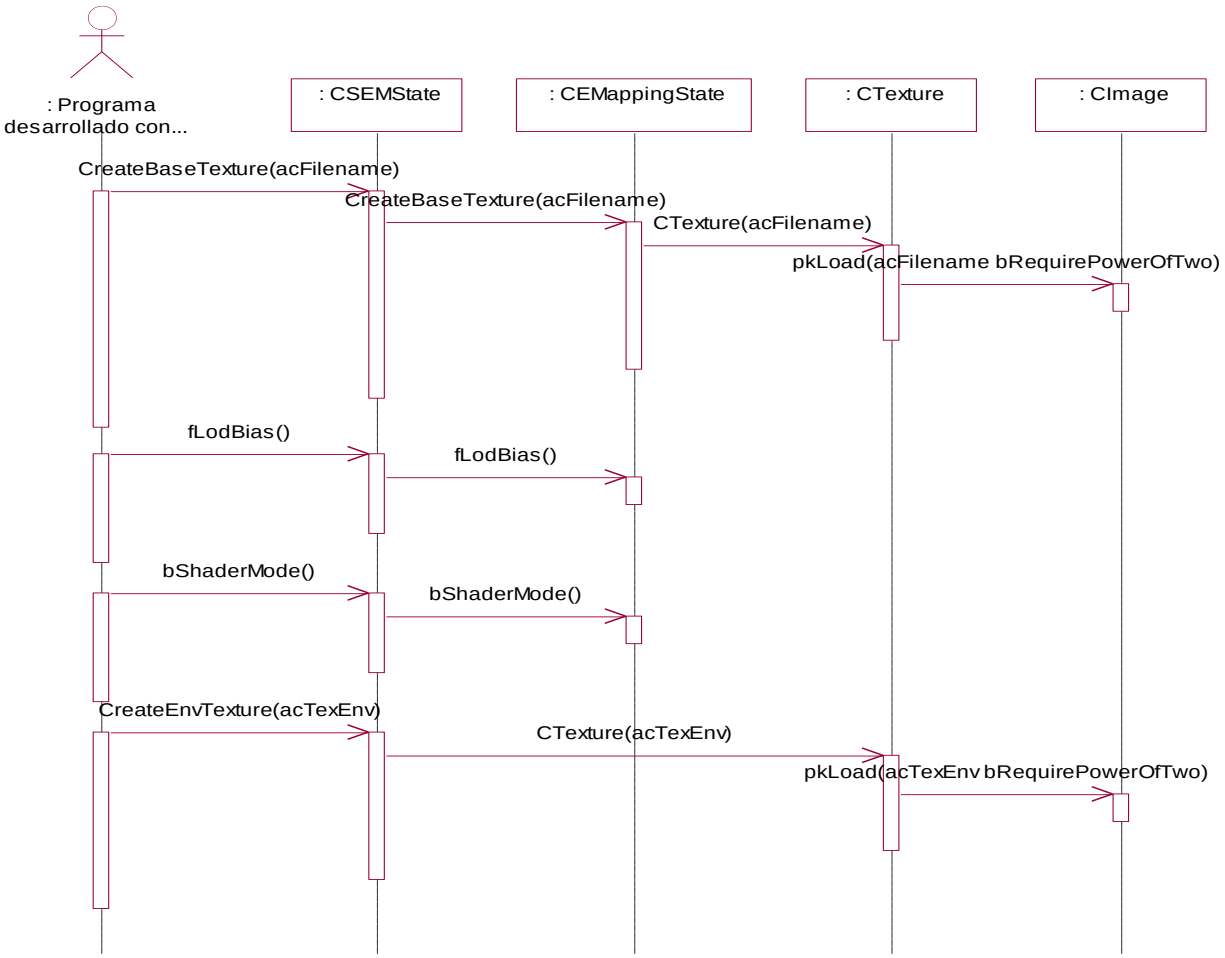


Figura 31: Diagrama de secuencia: Configurar Estado de Mapeo Esférico del Entorno (SEM) – Sección: Modificar Estado.

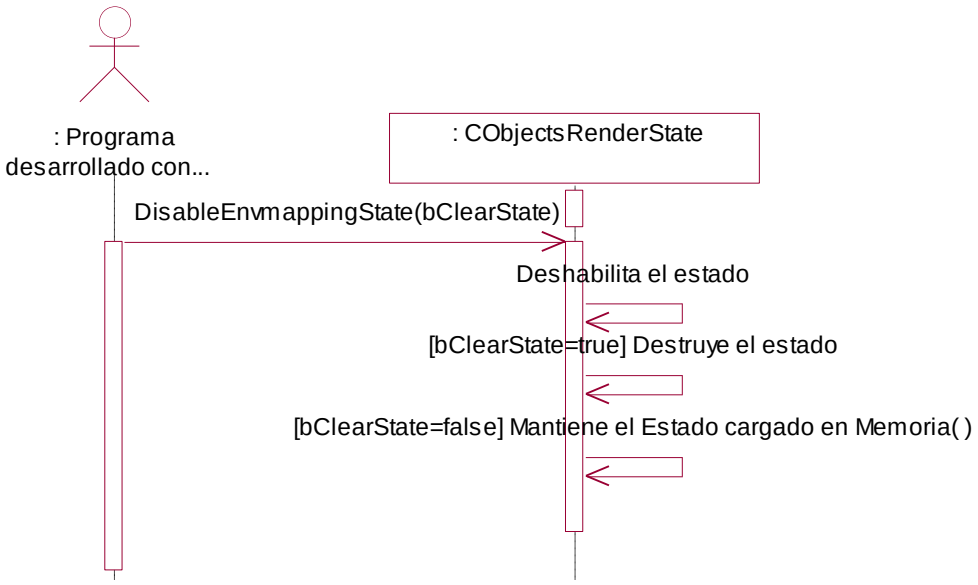


Figura 32: Diagrama de secuencia: Gestionar Estado de Mapeo del Entorno (EMapping) – Sección: Deshabilitar Estado.

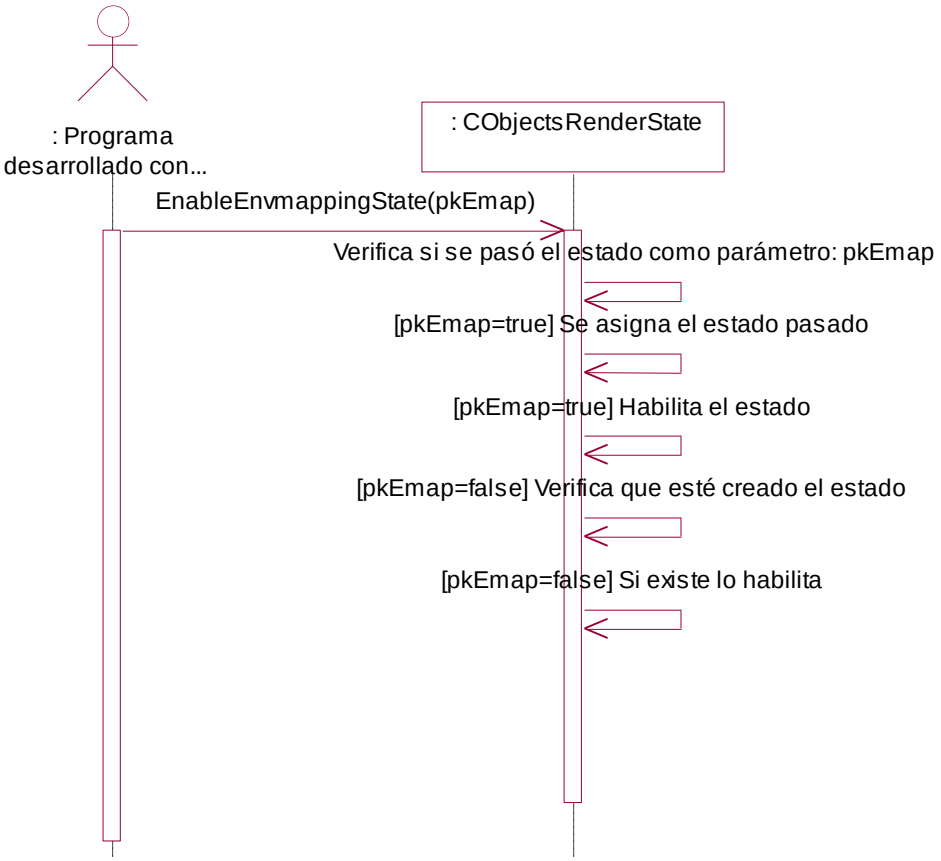


Figura 33: Diagrama de secuencia: Gestionar Estado de Mapeo del Entorno (EMapping) – Sección: Habilitar Estado.

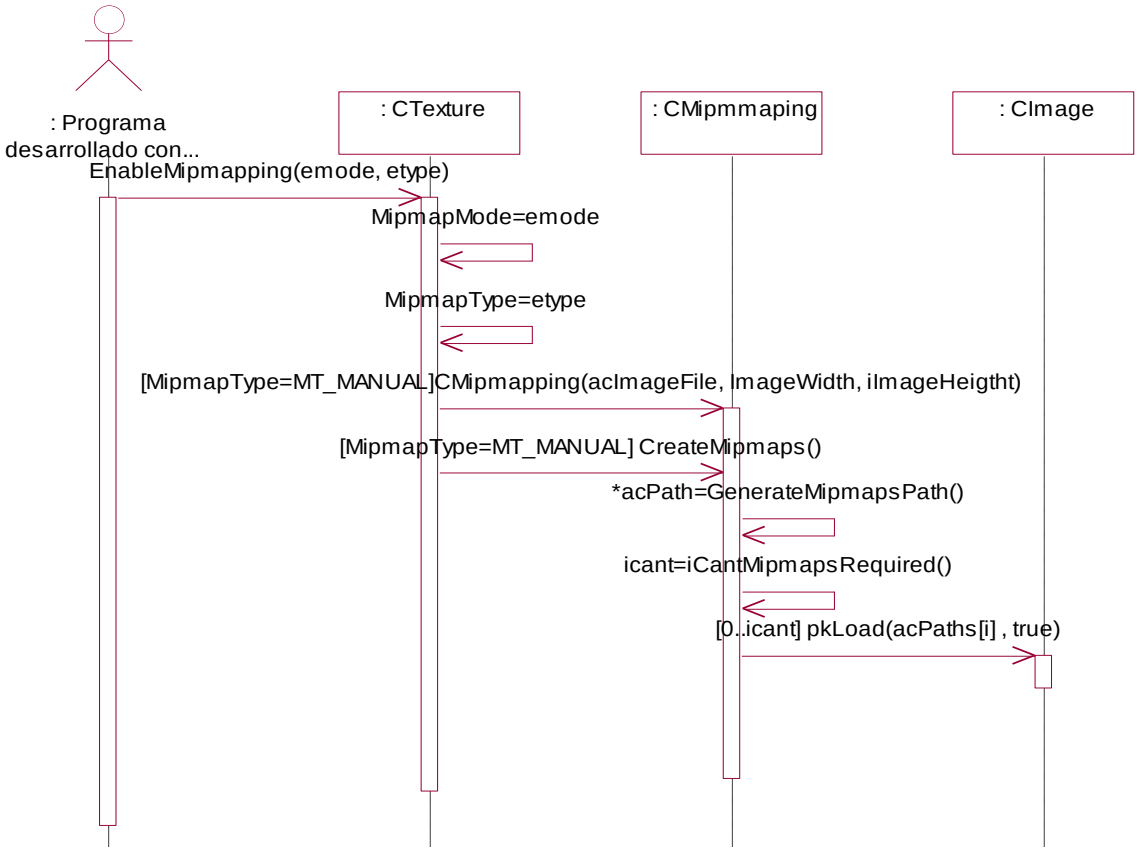


Figura 34: Diagrama de secuencia: Gestionar Estado de Mipmapping – Sección Habilitar

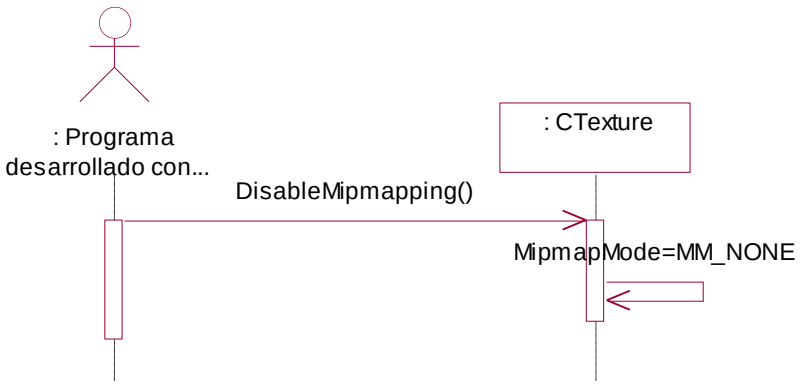


Figura 35: Diagrama de secuencia: Gestionar Estado de Mipmapping – Sección Deshabilitar Estado.

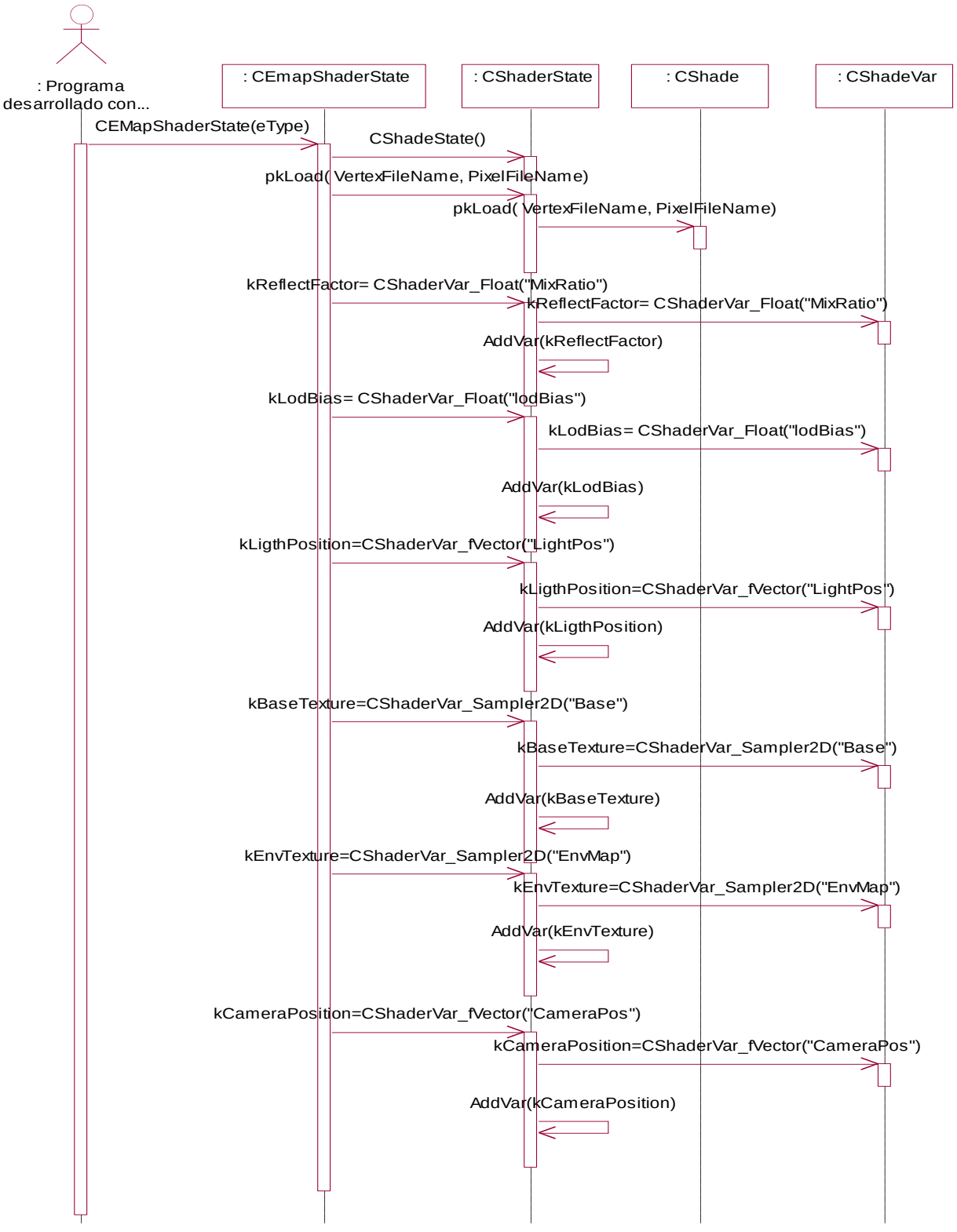


Figura 36: Diagrama de secuencia: Configurar Estado de Shader Mapeo de Entorno (EMapShaderState) – Sección: Crear Estado.

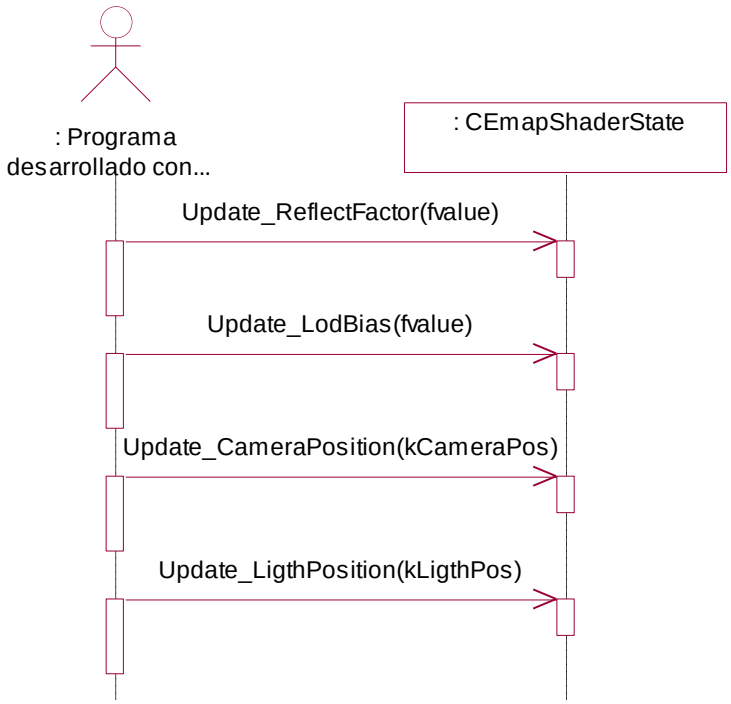


Figura 37: Diagrama de secuencia: Configurar Estado de Shader Mapeo de Entorno (EMapShaderState) – Sección: Modificar Estado.

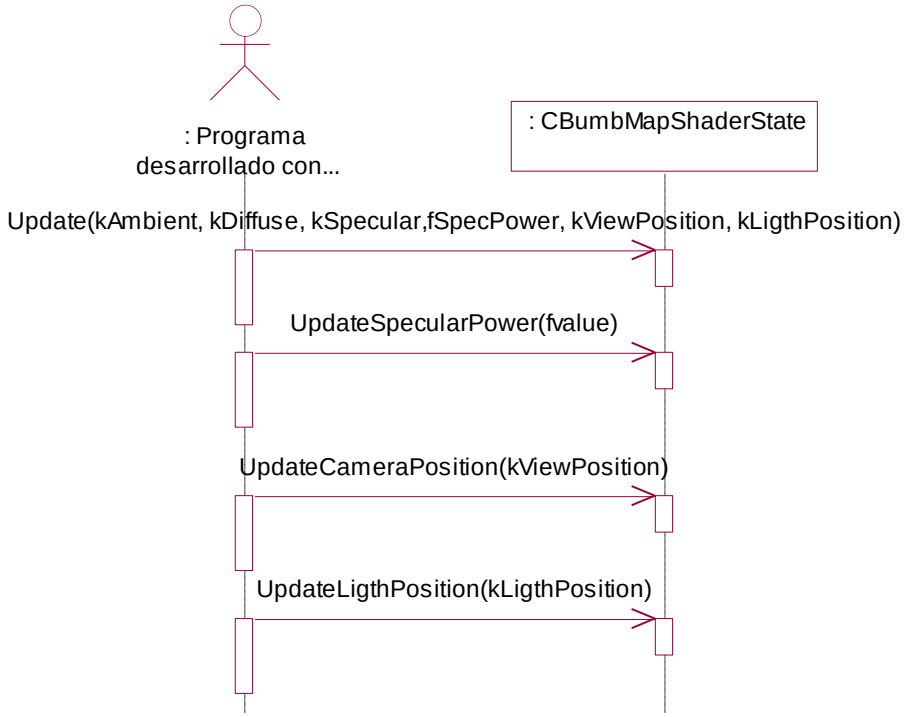


Figura 38: Diagrama de secuencia: Configurar Estado de Shader Bump Mapping (BumpmapShaderState) – Sección Modificar Estado.

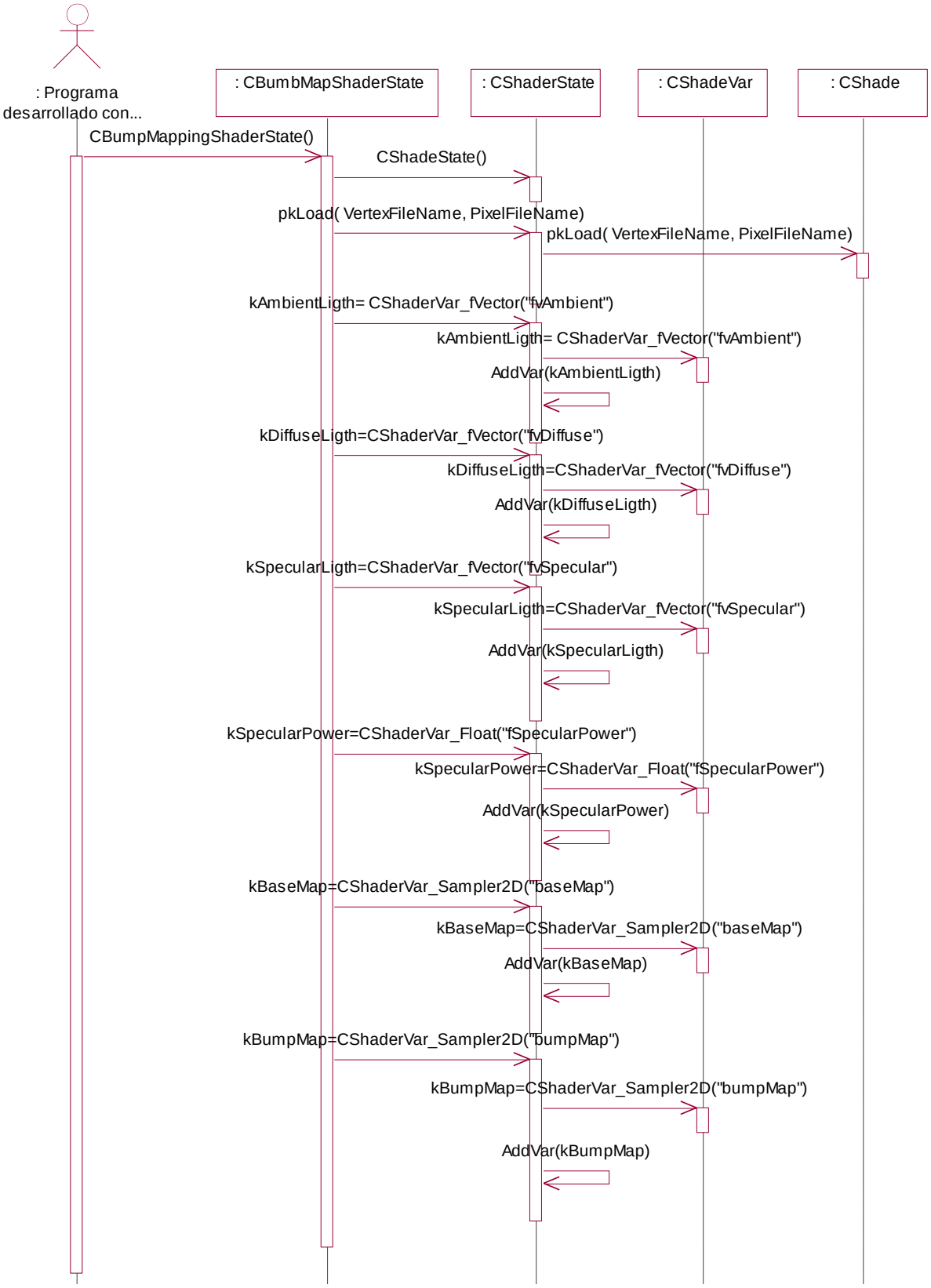


Figura 39: Diagrama de secuencia: Configurar Estado de Shader Bump Mapping (BumpmapShaderState) – Sección Crear Estado.

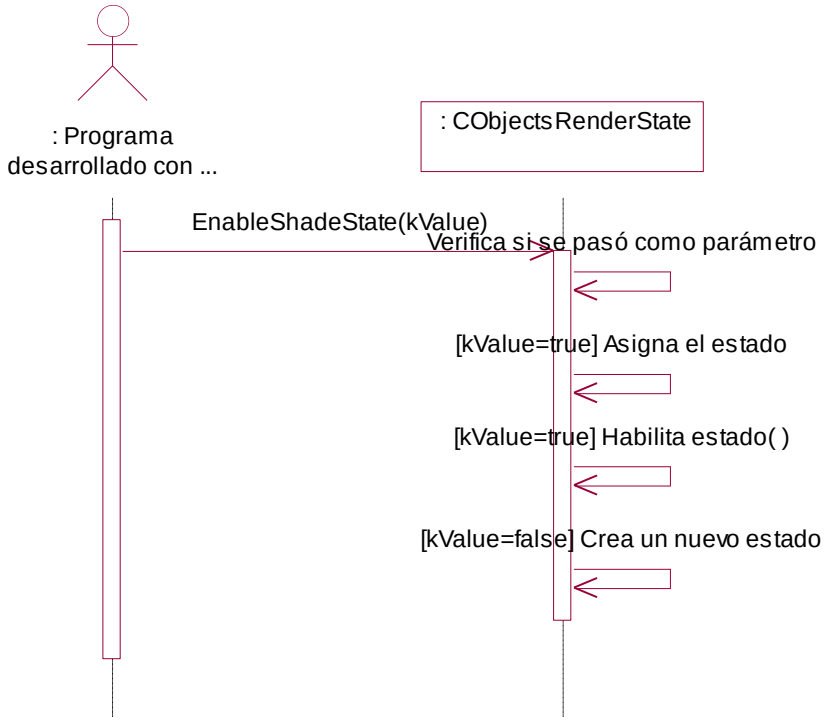


Figura 40: Diagrama de secuencia: Gestionar Estado de Shader – Sección: Habilitar Estado.

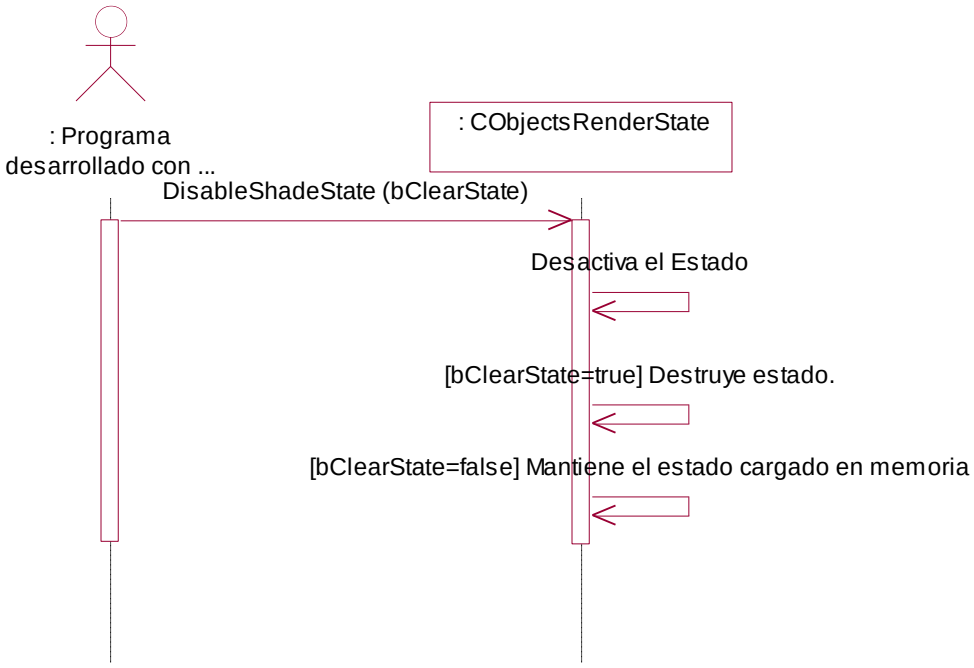


Figura 41: Diagrama de secuencia: Gestionar Estado de Shader – Sección Deshabilitar Estado.

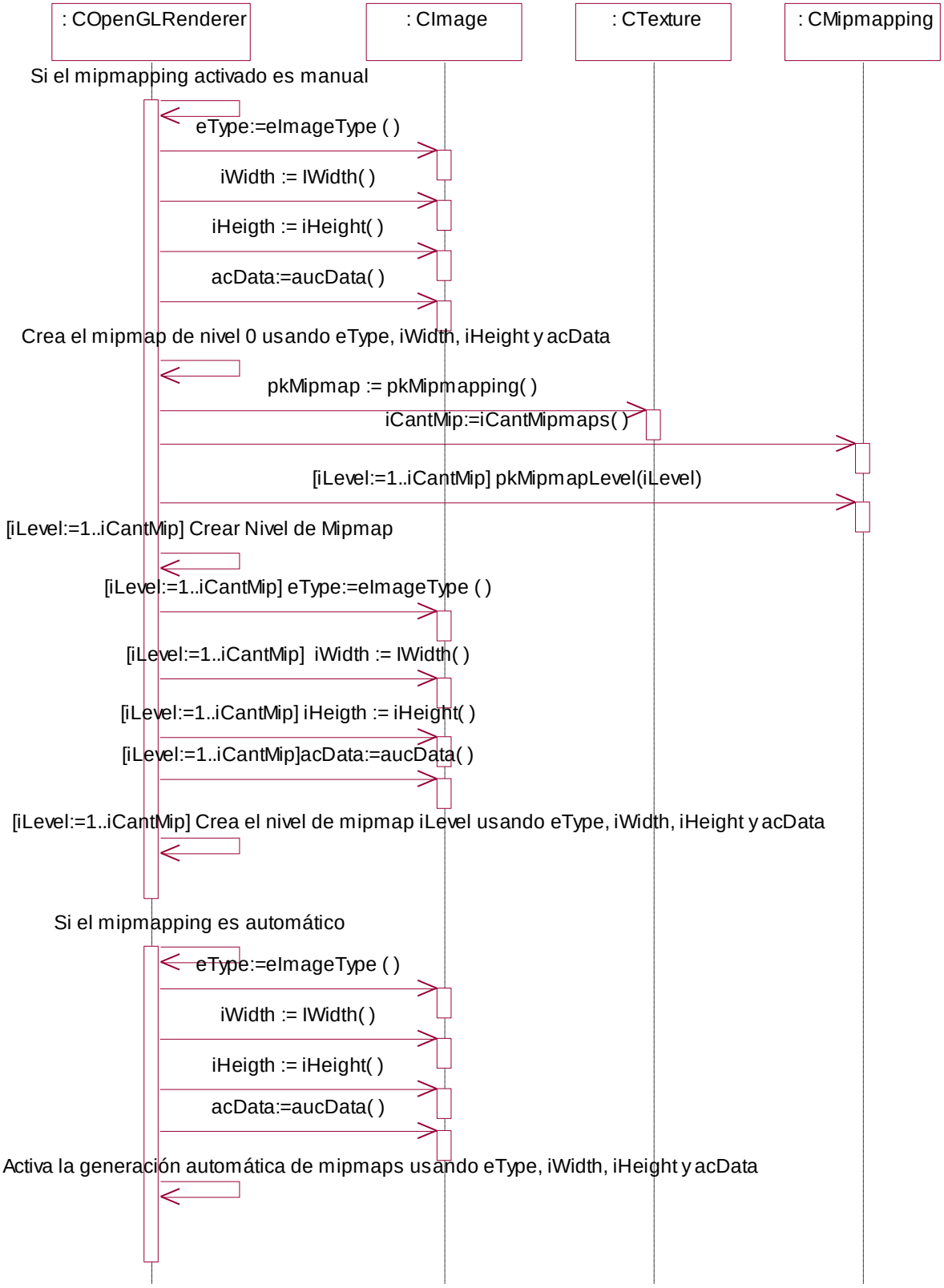


Figura 42: Diagrama de secuencia: Aplicar Mipmapping.

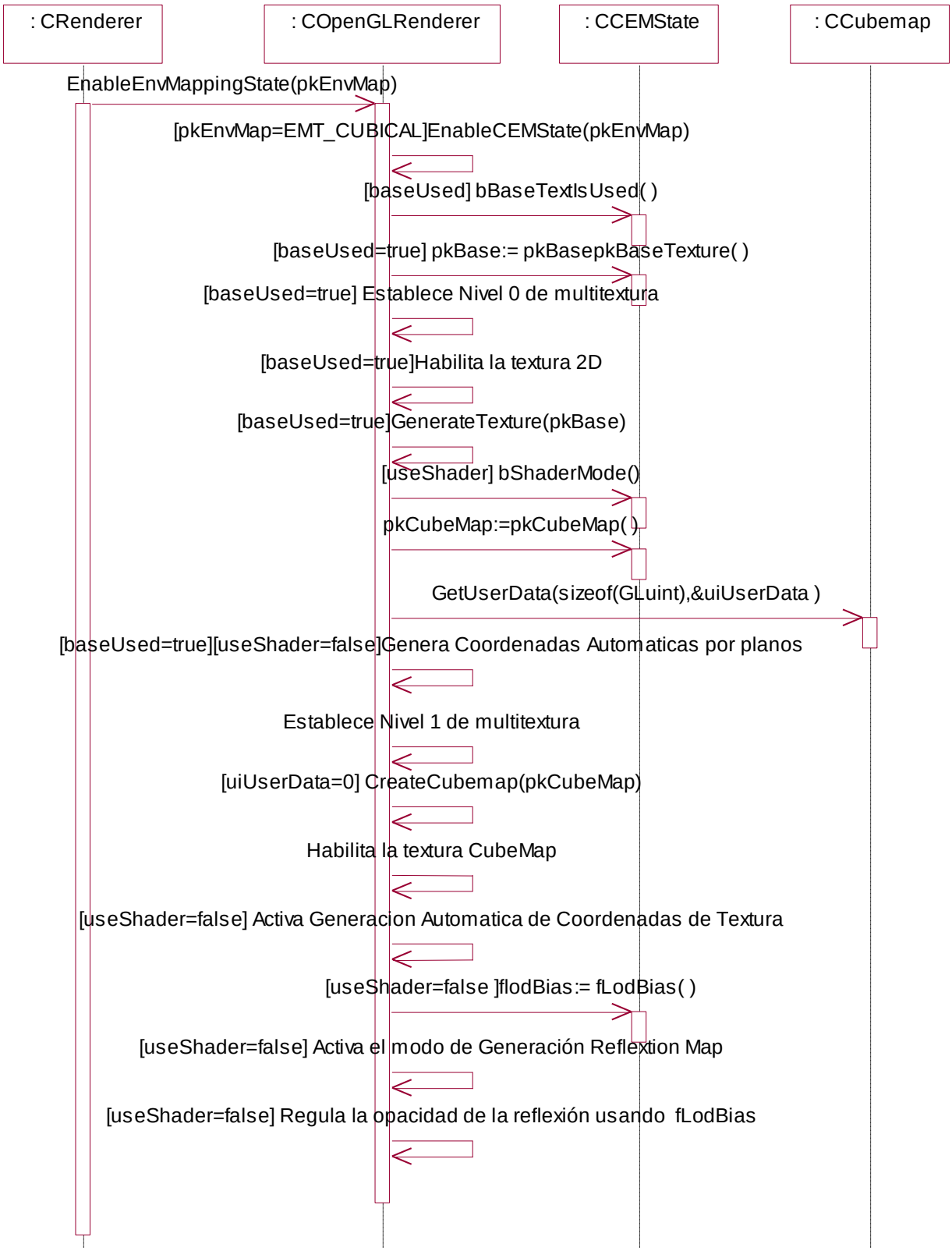


Figura 43: Diagrama de secuencia: Aplicar Mapeo de Entorno– Sección: Habilitar Estado de CEM.

Figura 44: Diagrama de secuencia: Aplicar Mapeo de Entorno– Sección: Habilitar Estado de SEM.

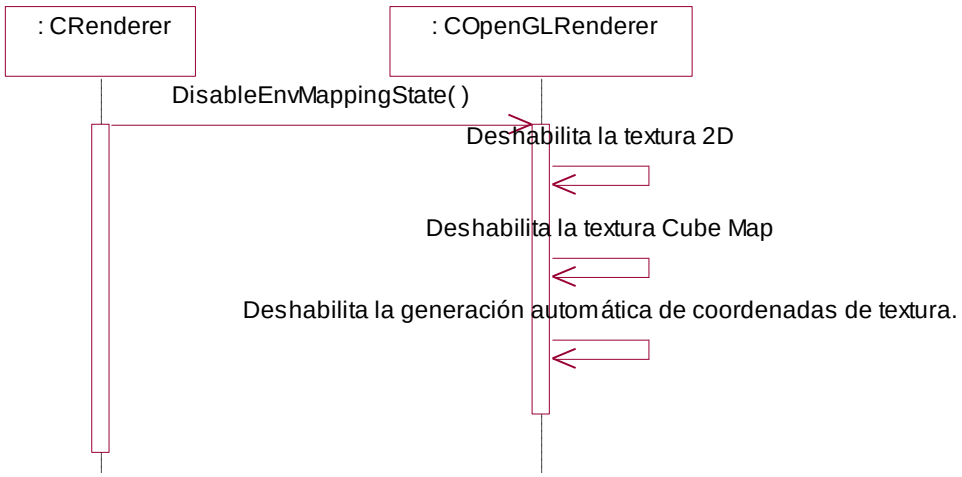


Figura 45: Diagrama de secuencia: Aplicar Mapeo de Entorno– Sección: Deshabilitar Estado.

Conclusiones

Con la conclusión de este capítulo se tiene concebido de una forma bien detallada el diseño completo del sistema y la secuencia de pasos traducidos en mensajes entre clases, con lo cual se puede pasar a la etapa de implementación del proyecto.

Capítulo 5: Implementación del sistema

Introducción

Esta etapa del proyecto constituye el paso del diseño de clases a la creación de componentes físicos, que se traducen en ficheros .h y .cpp correspondientes a la implementación en C++.

5.1- Estándares de codificación

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

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

Nombre de los ficheros:

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

STKNameOfUnits.cpp

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

Constantes:

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

Tipos de datos:

Los tipos se nombrarán siguiendo el siguiente patrón:

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

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

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

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

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

Clases: class **C**ClassName;

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

Declaración de variables:

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

Tipos simples:

bool **b**VarName;
int **i**Name;
unsigned int **ui**Name;
float **f**Name;
char **c**Name;
char* **ac**Name; // arreglo de caracteres
char* **pc**Name; // puntero a un char
char** **aac**Name; // bidimensional
char** **apc**Name; // arreglo de punteros
bool **m_b**MemberVarName; //variable miembro
char **g**cGlobalVarName; //variable global, no se le antepone ""
short **s**Name;

Instancias de tipos creados:

EMyEnumerated **e**Name;

SMyStructure **k**Name;

CClassName **k**ObjectName;

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

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

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

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

Métodos

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

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

Constructor y destructor:

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

~**C**ClassName ();

Funciones:

bool bFunction1 (…);

int* **pi**Function2 (…);

CClassName* **pk**Function3 (…);

Procedimientos:

void Procedure4 (…);

Métodos de acceso a miembros

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

int **i**MyVar; //variable

Obtención del valor:

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

Establecimiento del valor:

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

Obtención y establecimiento del valor:

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

5.2- Diagrama de Componentes

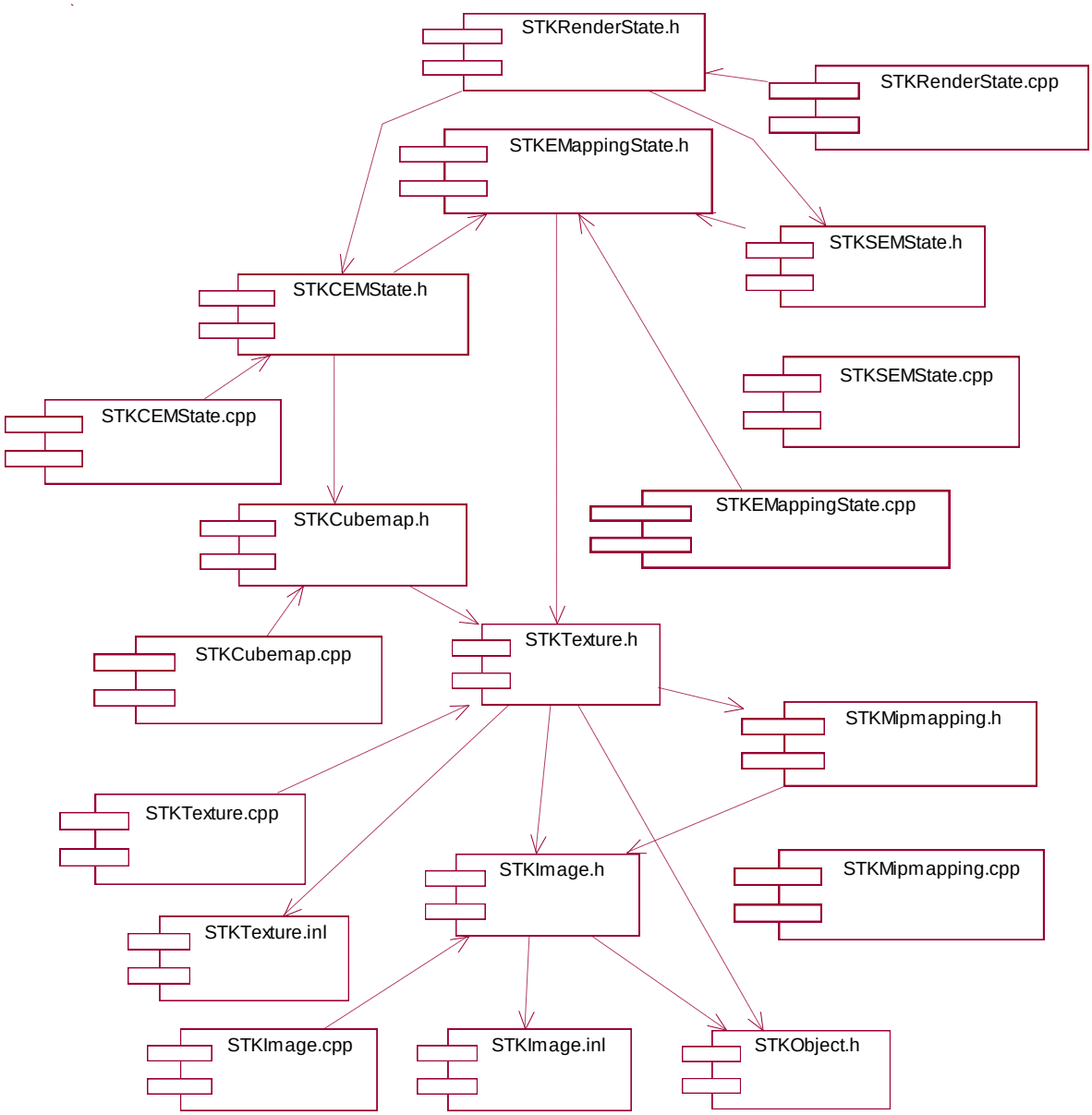


Figura 46: Diagrama de Componentes, Parte I.

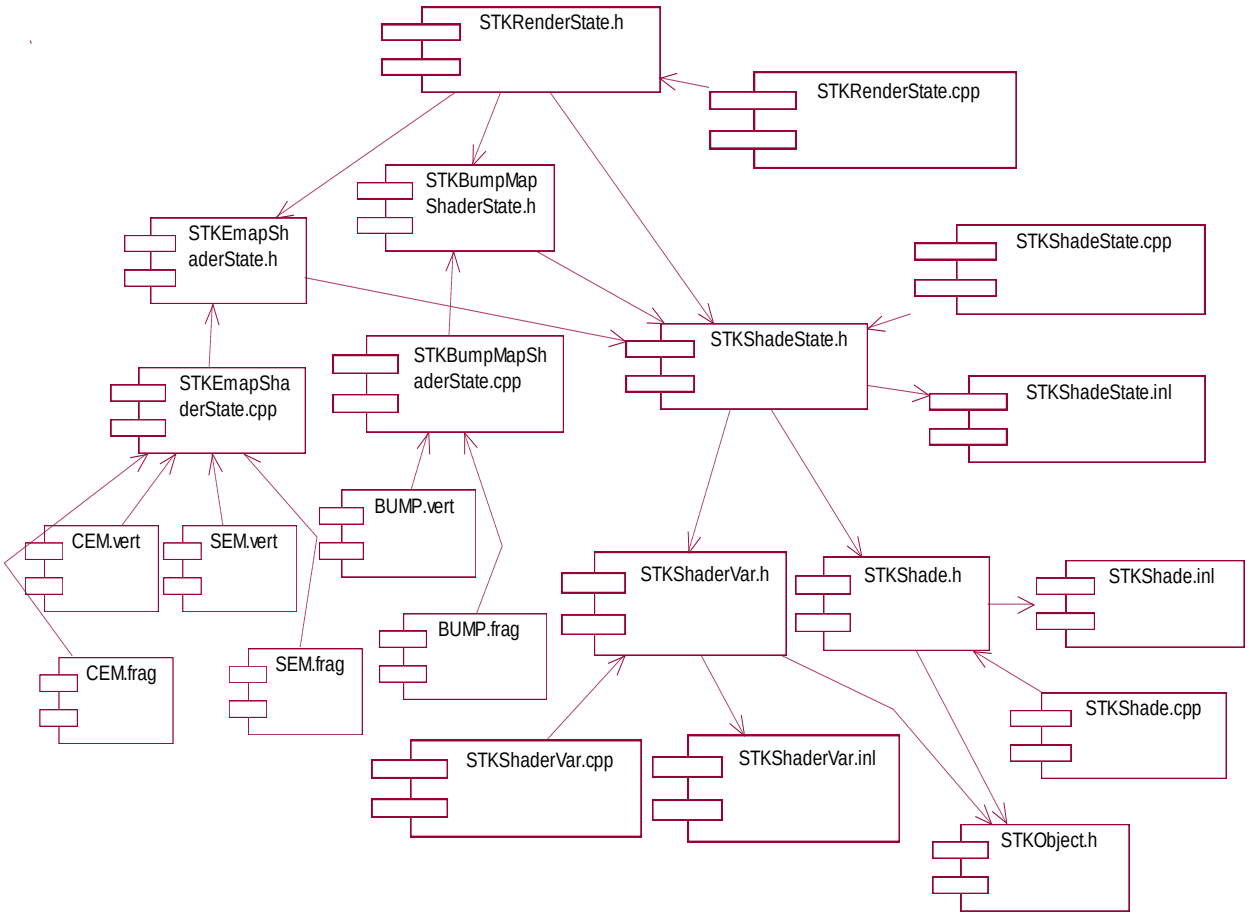


Figura 47: Diagrama de Componentes, Parte II.

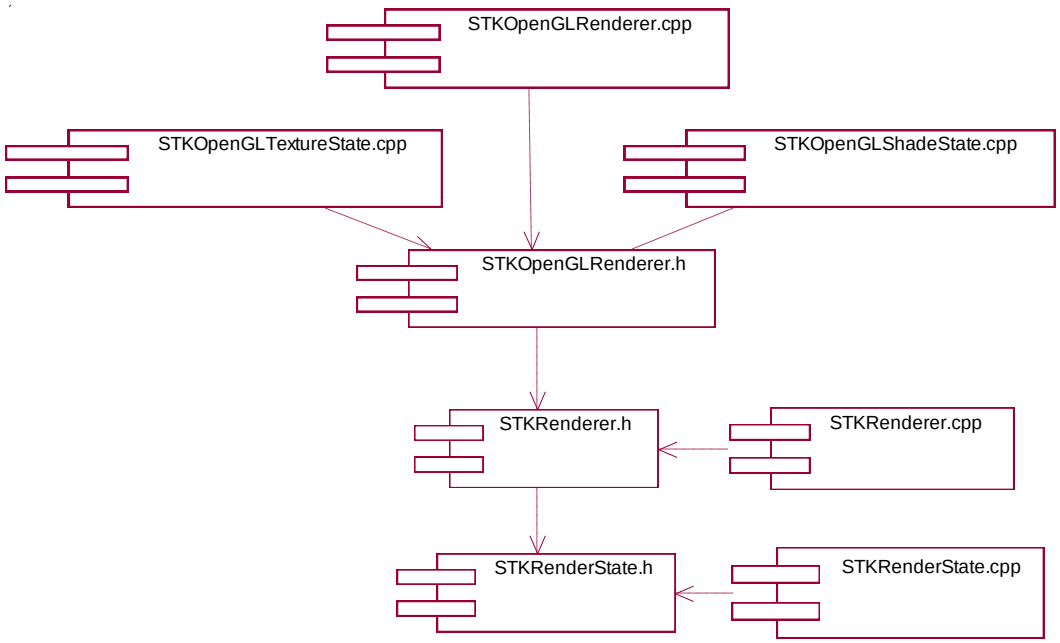


Figura 48: Diagrama de Componentes, Parte III.

5.3-Diagrama de Despliegue

Las relaciones físicas finales entre los componentes de hardware y software del sistema son representados a través de un solo nodo, el cual sería una PC.

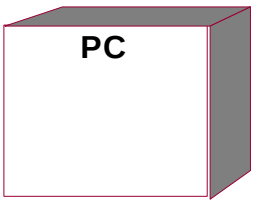


Figura 49: Diagrama de Despliegue.

Conclusiones

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

CONCLUSIONES

Durante la confección de este trabajo se realizó un estudio de los tipos de textura existentes, determinándose el uso de las 2D y las cúbicas. Se analizaron los distintos tipos de filtrado que se aplican a las texturas y se comprobó la necesidad real de utilizar estos para lograr la calidad requerida en las texturas, por lo que en la aplicación final, se aplican varios de estos y se le brinda al usuario la posibilidad de escoger según la necesidad de realismo que requiera y el hardware con que cuente. Además, para lograr mayor calidad en el filtrado se decidió aplicar la técnica de Mipmapping en sus variantes manual y automática.

Después de analizar varios de los efectos visuales que se aplican en la actualidad en los entornos de realidad virtual, se determinó desarrollar los efectos de reflexión y de rugosidad en los objetos, los cuales tienen mucha importancia para lograr el realismo deseado. Para la generación de los mismos se usaron las técnicas “Mapeo de Entorno” y “Bump Mapping” respectivamente, y se implementaron sus algoritmos correspondientes.

Mediante un Modelo de Dominio, se determinaron los principales conceptos en el contexto del problema. Además, se analizó cómo implementar nuevas clases que pudieran acoplarse a la STK reutilizando funcionalidades ya presentes en la misma, con el fin de incorporarle efectos visuales a través de texturas. Finalmente se diseñaron e implementaron las clases.

Como resultado final de este trabajo se lograron efectos visuales con un alto grado de realismo, listos para ser incorporados a la herramienta “Scene Toolkit”.

RECOMENDACIONES

Acoplar las clases desarrolladas a la herramienta (STK), y realizar los cambios necesarios, en algunas de las reutilizadas, para poder incorporarle los efectos visuales implementados.

Agregar nuevos efectos de textura a la STK para continuar incrementando cualitativamente el nivel de realismo de los entornos creados.

REFERENCIA BIBLIOGRÁFICA.

1. Sanz Sanfructuoso, Daniel and Carrasco de Pedro, Oscar. GSII-Página Personal de J.Corchado. *Informática Gráfica.* [Online] [Cited: Diciembre 14, 2007.] http://gsii.usal.es/~corchado/igrafica/.

2. **Calvo, Néstor, Calegaris, Ángel and Sotil, Walter.** WebHome-TWiki. *CIMEC - Computación Gráfica - Unidad VII: Modelado de la terminación superficial.* [Online] Octubre 11, 2007. [Cited: Noviembre 10, 2007.] http://www.cimec.org.ar/twiki/pub/Cimec/ComputacionGrafica/texturas.ppt.pdf.

3. **Hawkins, Kevin and Astle, Dave.** OpenGl Game Programming. *Capítulo 8 y 9.* s.l. : Premier Press, 2001 USA.

4. **Fernández, Marcos.** Dpto. Informática de la Universidad de Valencia. *Ampliación de Informática Gráfica, Tema 7: Técnicas Avanzadas de Modelado.* [Online] Curso 2007-2008. [Cited: Noviembre 16, 2007.] http://informatica.uv.es/iiguia/AIG/web_teoria/tema7.pdf.

5. **Szymczak, Andrzej and Vanderhyde, James.** Welcome to CS 3451A, Computer Graphics. *Spring 2007.* [Online] [Cited: Diciembre 12, 2007.] http://www-static.cc.gatech.edu/classes/AY2007/cs3451_spring/emap.pdf.

6. Wikipedia. [Online] 2006. [Cited: Diciembre 12, 2007.] http://en.wikipedia.org/wiki/Shader.

7. Wikipedia. [Online] [Cited: Mayo 8, 2008.] http://es.wikipedia.org/wiki/Bump_mapping.

8. Wikipedia. [Online] [Cited: Diciembre 12, 2008.] http://es.wikipedia.org/wiki/GPU

9. **Fernando, Randima and Kilgard, Mark J.** Environment Mapping Techniques. *Developer.com.* [Online] [Cited: Noviembre 16, 2007.] http://www.developer.com/lang/other/article.php/10942_2169281_1.

10. **Guinot, Jerome.** oZone3D.net-Tutorial. *oZone3D.net.* [Online] Abril 15, 2006. [Cited: Noviembre 22, 2007.] Addison Wesley Professional.. http://www.ozone3d.net/tutorials/glsl_texturing_p04.php

11. Conocimientoweb.net - La divisa del nuevo milenio. [Online] [Cited: Febrero 5, 2008.] http://www.conocimientosweb.net/portal/term3945.html.

12. Wikipedia. [Online] [Cited: Febrero 8, 2008.] http://es.wikipedia.org/wiki/C++.

13. Wikipedia. [Online] [Cited: Febrero 8, 2008.] http://en.wikipedia.org/wiki/HLSL.

14. Wikipedia. [Online] [Cited: Febrero 8, 2008.] http://en.wikipedia.org/wiki/GLSL.
15. Práctica 1: Texturas y mapeado de entorno usando OpenGL. TÉCNICAS AVANZADAS DE GRÁFICOS 3D, Máster en Informática Gráfica, Juegos y Realidad Virtual.
16. **JeGX, Jérome.** oZone3d.net. *Tutorial: Bump Mapping using GLSL.* [Online] Marzo 8, 2006. [Cited: Mayo 8, 2008.] http://www.ozone3d.net/tutorials/bump_mapping_p2.php.
17. Nvidia. Mapeo cúbico del entorno. [Online] [Cited: Noviembre 22, 2007.] http://www.nvidia.es/object/IO_20011012_6729.html.
18. Perfect Reflections and Specular, Lighting Effects With Cube, Environment Mapping. s.l. : Nvidia.
19. OpenGL Shading Language. GLSL Tutorial. [Online] [Cited: Abril 1, 2008.] http://www.lighthouse3d.com/opengl/glsl/index.php?pipeline.
20. Rodríguez Pérez, Francisco José and Vicente Tocino, Almudena. Informática Gráfica Capítulo 9. GSII - Página Personal de J. M. Corchado. [Online] [Cited: Febrero 5, 2008.] http://gsii.usal.es/~corchado/igrafica/descargas/temas/Tema09.pdf.
21. Starco Studios Blog. Blog de la comunidad virtual de Starco Studios-opengl. [Online] [Cited: Abril 27, 2008.] http://www.starcostudios.com/community/blog/tag/opengl/.
22. Wikipedia. [Online] [Cited: Febrero 5, 2008.] http://es.wikipedia.org/wiki/OpenGL.

BIBLIOGRAFÍA CONSULTADA

1. OpenGL Programming Guide. *The Official Guide to Learning OpenGL, Version 1.1 Chapter 9. Texture* Mapping. [Online] http://www.glprogramming.com/red/index.html.

2. Hawkins, Kevin and Astle, Dave. OpenGl Game Programming. Capítulo 8 y 9. s.l. : Premier Press, 2001 USA.

3. Práctica 1: Texturas y mapeado de entorno usando OpenGL. Técnicas avanzadas de gráficos 3D, Máster en Informática Gráfica, Juegos y Realidad Virtual.

4. **Sanz Sanfructuoso, Daniel and Carrasco de Pedro, Oscar.** GSII-Página Personal de J.Corchado. *Informática Gráfica.* [Online] [Cited: Diciembre 14, 2007.] http://gsii.usal.es/~corchado/igrafica/.

5. Rost, Randi J. OpenGL® Shading Language. USA : Addison Wesley Professional, 2006.

6. Nodarse Valdés, Yaima and Muguercia Torres, Lien. Módulode Efectos Visuales Para Motores De Realidad Virtual Luces y Sombras Dinámicas. Cuba : UCI, 2007.

7. Camacho Román, Yanoski Rogelio and Jiménez López, Fernando. Biblioteca Gráfica Para Sistemas de. Cuba : CUJAE, 2004.

8. Nvidia. Mapeo cúbico del entorno. [Online] [Cited: Noviembre 22, 2007.] http://www.nvidia.es/object/IO_20011012_6729.html.

9. Práctica 1: Texturas y mapeado de entorno usando OpenGL. TÉCNICAS AVANZADAS DE GRÁFICOS 3D, Máster en Informática Gráfica, Juegos y Realidad Virtual.

10. OpenGL Programming Guide - Red Book. *Capítulo 9.* [Online] [Cited: Febrero 5, 2008.] http://www.glprogramming.com/red/chapter09.html.

11. Virtual, Proyecto Herramientas de Desarrollo para Sistemas de Realidad. SceneToolKit: herramienta básica para desarrollo de sistemas de realidad virtual v2.3. 2007.

APÉNDICES.

Índice de Figuras y Tablas.

Referencia a Imágenes

Referencia a Tablas

Glosario de Abreviaturas

1D: Una dimensión

2D: Dos dimensiones.

3D: Tres dimensiones.

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

BMP: Bitmap o Mapa de Bits

CEM: "Cube Environment Mapping". Mapeo cúbico del entorno.

CG: Lenguaje para escribir shader. Propiedad de Nvidia.

CPU: Unidad Central de Procesamiento.

CU: Caso de Uso

CUS: Caso de Uso del Sistema.

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

GLUT (GLU): OpenGL Utility ToolKit (Herramientas y Utilidades de OpenGL).

GPU: Unidad de Procesamiento Gráfico.

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

HW: Hardware

I: Vector Incidente.

N: Vector Normal.

R: Vector Reflejado.

RF: Requisito Funcional.

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

RV: Realidad Virtual

SEM: "Spherical Environment Mapping". Mapeo esférico del entorno.

SGI: Silicon Graphics Inc

SRV: Sistemas de Realidad Virtual.

TGA: Formato de una imagen.

Glosario de Términos

A

Aliasing: Es el dentado o irregularidad que aparece en las líneas debido a su representación por pixel, fundamentalmente las que están casi verticales u horizontales.

Antialiasing: técnicas que se utilizan para reducir el efecto de *aliasing.*

Aplicaciones: Cada uno de los programas que, una vez ejecutados, permiten trabajar con el ordenador.

Apropiado: Acomodado o proporcionado para el fin a que se destina.

Arista: Borde, esquina, línea que resulta de la intersección de dos superficies.

B

Brillo: Luz o resplandor que refleja o emite un cuerpo.

Bump mapping: Técnica avanzada de mapeo de texturas que consiste en dar un aspecto rugoso a las superficies de los objetos. En la misma lo que se modifican son las normales de la superficie y no la geometría del objeto.

C

Clamping: permite repetir los pixels de los bordes de la textura cuando se referencia fuera de ella.

Complejidad geométrica: La determina la cantidad de polígonos que conforman el objeto.

Contigua: Que está muy cerca de otra cosa, y sin nada igual en medio.

Coordenadas UV: Tipo de mapeo de textura donde a cada pixel del objeto tiene un par de coordenadas UV correspondientes.

Cuadrícula: Conjunto de cuadrados que resultan de cortarse perpendicularmente dos series de rectas paralelas.

Cubemap: es un tipo de textura que está formada por 6 caras correspondientes a cada eje de coordenada en sentido negativo y positivo. Muy usadas para simular reflejos del entrono.

D

DirectX: es una colección de APIs creadas para facilitar tareas relacionadas con la programación de juegos en la plataforma Microsoft Windows.

Distorsionar: Deformar.

E

Efecto de dentado: Aliasing.

Esculpir: Grabar, labrar en hueco o en relieve sobre una superficie dura.

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

F

Factible: Que se puede hacer

Filtro Bilinear: La interpolación se realiza en 2 dimensiones (horizontal y vertical).

Filtro de Magnificación: estos filtros van a indicar cómo calcular el color de un pixel de la imagen cuando los texels ocupan varios pixels.

Filtro de Minimización: indican cómo calcular el color de un pixel de la imagen cuando los texels son más pequeños que los pixels.

Filtro Trilinear: realizan 2 interpolaciones bilineares para cada pixel, correspondientes a los 2 niveles de mipmap más cercanos a la resolución actual con la que se está visualizando el objeto y se promedia el resultado asignándole este al pixel de pantalla.

Fluctuación: Cambio alternativo, oscilación.

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

I

Imagen: representación gráfica plasmada en una superficie.

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

M

Mapeo de entorno: Técnica avanzada de mapeo de texturas que permite simular reflejos en la superficie de un objeto con gran precisión e incluso en tiempo real. Es ideal para objetos convexos.

Mapeo de textura: El mapeo de texturas consiste en aplicar una serie de dibujos o imágenes a las caras de un objeto tridimensional consiguiendo de esta forma elevar su realismo.

Mapeo esférico del Entorno: Es el tipo más simple de mapeo de entorno, esta solo usa una textura 2D para la reflexión, no se logra mucho realismo pues la imagen que se ve es siempre la misma aunque giremos alrededor del objeto reflectante. Presenta problemas como la deformación de la imagen y la incapacidad de reflejar el entorno en tiempo real.

Mapeo Cúbico del Entorno: Es el tipo de mapeo de entorno que proporciona los mejores resultados permitiendo crear reflexiones muy precisas y en tiempo real. Requiere de un cubemap como para la reflexión, es decir 6 texturas cargadas en memoria al mismo tiempo. Se eliminan los problemas de deformación de la imagen del "mapeo esférico".

Mapeo Parabólico del Entorno: Tipo de mapeo de entorno que utiliza una parábola como herramienta matemática para realizar el render.

Matriz: Arreglo de elementos.

Mipmapping: Técnica que elimina los problemas de dentado y oscilación de los colores en las texturas a través del uso de varias imágenes que se intercambian o interpolan según la distancia del objeto al punto de visión. Requiere incrementar ligeramente el espacio de almacenamiento de las texturas en un 33% del tamaño de la textura original.

Módulo: Pieza o conjunto unitario de piezas que se repiten o encajan en una construcción de cualquier tipo.

Modular: Pasar de una tonalidad a otra.

N

Nivel de mipmap: Imágenes usadas en la técnica Mipmapping. Cada una posee una resolución igual a la mitad de la resolución del nivel anterior.

Normalizar: Transformar un vector de modo que su norma sea 1.

O

OpenGL: es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D

P

Pixel: Abreviatura de "picture element". Es la unidad menor de almacenamiento de información de una imagen digital.

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

Pixel Shader: Estado del Pipeline Gráfico que actúa sobre las coordenadas, color, coodenadas textura, etc. de un vértice

Polígono: Figura geométrica plana limitada por segmentos rectos consecutivos no alineados, llamados lados.

R

Realismo: Forma de presentar o concebir la realidad tal como es.

Renderer: (como se usa en este documento): componente del sistema gráfico de la computadora, responsable de dibujar los triángulos de un modelo utilizando la información de estados de dibujo, llamada estados de render.

Rendering: crear en forma automática una imagen de acuerdo al modelo tridimensional que existe en el ordenador.

Rendering pipeline: serie de pasos que se siguen para hacer el rendering.

Renderizar: (ver rendering)

Requerimiento: Necesidad o solicitud.

Rol: papel que trae consigo un grupo de acciones y que formará parte de los atributos de un personaje animado.

S

Scene ToolKit: Herramienta de desarrollo de aplicaciones de realidad virtual.

Shader: Programa o conjunto de instrucciones que pueden ser ejecutadas por la tarjeta Gráfica. Es muy usado en la aplicación de efectos especiales así como para realizar cálculos pesados con la tarjeta gráfica.

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

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.

Texel: Pixel de textura.

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

Texturizar: aplicar de texturas.

V

Vector: cantidad que expresa magnitud y dirección.

Vector incidente: Vector que va desde el punto de visión hasta la superficie del objeto.

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

Vector reflejado: Vector que rebota en la superficie de un objeto, depende el vector incidente y el vector normal a dicha superficie.

Vector unitario: vector cuyas combinación son entre 0 y 1, y su norma es 1.

Vertex Shader: Estado del Pipeline Gráfico que actúa sobre el color de cada pixel.

Viewpoint: Punto de visión

W

Wrapping: repetir la textura completa cuando se referencia fuera de ella.