Universidad Nacional Autónoma de México

Facultad de Ingeniería

Arquitectura de software para control de nivel de detalle aplicado a superficies basadas en mallas de polígonos.

Tesis que para obtener el título de

Ingeniero en Computación

presenta

Rodríguez Bojorge Sebastián

Director de tesis

Ing. Emmanuel Hernández Hernández.

Ciudad Universitaria, México, D. F., Octubre 2007

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Con cariño a mi familia.

Agradecimientos

Con todo cariño a mi familia, que me ha brindado su apoyo, confianza, paciencia y sobretodo amor no sólo durante la elaboración de este trabajo sino día a día, que sin duda alguna me permiten alcanzar mis metas y recibir una valiosa formación, que de otro modo hubiera sido muy difícil.

A mis amigos que brindaron su fiel amistad, que considero como algo de lo más valioso que uno puede llegar a compartir, a todos ustedes saben que realmente se los agradezco.

A los profesores que con su dedicación y enseñanzas contribuyeron en mi formación como ingeniero y que con su ejemplo me impulsan para dar lo mejor de mí.

Al Ing. Emmanuel Hernández Hernández que contribuyó como un excelente profesionista a desarrollar mis conocimientos y habilidades en el área de computación, en especial por ayudarme en el estudio de las gráficas generadas por computadora y por la oportunidad de realizar el presente trabajo.

A todos aquellos que junto conmigo comparten la pasión por el desarrollo mostrando su apoyo, compartiendo su experiencia, su conocimiento y algunas veces también sus dudas.

A Esa Ruoho (Lackluster) por tu interés en este trabajo y por la buena música que compartiste y me acompaño en largas horas de trabajo.

ÍNDICE

Capitulo 3

Capitulo 4

Conclusiones

Apéndice

Referencias

INTRODUCCIÓN

La construcción de una escena tridimensional requiere demasiados recursos tanto en software como hardware, adicionalmente de los aspectos no técnicos como el contenido y diseño que son requeridos para formar una escena sólida. Actualmente el hardware ofrecido en las computadoras personales es bastante poderoso y tienen un excelente poder de cómputo con una buena relación costo/desempeño. Por la parte de software, existen diversas alternativas que permiten la creación de escenas tridimensionales, ya sean soluciones para tiempo real o soluciones de preprocesamiento para escenas con calidad fotorealista. Al proceso de cálculo destinado a generar una imagen a partir de un modelo se le denomina *rendering*, no existe una palabra en español que encierre por completo dicho significado, por lo que nos referiremos a este proceso a lo largo de esta tesis como *rendering*.

El caso que nos interesa para el presente trabajo es gráficos en tiempo real; la elección del software que permita cumplir nuestro objetivo depende de muchos factores: la naturaleza de nuestro objetivo (recorrido virtual, visualización, videojuego…), la plataforma y hardware disponibles, incluso la documentación y los usuarios, entre otros.

Es posible que cuando hablemos de una solución en software ello se traduzca en un desarrollo por completo y no en la utilización de componentes ya desarrollados, o bien en el acoplamiento entre componentes existentes y otros por desarrollar. Adicionalmente podemos considerar si el software elegido será propietario u *open source*.

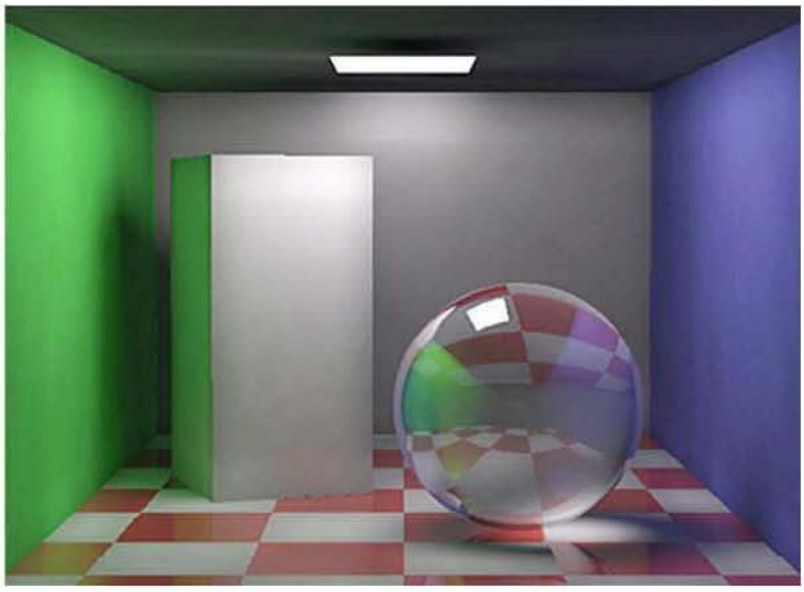
En la investigación precedente a la presente tesis, no se encontró alguna arquitectura que permitiera representar superficies dinámicas basadas en mallas de polígonos con soporte para nivel de detalle. Dentro de las opciones disponibles se encontró software que lleva a cabo *rendering* de superficies con calidad fotorealista pero no en tiempo real y en algunos casos sin soporte de nivel de detalle. Existen soluciones basadas en técnicas de nivel de detalle que permiten la representación de grandes superficies, sin embargo, están limitadas a superficies estáticas (generalmente terrenos).

No habiendo disponible alguna solución que se adaptara a nuestras necesidades, se decidió diseñar una arquitectura que fuese capaz de representar superficies basadas en mallas de polígonos con nivel de detalle y con la posibilidad de adaptarse para la representación de superficies de agua.

El objetivo del presente trabajo pretende lograr el diseño de una nueva arquitectura para la representación de superficies con nivel de detalle, capaz de integrarse en aplicaciones de tiempo real y pueda ser empleado en la representación de diversas superficies, por ejemplo agua.

Capítulo 1

Marco Teórico



"Creemos que las simulaciones gráficas por computadora nunca llegarán a predecir la realidad al menos que modelemos correctamente la física de la reflexión y propagación de la energía de la luz dentro de ambientes físicos."[1]

[1] http://www.graphics.cornell.edu/online/box/ El programa de gráficas por computadora de la Universidad de Cornell es bien conocido por sus investigaciones acerca de rendering basado en física. La imagen superior muestra la caja de Cornell, que se ha convertido en un símbolo del rendering.

1.1 APIs DE GRAFICACIÓN

Actualmente las principales APIs que existen para comunicarse con el hardware gráfico son OpenGL y Direct3D. Ambas APIs cumplen su objetivo satisfactoriamente de manera distinta y con cada nueva versión se incorporan nuevas funcionalidades que hacen uso del nuevo hardware disponible.

1.1.1 OpenGL

OpenGL provee una interfaz para el hardware gráfico. Es una poderosa librería de bajo nivel para modelado y *rendering*, con una amplia gama de hardware soportado. Está diseñada para su uso en cualquier aplicación gráfica, desde juegos de video hasta modelado o diseño asistido por computadora. Bastantes aplicaciones sustentan su sistema gráfico en OpenGL.

OpenGL provee sólo rutinas de bajo nivel, lo que permite al desarrollador tener un excelente control y gran flexibilidad, con ello es posible desarrollar librerías de *rendering* y modelado de más alto nivel.

Originalmente desarrollada por Silicon Graphics, Inc. (SGI) como una librería de propósito general multiplataforma, que a partir de 1992 ha sido supervisada por Architecture Review Board (ARB), conformada por varios líderes de la industria, el objetivo del ARB establecer y mantener las especificaciones de OpenGL que determina cuales características deben de ser incluidas cuando se esta desarrollando una distribución de OpenGL. Los proveedores de hardware gráfico tales como Nvidia® o ATI Inc. por ejemplo, proveen de capacidades específicas en su hardware, estas capacidades adicionales de carácter específico son conocidas como *extensiones*[2], que eventualmente podrían formar parte del estándar de OpenGL.

La arquitectura de OpenGL actúa como una máquina de estados que indican a OpenGL que hacer. Es posible mediante el API establecer las condiciones de iluminación, color, mezclado de colores… cuando se realiza el *rendering* todo esta siendo afectado por las condiciones de la máquina de estados. Por ello es necesario conocer los distintos estados y su condición para evitar resultados inesperados. En el núcleo de OpenGL se encuentra el *rendering pipeline*[3], no es necesario comprender por completo lo que sucede en el *pipeline* para ser capaz de emplear el API, pero si debe uno entender que el resultado final fue consecuencia de varios pasos que afortunadamente son manejados por la misma API.

1.1.2 DirectX

DirectX es un conjunto de APIs que proveen de acceso directo al hardware en sistemas que cuenten con Microsoft Windows®. Cada API tiene un conjunto de funciones de bajo nivel que permiten el acceso a distintas partes del hardware o bien emula el

---

[2] *Ver* en apéndice Extensiones GL

[3] *Redering Pipeline*: Clásicamente se le denomina así a una serie de procesos que terminan por generar una imagen. De primitivas o modelos a escena, posteriormente de escena a imagen (*rendering*).

hardware en caso de que este no exista. Las funciones incluyen soporte para gráficos 2D y 3D con aceleración por hardware, control sobre dispositivos de entrada, control de audio que permiten mezcla y reproducción de audio, control sobre la red y juegos multijugador, adicionalmente permite el control sobre formatos multimedia.

DirectX ha sufrido numerosos cambios tras cada versión disponible, al momento de escribir el presente trabajo la última versión corresponde a DirectX 9.0 con Direct3D10 HLSL (High Level Shader Language)[4].

DirectGraphics es una de las APIs que forman parte de DirectX, responsable dar soporte para gráficos 2D, 3D y emular el hardware en caso de que éste no exista. En versiones anteriores, se hacía una distinción entre Direct3D y DirectDraw (responsable del manejo de gráficos en 2D), ahora ambas están fusionadas en DirectGraphics, aunque esto no presupone ningún problema si la aplicación fue desarrollada en versiones anteriores de DirectX, porque las nuevas versiones de DirectX soportan desarrollos con versiones anteriores.

1.1.3 OpenGL o DirectX

Indistintamente del API que se este empleando es necesario actualizar a la última versión estable disponible; en el caso de DirectX se requiere actualizar a la nueva versión para hacer uso de las últimas características que se incorporan en el hardware, mientras que en el caso de OpenGL será necesario contar con los últimos *drivers,* que proveen de extensiones dando soporte a nuevas características.

Existen numerosas discusiones y foros enfocados a responder a la pregunta más común entre los usuarios de estas APIs ¿Cuál API es mejor? No hay una respuesta sencilla ni determinante a esa pregunta. Son varios los aspectos a evaluar y el objetivo de la aplicación a desarrollar.

El rendimiento desata numerosas discusiones, puesto que es dependiente de la implementación y hardware donde se realicen las pruebas. Existen numerosas pruebas comparativas (*benchmarks*) que tienen por intención evaluar el rendimiento de los equipos haciendo uso de estas dos APIs y en gran medida los resultados son dependientes del hardware, por ejemplo las tarjetas video.

Respecto a la facilidad de uso, la generalidad de los desarrolladores apuntan que OpenGL es más sencillo de aprender y de implementar en sus aplicaciones, sin embargo esto depende de las preferencias y habilidades del programador, por lo que es válido que usted difiera en este punto. Alguna vez le fue sugerido a Microsoft abandonar Direct3D para dar paso libre a OpenGL, posiblemente la crítica más fuerte al respecto fue hecha por John Carmack de ID Software. Afortunadamente las nuevas versiones de DirectX son más sencillas que las anteriores.

Un punto a favor de OpenGL es la portabilidad, DirectX es soportado exclusivamente por Microsoft Windows. OpenGL es soportado por un buen número de plataformas: Linux, Mac OS, Microsoft Windows® y todas las workstation UNIX. Por ello OpenGL permite portar una misma aplicación a distintas plataformas, dejando en desventaja a DirectX. Adicionalmente los ambientes de realidad virtual generalmente no

[4] Para las ultimas referencias acerca de DirectX visite:
http://www.microsoft.com/windows/directx/default.mspx

utilizan Microsoft Windows® como plataforma, haciendo preferente a OpenGL para aplicaciones de realidad virtual. También existe una versión de OpenGL para sistemas embebidos, esta implementación se llama OpenGL ES[5], un ejemplo de estos sistemas es PlayStation3, de Sony.

[5] Para mayor información acerca de las distintas implementaciones de OpenGL visite: http://www.opengl.org/documentation/implementations/

1.2 INGENIERÍA DE SOFTWARE

La meta de la ingeniería de software es ayudar a producir software de calidad, esto es desde el punto de vista de usuarios finales como desarrolladores. Las cualidades deseadas del software podemos dividirlas en dos categorías[1]:

- Externas: El software es rápido, confiable y fácil de usar. A los usuarios finales esto les importa.
- Internas: El código es legible, modular y estructurado. Los desarrolladores dan importancia a esas cualidades.

Antes de que el software se pueda construir, el sistema en el que residirá debe ser comprendido. En vía de conseguir el éxito debe definirse los objetivos generales del sistema; considerar el hardware, el software, las personas en el equipo de trabajo, procedimientos y demás componentes del sistema. Los requerimientos operacionales deben ser identificados; analizados y especificados.

Lo anterior descrito debe ser parte de un proceso, proceso descrito mediante la ingeniería de software. Parte del proceso implica un enfoque para el desarrollo del software, los paradigmas de programación representan el enfoque particular en la construcción del software. Un paradigma de programación es una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan, al final, la estructura de un programa.

Hay muchas formas de enfocar un problema utilizando una solución basada en software. Un enfoque muy utilizado es la visión orientada a objetos. Las cualidades externas son más importantes puesto que la meta de la construcción del software es construir lo que el cliente quiere. Sin embargo, las cualidades internas son la llave para alcanzar las cualidades externas. El diseño orientado a objetos esta pensado para tratar con lo interno, pero el resultado final debe satisfacer las cualidades externas.

La primera vez que se propuso un enfoque orientado a objetos para el desarrollo de software fue a finales de los años sesenta. Sin embargo, las tecnologías de objetos han necesitado casi veinte años para llegar a ser ampliamente usadas. Durante los 90's la ingeniería del software orientada a objetos se convirtió en el paradigma de elección para muchos productores de software y mientras más pasa el tiempo sigue sustituyendo a enfoques clásicos de desarrollo de software.

1.2.1 Programación Orientada a Objetos

La programación orientada a objetos (POO) es un paradigma de programación que permite desarrollar soluciones computacionales utilizando componentes de software (objetos). Un objeto es un componente de software o código que contiene en sí mismo tanto sus características (campos) como sus comportamientos (métodos).

La idea central es simple: organizar programas a imagen y semejanza de la organización de los objetos del mundo real.

Anteriormente se tendía a programar las *funciones* que resolvieran el problema y posteriormente adecuar la forma de organizar la información de modo que se adecuara a las *funciones* ya definidas. Esto es lo que tradicionalmente sucede en un paradigma estructurado, en la POO se definen primeramente los datos antes de definir los métodos.

Lo anterior da lugar a la pregunta ¿Qué es primero, las funciones o los datos? La clave para responder a esta pregunta es el problema de la extensibilidad y en particular del principio de continuidad[1]. Consideremos el ciclo de desarrollo de un sistema, las funciones tienden a ser modificadas un poco (en el mejor de los casos), puesto que los requerimientos del sistema tienden a cambiar regularmente. Sin embargo, los datos sobre los que operan las funciones son más persistentes y cambian generalmente poco. El paradigma *orientado a objetos* se enfoca más en construir módulos basados en objetos que en funciones, es decir, la POO parte del *qué* antes que del *cómo*.

Un buen sistema de software organizado puede ser visto como un modelo operacional de algún aspecto de la realidad. Los objetos simplemente reflejarán objetos del mundo real. Un acercamiento estándar para describir objetos es mediante *estructuras de datos*, *funciones* (operaciones aplicadas), *precondiciones* (deben ser satisfechas antes de que las operaciones sea aplicadas) y *postcondiciones* (deben ser satisfechas después de que las operaciones son aplicadas).

Las tecnologías de objetos llevan a reutilizar los componentes de software existentes, lo que se traduce en un desarrollo de software más rápido y a programas de mejor calidad. El software orientado a objetos es más fácil de mantener debido a que su estructura es inherentemente menos acoplada. Esto lleva a menores efectos colaterales cuando se deben hacer cambios y provoca menos frustración en el ingeniero del software y cliente. Los sistemas orientados a objetos son más fáciles de adaptar y más fácilmente escalables.

Las propiedades más importantes del paradigma *orientado a objetos* son:

- Abstracción
- Encapsulación
- Modularidad
- Herencia
- Polimorfismo

La *abstracción* es la forma en que se hace frente a la complejidad inherente de los problemas que intentamos resolver mediante el software. La abstracción permite distinguir entre las características esenciales de un objeto de las restantes que no lo son. La abstracción es la propiedad más importante dentro del paradigma *orientado a objetos* en lo que respecta al modelado de los entes que abarca nuestro problema, pues es posible definir cualquier ente por complejo que sea. Si bien la idea de escribir programas definiendo una serie de abstracciones no es nueva, sí lo es hacerlo mediante el uso de clases.

La *encapsulación* o *encapsulamiento* es la propiedad que brinda seguridad y limpieza a nuestras clases, mediante la *ocultación de información*. En principio esto es ocultar la implementación de la *interfaz*; donde la implementación son los mecanismos que logran el comportamiento deseado así como la representación de la abstracción, mientras que la *interfaz* es la vista externa de la clase exponiendo sus características esenciales.

La *modularidad* permite subdividir la aplicación en partes más pequeñas (llamadas módulos), donde se pretende que cada una sea tan independiente como sea posible de la aplicación y de los otros módulos. Esto permitiría compilar por separado los distintos módulos, además una mayor independencia entre los módulos evitaría la recopilación de otros módulos además de permitir modificaciones de manera más sencilla puesto que no afectaría al resto de módulos involucrados en el sistema.

La *herencia* es una propiedad que permite jerarquizar nuestras clases. Básicamente la *herencia* define una relación entre clases, en donde una clase comparte la estructura y comportamiento definido en una o más clases (conocido como *herencia simple* o *herencia múltiple*, respectivamente).

La última propiedad aquí citada y no menos importante es llamada *polimorfismo*, algunos autores no la consideran fundamental, sin embargo considero que es una propiedad bastante importante en el modelo *orientado a objetos*, por las posibilidades que permite. En sentido literal, el *polimorfismo* significa que una entidad adopte muchas formas, es decir, se puede referir a objetos de distinta clase mediante el mismo elemento de programa y realizar la misma operación de igual o distinta forma, según sea el objeto que se referencia en ese momento.

El polimorfismo lo considero importante pues potencia el concepto de clase y herencia, elementos esenciales en la *orientación a objetos*. Cuando se genera una clase a partir de una ya existente (*herencia*), se pueden enviar mensajes iguales tanto a las clases derivadas como a la clase base, o bien, permite el envío del mismo mensaje y que la tarea se lleve a cabo de distinta manera según la clase donde se aplique. Esto requiere de *ligadura tardía* y esto sólo aplica en lenguajes de programación orientados a objetos. En POO la dirección de código a ejecutar se determina hasta el momento de ejecución, cuando el mensaje se envía a un objeto, el compilador asegura que la función existe y realiza la verificación de tipos de los argumentos y del valor de retorno, pero no conoce el código exacto a ejecutar.

1.3 ESTRUCTURA DE DATOS

Las computadoras realizan una serie de tareas y procesos en los que son indispensables unidades que se conocen como datos o elementos. Estos datos pueden ser diferenciados como datos simples y datos compuestos.

Los datos compuestos, se conforman de varios datos simples. Los datos simples más empleados son de tipo:

- Entero
- Real
- Carácter
- Booleano

Donde un tipo de dato es un conjunto de valores y un conjunto de operaciones definidas para esos valores.

Generalmente los tipos de datos que vienen definidos en los lenguajes de programación resultan insuficientes para el tratamiento de la información que se desea lograr. Algunos lenguajes de programación tienen características que nos permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el desarrollador se denomina *tipo abstracto de dato* (TAD).[2]

La *abstracción de datos* es la técnica de programación que permite inventar o definir nuevos tipos de datos adecuados a la aplicación que se desea realizar. La abstracción de datos es una técnica muy potente que permite diseñar programas más cortos, legibles y flexibles.

Los tipos abstractos de datos proporcionan numerosos beneficios al desarrollo:

- Permite un modelado del mundo real, mejorando la representación y comprensibilidad. Refina el código basado en estructuras y comportamientos comunes.

- Creación de componentes de software, que permite la extensibilidad del sistema, así como un mantenimiento más sencillo.

- La definición de tipos abstractos de datos permite la comprobación de tipo, evitando errores de tipo durante la compilación, lo que se traduce en un incremento de robustez dentro del sistema.

- Permite una separación importante entre la implementación y la especificación. Con ello es posible modificar y mejorar la implementación sin alterar la interfaz pública del tipo de dato.

Existen numerosas TADs que formalizan la estructura de la información independientemente del tipo concreto de dato que almacenen. Como listas, pilas, colas, listas doblemente ligadas, grafos, árboles, etc. Aun así el desarrollador puede crear cualquier TAD que se adecue a sus necesidades lo cual provee una gran flexibilidad de diseño que aminora la distancia entre el modelo abstracto y el mundo real.

1.4 CONTROL DE MALLAS E INTERPOLACIÓN

Se denominan métodos de interpolación a los métodos que permitan recrear información a partir de un conjunto conocido de datos discretos.

Uno de los principales problemas asociados a los procesos de graficación es la alta demanda de recursos, generalmente se pretende dentro de lo posible simplificar el tratamiento de los datos de modo que se tenga un costo menor en su procesamiento. En otros casos se cuenta solo con información parcial, como producto de una medición o *sampling* (muestreo) y resulta necesario reproducir la totalidad de la información.

Los métodos de interpolación son empleados frecuentemente en tareas de graficación, para solventar los casos antes mencionados. En el primero de ellos, para reducir la complejidad de las funciones que acompañan a la geometría; es posible que se conozca la función sin embargo puede que el costo de evaluación sea elevado por su complejidad. Una alternativa podría ser evaluar dicha función en una serie de puntos y generar una tabla base, después será solo necesario buscar el punto a evaluar en dicha tabla evitando el cálculo y si este valor no se encuentra en la tabla se procede a interpolar para inferir el valor que se busca, muy posiblemente el valor difiera del valor obtenido mediante la evaluación de la función pero el error se compensa con el bajo costo de cálculo y el error puede ser disminuido dependiendo del método de interpolación empleado.

Reconstruir o ajustar con la mayor suavidad posible el conjunto discreto de datos con que se dispone puede reflejarse en superficies más suaves, efecto deseado frecuentemente en graficación.

1.4.1 Métodos de Interpolación

Lineal

Es el más simple de los métodos. La interpolación lineal solamente considera dos puntos (x_a, y_a) y (x_b, y_b) de la siguiente manera:

$$y = y_a + \frac{(x - x_a)(y_b - y_a)}{(x_b - x_a)}$$

Si bien es un método que carece de exactitud, es el más simple y rápido. Otra desventaja asociada a este método es que los datos obtenidos no son diferenciables. Si se desea mayor exactitud pueden emplearse polinomios de mayor grado.

Polinomial

Es un método más costoso que la interpolación lineal, sin embargo reduce el error, produce curvas más suaves pues considera un mayor número de puntos para calcular el nuevo valor. Para la interpolación se emplea un polinomio de grado *n-1* puntos, es decir, se puede encontrar un polinomio de grado *n-1* tal que se ajuste a los *n* puntos que se tienen.

$$f(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + a_{n-3}x^{n-3} + a_{n-4}x^{n-4} \ldots + a_0$$

Adicionalmente al costo de cómputo que implica el polinomio, se ve comprometida la exactitud en los puntos extremos (puntos finales e iniciales de la curva) que lo definen, aunque esto puede ser solucionado mediante el uso de *splines*.

Spline

Las *splines* son un tipo especial de funciones, definidas por un conjunto de polinomios que son definidos en intervalos. Se prefieren sobre los polinomios pues además de emplear polinomios de menor grado, disminuye el error en los puntos extremos.

Resultan bastante útiles en la representación de curvas por la simplicidad de su construcción, su exactitud y evaluación, por ello es común su presencia en el campo de diseño asistido por computadora pues tienen la capacidad de ajustarse a complejas curvas, permitiendo interactividad en su diseño.

Como en los anteriores casos de interpolación, las *splines* también poseen errores de exactitud con respecto a la curva original, sin embargo la interpolación es muy suave y el error es más pequeño que en la interpolación lineal o polinomial.

El siguiente es un ejemplo de una *spline* cúbica para lograr la interpolación en un intervalo de [-1,3]

$$f(x) = \begin{cases} -0.145x^3 - 0.7599x + 0.5267 & x \in [-1,0] \\ -0.315x^3 + 0.641x^2 - 0.7599x + 0.5267 & x \in [0,1] \\ 0.615x^3 - 0.014x^2 + 0.3599x - 0.4342 & x \in [1,2] \\ 0.062x^3 - 0.714x^2 + 0.7599x - 0.2593 & x \in [2,3] \end{cases}$$

Trigonométrica

La interpolación trigonométrica hace uso de de polinomios basados en funciones trigonométricas, aunque su uso es más particular dependiente del tipo de información que se este tratando es ampliamente usado. Un ejemplo particular de interpolación trigonométrica es Fourier.

1.4.2 Control de mallas

Existen diversas formas de construir superficies, que en este caso es lo que nos ocupa. El concepto de spline puede ser llevado a la práctica en la construcción de superficies, mediante NURBS, las NURBS (Non-Uniform, Rational B-Spline), es un modelo matemático que nos permite la construcción tanto de curvas como superficies. Las superficies de NURBS engloban diversas superficies logradas mediante distintos tipos de splines, es por ello que en las herramientas de modelado se emplean NURBS.

La tendencia entre muchos desarrolladores es emplear herramientas de modelado, para después exportar una malla poligonal a la aplicación. Hay que considerar siempre que sea posible la posibilidad de emplear NURBS en nuestras aplicaciones gráficas. En la mayoría de los casos, será necesaria la creación de los polígonos antes de *renderizar*, sin embargo el uso de NURBS posibilita la generación de diferentes niveles de detalle. Así mismo el soporte de NURBS en el hardware se esta volviendo más frecuente[5].

La representación de superficies mediante NURBS permite tener más control sobre la superficie en la forma de pesos para los puntos de control. Un sistema de NURBS puede requerir gastos indirectos adicionales de cómputo, sin embargo si la geometría se mantiene estática puede no ser un problema.

Una ventaja importante de las NURBS sobre el algún modelo polígonal tradicional es que los formatos matemáticos son inherentes más compactos. Es decir, es razonable pensar que el modelo generador es menor que el modelo finalmente generado por el mismo; de hecho en el envío de información por red sería preferente enviar el modelo generador de la geometría que la geometría propiamente.

Podría perder valor la anterior argumentación, considerando los medios de almacenamiento actuales o las posibilidades que permite el ancho de banda. Sin embargo las representaciones matemáticas tales como las NURBS no están atadas a un nivel de detalle en particular. Esto significa que numerosos niveles de detalle pueden ser generados con el mismo control de malla NURB. Con ello se crean nuevas posibilidades, no solo sería posible instanciar varios modelos, sino podrían generarse aplicaciones con modelos auto-ajustados a las características técnicas y configuración del equipo final que el usuario estuviera empleando, esto significaría que si el usuario actualizara su hardware no sólo incrementaría el rendimiento sino también los modelos de hecho serían *mejorados* para tener una representación mucho más suave. Con ello sería innecesario contar con diferentes versiones del modelo en un empaquetado de software.[5]

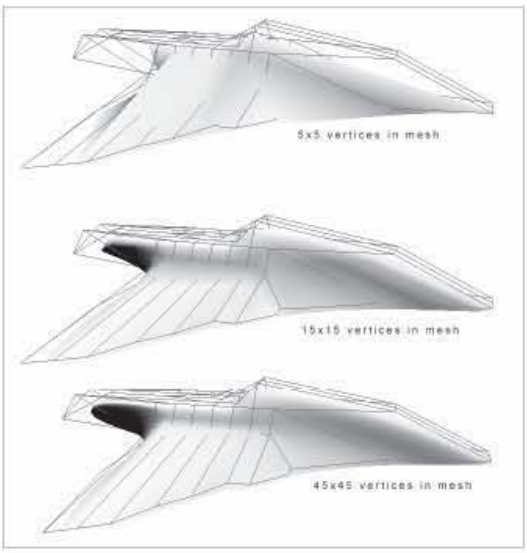


Figura 1.1[6]
Distintas versiones de un mismo modelo basado en las mismas *nurbs*.

[6] Imagen de Focus On Curves and Surfaces[5] “naves para ayer, hoy y mañana”.

Nuevamente podría objetarse la ventaja anterior argumentando que sería posible en algunos casos lograr lo mismo mediante subdivisión de superficies aprovechando las posibilidades que el hardware ofrece para tal objetivo. Pero podrían dificultarse las tareas asociadas a la generación de sombras y detección de colisiones, puesto que sería mediante los vértices y no mediante el modelo que lo define. En cambio las NURBS podrían solventar ambos problemas de muy buen modo, generar geometrías pobres para tareas de colisión, mientras que el mismo modelo matemático podría proveer geometrías detalladas para propósitos de *render*.

Habrá que evaluar los requerimientos y circunstancias que acompañen al problema por resolver y entender las ventajas que cada solución provee para adaptar entre las posibles la mejor.

1.5 CONTROL DE NIVEL DE DETALLE

El nivel de detalle o LOD (por sus siglas en inglés *Level Of Detail*) corresponde a los algoritmos que tienen como objetivo regular el detalle de los modelos en la escena de acuerdo a lo que observa el visor o cámara. El propósito que se persigue al aplicar LOD es mejorar el rendimiento de la aplicación, manteniendo un *frame rate*[7] constante que se traduce en un flujo suave de imágenes.

La regulación del detalle, se logra modificando el número de vértices en escena ya sea incrementándolos o disminuyéndolos. La disminución de vértices es un efecto siempre deseado para tareas de *rendering*, pues lo facilita en términos de procesamiento; sin embargo se pretende que la disminución de vértices o polígonos no deteriore la calidad visual de la escena, por lo que se procura que esta disminución se traduzca en beneficios de procesamiento pero resulte invisible en términos de percepción.

En las últimas décadas se ha trabajado notoriamente en el desarrollo de técnicas de LOD, sin embargo la aplicación de estas técnicas es muy dependiente de la aplicación que haga uso de ellas, por lo que no existe una metodología rígida para llevarse a cabo, cualquier técnica que se traduzca en un mejor rendimiento es válida para este fin.

Bastantes aplicaciones hacen un uso de gráficos en tres dimensiones y algunas de ellas requieren de capacidades de interacción como los recorridos virtuales, aplicaciones CAD (*Computer Aided Design*) y videojuegos por citar algunos ejemplos. Grandes cantidades de información tridimensional es asociada a estas aplicaciones, ello requiere de gran poder de procesamiento para las tareas de *rendering*; existen sistemas especializados para llevar a cabo estas tareas. Sin embargo aun los equipos más poderosos no son capaces de brindar *frame rates* adecuados para interactividad cuando de modelos complejos[8] se trata. Este problema nos obliga a idear técnicas que aminoren la carga de procesamiento, como lo son las técnicas para el control de nivel de detalle.

Si bien la meta en las técnicas de LOD es reducir el número de polígonos a procesar sin con ello sacrificar la calidad visual de la escena, el problema ha sido abordado de distintas maneras. En algunos casos reducir el número de polígonos es traducido en no *renderizar* el objeto en la escena (desaparecerlo), cuando este sobrepasa la distancia máxima establecida.

No es sencillo establecer parámetros que permitan medir con exactitud la calidad de las técnicas LOD. Se tiene como parámetro esencial la mejora en el *frame rate* después de haber aplicado alguna técnica LOD; sin embargo medir la degradación en la calidad visual de la escena no resulta sencillo, pues un *error métrico* menor no siempre se traduce en la mejor aproximación poligonal posible. Es cierto, los algoritmos de LOD pretenden como primera guía ocasionar el menor *error métrico* posible, aunque el resultado es juzgado visualmente. Sería un error grave evaluar las técnicas LOD exclusivamente por su *error métrico*, el *error métrico* pasa a segundo plano cuando en términos prácticos lo que se busca finalmente es lograr la mejor aproximación en términos de apariencia.

---

[7] *Frame rate:* Número de cuadros por segundo. Los dispositivos gráficos deben ser capaces de producir suficientes cuadros por segundo si se desea proporcionar un efecto de movimiento y continuidad, en caso contrario el observador percibirá una simple sucesión de imágenes.

[8] Modelos complejos: Modelos conformados por demasiados polígonos. En el ámbito de los videojuegos, se acostumbra a emplear modelos de *pobre resolución* poligonal, llamado modelado poly-low, con el objetivo de conseguir un buen *frame rate*. Sin embargo, poly-low es un término subjetivo relativo al tiempo, es decir modelos considerados hoy poly-low no siempre han sido considerados de esa manera,

Calificar la mejor aproximación en términos de apariencia se delega al juicio personal a falta de una métrica que estableciera el *error visual* entre el modelo original y el simplificado mediante técnicas LOD. Por tanto el desarrollo de técnicas LOD es una integración de técnicas heurísticas y minimización de *errores métricos*, con la única finalidad de lograr la mejor apariencia. Entonces podemos decir que una buena técnica LOD es la que produzca la mejor aproximación visual al objeto original, acorde a nuestra percepción.

Existen diversas técnicas LOD que podemos agrupar en dos categorías, *nivel de detalle discreto* (DLOD) y *nivel de detalle continuo* (CLOD).

1.5.1 Nivel de Detalle Discreto (DLOD)

Las técnicas de *nivel de detalle discreto* (a veces llamado *nivel de detalle estático*), es un acercamiento simplista a las técnicas LOD.

La idea consiste en tener distintas resoluciones de los modelos 3D (de la mayor a la menor calidad deseada) y seleccionar el modelo con la resolución adecuada en función de la distancia al observador. La idea es básica y funciona generalmente muy bien, a mayor distancia los modelos son definidos por un menor número de píxeles haciendo muy difícil o imposible apreciar los detalles del modelo, aun cuando este consista en miles de polígonos. Podemos sustituir el modelo por uno con una resolución poligonal más pobre sin que con ello el usuario vea afectada su experiencia visual.

La generación de distintas resoluciones del mismo modelo es una tarea generalmente delegada al artista o diseñador, que se encarga de modelar y texturizar las distintas versiones minimizando dentro de lo posible los *artefactos* (imperfecciones) en las versiones con resoluciones más pobres. Actualmente existen herramientas LOD que se encargan de generar las distintas versiones del modelo de manera automática a partir del modelo original (el de mayor *resolución poligonal*), algunos programas de modelado como *LightWave*® o *3D Studio Max*® incluyen herramientas que facilitan esta tarea.

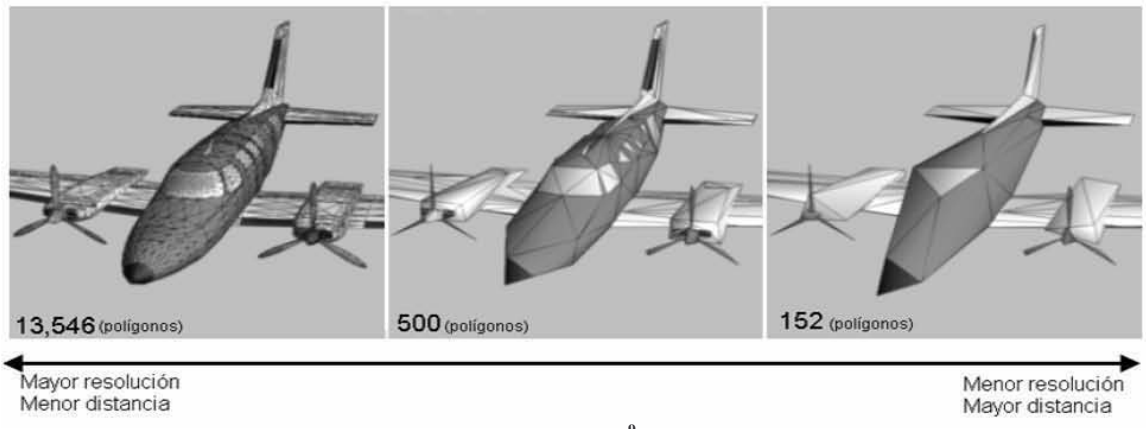


Figura 1.1[9]

Las técnicas de LOD relacionan la distancia del observador al objeto con la resolución del modelo. Las versiones del modelo 3D han sido calculadas mediante el método de mesh progresivve (malla progresiva) de Hugues Hoppe.

[9] Imagen presentada originalmente por Hugues Hoppe, extraída del libro *3D Game Engine Programming* [3]

1.5.2 Nivel de Detalle Continuo (CLOD)

Las técnicas de *nivel de detalle continuo* (a veces llamado *nivel de detalle progresivo*), son técnicas de LOD más complejas. En ellas solo se requiere el modelo con la *mayor resolución poligonal* y las modificaciones de la *resolución poligonal* se llevan a cabo durante la ejecución de la aplicación.

CLOD proporciona transiciones suaves entre las distintas *resoluciones poligonales* (un efecto deseado), pues las transformaciones se hacen dinámicamente sobre una única malla evitando la conmutación entre modelos.

Definitivamente es una técnica más compleja que requiere de un cálculo más intenso, puesto que los cálculos de LOD se llevan a cabo durante la aplicación. Puesto que la simplificación de un modelo 3D no es una tarea trivial, se considera prohibitiva la simplificación de mallas para propósitos de tiempo real, por su costo de procesamiento. Eventualmente esto podría dejar de ser cierto con los avances en hardware.

Para que CLOD sea posible en tiempo real es necesario hacer un preprocesamiento que permitirá tener CLOD en la aplicación basado en los cálculos que se hicieron anteriormente; esto es, determinar los polígonos que serán removidos en la simplificación de la malla, en lugar de construir un modelo simplificado se mantiene la malla original con información adicional acerca del modo de simplificación que permitirá aplicarlo durante la ejecución.

Hay distintas técnicas que permiten lograr la idea anterior, probablemente una de las técnicas más empleadas sea PM[10] (*progressive meshes*) desarrollada por Hugues Hoppe de Microsoft.

PM es una técnica versátil pues se acopla bastante bien a diversas tareas. Además de ser útil para llevar a cabo CLOD durante la ejecución de la aplicación, PM también es empleado en simplificación de mallas, compresión de geometrías y transmisión progresiva de geometría (Al principio, una versión pobre de los componentes de la escena es transmitida, seguido por una posterior transmisión de los detalles).

El algoritmo empleado por PM, es el siguiente: Se toma la malla en el nivel más bajo de resolución, adicionalmente se tiene almacenada información progresiva acerca de cómo reconstruir el modelo para obtener un mejor detalle disminuyendo la diferencia entre este y la malla original. La información de reconstrucción almacena datos acerca de una transformación simple denominada *vertex split* (división de vértice) que se traduce en la creación de nuevos vértices y la división del polígono generando superficies más detalladas. La transformación de *vertex split* se repite una y otra vez en la malla incorporando detalle de una manera progresiva. Adicionalmente si es conocido el número de polígonos finales que se desean en la malla, es posible conocer el número de operaciones que se necesitan realizar para llevar el modelo de bajo detalle al número de polígonos deseado.

Antes de poder aplicar *vertex split* fue necesaria realizar una simplificación durante la creación de la malla (en tiempo de autoría). Cuando se describe la reconstrucción de la malla se hace referencia a *vertex split*, en cambio durante la simplificación de la malla la transformación inversa a *vertex split* se denomina *edge*

[10] Progressive Meshes http://research.microsoft.com/~hoppe/#pm

collapse. *Edge* collapse consiste en tomar una arista y unir los dos vértices que la conforman en uno solo (colapsar), para obtener una versión simplificada de la geometría.

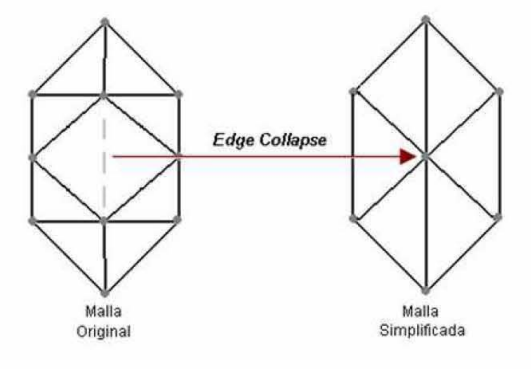


Figura 1.2
Edge Collapse

Se aplicará *edge collapse* tantas veces sea necesario hasta alcanzar el modelo con el detalle más *pobre* deseado. Posteriormente se podrá alcanzar el modelo original aplicando *vertex split* para reconstruir las simplificaciones hechas por medio de *edge collapse* anteriormente.

Dependiendo de los propósitos y del tipo de información que se este tratando, sería necesario conservar información adicional acerca de los vértices simplificados, es posible que se requiera en la reconstrucción de la malla conocer la normal asociada al vértice así como su color o coordenadas para el mapeo de la textura adicionalmente de su posición, por citar algunas propiedades.

La simplificación mediante *edge collapse* no es un proceso arbitrario. Deberá establecerse alguna manera de aplicar la transformación de modo tal que disminuya el *error métrico*, y se obtenga el "mejor" pobre modelo posible, a partir de cual se podrá empezar la reconstrucción[10].

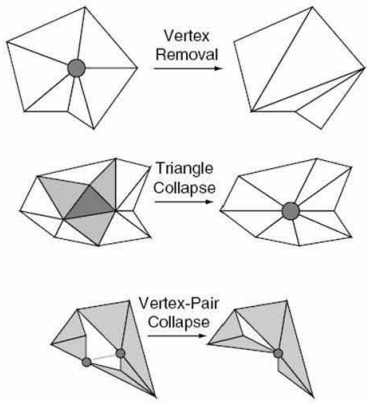


Figura 1.3
Operadores de simplificación local[11]

[11] Imagen obtenida de The Visualization Handbook[13].

Existen diversas transformaciones adicionales a *edge collapse*, que pueden ser implementadas en la tarea de simplificación (figura 1.4)[13].

Adicionalmente se recomienda una métrica compuesta de varios parámetros que permitan lograr una métrica más completa, por ejemplo:

$$E(M) = E_{\text{distancia}}(M) + E_{\text{distribución}}(M) + E_{\text{atributos}}(M) + E_{\text{discontinuidad}}(M)$$ [12]

$E_{\text{distancia}}(M)$ Minimiza el cambio de forma de la malla, es una métrica simple e importante, pues confronta la malla simplificada frente a la malla original, en términos de distancia cuadrática. Es una distancia geométrica que involucra todos los vértices con su posición original, no sólo el vértice que se haya modificado directamente.

$E_{\text{distribución}}(M)$ Regulariza el proceso de optimización, en esencia asegura que los vértices simplificados son tomados uniformemente de toda la malla y evita que la simplificación se concentre en una sola área de la malla.

$E_{\text{atributos}}(M)$ Preserva los atributos de la malla, la malla no es definida exclusivamente por la posición de sus vértices, sino también por los atributos de los mismos. La métrica empleada para minimizar el error en los atributos es similar a la empleada en el caso de la distancia.

$E_{\text{discontinuidad}}(M)$ Minimiza la discontinuidad, por que algunos atributos como las texturas, es posible que no sean continuos a lo largo de la malla. Es necesario evitar la simplificación en estos vértices por que estos podrías ocasionar grandes errores visuales.

Una vez definida la métrica, será necesario aplicarla sobre todos los vértices que conforman la malla y seleccionar el vértice que ocasione menor error métrico en su simplificación. Se simplifica el vértice seleccionado y se genera la información para la posterior reconstrucción. Este proceso será aplicado cuantas veces sea necesario hasta alcanzar la reducción deseada.

El proceso de simplificación es laborioso, razón por la que no es posible llevarlo a cabo en tiempo de ejecución. La ventaja de esta técnica es que puede llevarse a cabo mediante preprocesamiento, para después emplear los resultados obtenidos en tiempo real aplicando *vertex split* como se comentó anteriormente. Si bien esta técnica de LOD es compleja y laboriosa, los beneficios en cuanto a rendimiento y calidad en la aplicación son notables.

[12] Métrica descrita en el paper de Hoppe, *Progressive Mesh*

1.5.3 Observaciones de las técnicas LOD

Las ventajas expuestas acerca de CLOD no deben ser entendidas como un sustituto o desplazamiento de las técnicas DLOD, ambas tienen gran utilidad dentro de las aplicaciones 3D.

Probablemente el punto más endeble en las técnicas LOD, sea al establecer la(s) métrica(s), puesto que el resultado en términos netos es buscar la mejor apariencia, parámetro no considerado explícitamente en la métrica. Hay un par de problemas asociados a un mal diseño de métricas e implementación de las técnicas LOD.

El primero de los problemas debido a una carencia en la lógica de la aplicación de las operaciones propias de LOD. En términos simples propongamos incremento y decremento de vértices como operaciones básicas de LOD. Indistintamente sea CLOD o DLOD debemos establecer un umbral que evite estar alternando entre incremento y decremento por ciclo de evaluación LOD.

Es decir, un incremento en el nivel de detalle no puede derivar en un decremento necesario para el siguiente ciclo de evaluación del LOD. Esto provocaría una oscilación permanente entre ambos estados, señal de una mala lógica en el diseño de nuestro algoritmo de LOD (figura 1.5).

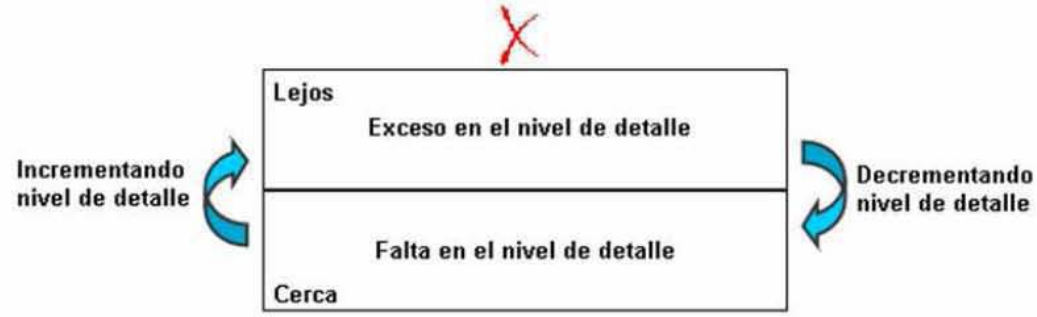


Figura 1.4
Comportamiento no deseado, oscilación permanente entre dos estados

Establecer un límite exacto para decidir acerca de la operación de LOD a tomar, podría derivar en una *oscilación* entre dos estados de LOD, en cambio establecer un umbral permite conseguir un estado definido (figura 1.6).

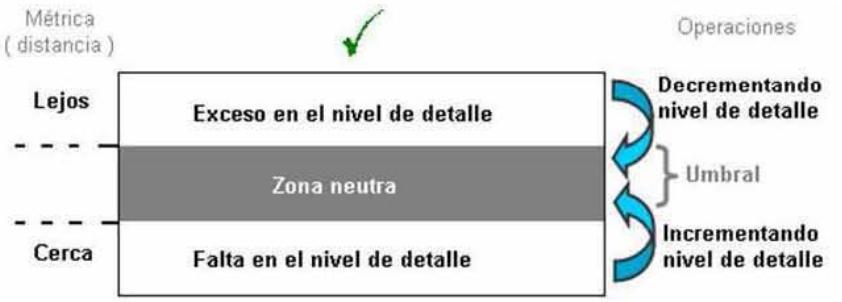


Figura 1.5
Establecer una zona neutra (umbral), permite conseguir un estado *definido* eliminando la oscilación entre las condiciones que definen cada uno de los estados.

Otro problema bastante común asociado a las técnicas LOD es conocido como *popping*, así se le llama a los cambios notables en la malla durante el cambio en el nivel de detalle. Para DLOD esto ocurre cuando se intercambia la totalidad de la resolución de la malla. En el caso de CLOD es notorio cuando se colapsan los vértices o bien cuando existe una división. *Popping* es un efecto no deseado y puede ser evitado estableciendo en la métrica de distancia una menor tolerancia, es decir, efectuar los cambios necesarios en el nivel de detalle aun cuando el observador no se encuentre *muy cerca*, auque siempre hay que recordar que existe un compromiso entre rendimiento y calidad visual.

Las técnicas de LOD son en general demandantes en procesamiento, especialmente CLOD, por lo que se debe prestar especial atención al rendimiento. Es deseado mantener un *frame rate* constante, el *frame rate* es dependiente de las operaciones de LOD y los polígonos en escena entre otras cosas, por lo que se debe decidir cautelosamente cuando las aplicaciones LOD. Si se toman las decisiones de LOD en función del estado de un *frame* es posible tener fuertes fluctuaciones en el *frame rate* cuando sucedan distintos eventos en la escena; por ejemplo, atravesar un portal[13], una explosión (muchas partículas), etc. Por lo que resulta buena idea decidir en función del rendimiento promedio de varios *frames* y no sólo del *frame* actual[4].

Generalmente se sugiere la aplicación de DLOD en modelos que se muestran completamente en escena o modelos *finitos*, por ejemplo personas, edificios, vehículos, animales, etc. Pues el artista o modelador tiene control total sobre la apariencia de los distintos modelos, por lo que se espera una mejor calidad en las distintas versiones que los logrados automáticamente mediante algoritmos, aunque no es una razón de suficiente peso para decidirse por esta técnica. Se obtiene un mejor rendimiento mediante DLOD, pues no se llevan a cabo cálculos de LOD en tiempo real. *Progressive meshes*, es una técnica que permite la aplicación de CLOD en esta clase de modelos.

DLOD resulta insuficiente en la mayoría de los casos cuando se trata de modelos *infinitos,* modelos que no se muestran en su totalidad dentro de escena, ejemplo de ello son los terrenos. En primera instancia resulta conveniente CLOD en esta clase de mallas, por la extensión de su geometría no nos interesa la forma total del modelo (pues no lo *vemos*), sino el mayor detalle en la vecindad del observador y CLOD podría aplicarse exclusivamente al área de interés, sin tener que modificar la geometría que se encuentra fuera del rango de visión.

1.5.4 Recursos adicionales para técnicas LOD

Existen numerosos recursos adicionales que pueden ser incorporados a las técnicas LOD.

Niebla

Uno de los primeros recursos que se emplearon para facilitar las técnicas LOD, fue el uso de niebla. La idea básica es reducir el alcance visual, con ello, los objetos dejaban de ser *renderizados* cuando salían del rango visual, que estaba perfectamente definido por la niebla, o bien disminuye los efectos de *popping*, al disminuir los detalles de los objetos a distancia.

---

[13] Portal: Frontera entre escenarios outdoor/indoor o entre distintos sectores de la escena. Un portal es una manera de optimizar el rendering, porque permite dividir claramente el escenario haciendo menos demandante la tarea de rendering.

Billboard

Otro recurso bastante útil y disminuye notablemente la carga de procesamiento, es el empleo de *billboards*. Un *billboard* es una imagen representada mediante un plano con la normal orientada siempre hacia el observador. En algunos casos es posible sustituir los modelos, por un *billboard* sin que sea notorio por parte del observador. Por ello a veces los *billboards* son conocidos también como impostores.

Bump Mapping

En algunas técnicas LOD, la simplificación de la malla original pudiera traducirse en la pérdida de algunos atributos, como se comentó anteriormente los vértices simplificados pueden tener propiedades adicionales a la posición. En técnicas de LOD más depuradas sería ideal no solo restituir la geometría en lo que posición de vértices respecta sino en los detalles que esta poseía. Es posible compensar la calidad perdida en la simplificación por medio de técnicas de mapeo (como lo son las técnicas *bump-mapping*) y mantener características de la geometría para propósitos de iluminación.

Figura 1.6
Simplificación de la geometría preservando los detalles mediante bump-mapping[14]

Existen algunas herramientas que facilitan la aplicación de esta técnica, haciendo uso de algunas características que las tarjetas de video ofrecen hoy en día. Como ejemplos de herramientas tenemos Melody (Multiple Levels-Of-Detail Extractor) desarrollado por NVIDIA y NormalMapper por ATI Technologies.

[14] Imagen obtenida de Shaders for game programmers and artists[10], una buena referencia para iniciarse en el mundo de los *shaders* con un buen apartado de comentarios acerca de LOD.

1.6 AGUA EN TIEMPO REAL

El agua es uno de los fenómenos más complejos de imitar en el campo de las gráficas por computadora, por sus numerosas propiedades y singular comportamiento al ser un fluido.

Podemos estudiar el agua con distintos enfoques, sin embargo para el caso que nos compete, nos enfocaremos principalmente en su apariencia y parcialmente en su comportamiento hidráulico. Por ello se sugiere como primera aproximación para su modelado en 3D un acercamiento de carácter físico.

Podemos subdividir la representación del agua en dos módulos, ambos correspondientes a la simulación física, el módulo de *animación* y el módulo de *óptica.*

1.6.1 Módulo de animación

Es el responsable de dotar de movimiento al cuerpo de agua en función de distintos parámetros, es aquí donde se incorpora la parte de hidráulica. Estrictamente le corresponde la creación el movimiento en función de las fuerzas internas y externas que afectan el cuerpo de agua. Recrear el movimiento del agua, implica la resolución de ecuaciones diferenciales que representen las leyes de la hidrostática e hidrodinámica; resultado, el agua cambia de forma, se mueve e interactúa con el ambiente.

Pensar en un *modelo de onda* para las ondas que generalmente están presentes en el movimiento del agua, resulta sencillo. Una onda mecánica es una onda que se propaga a través de un material (sólido, líquido o gas), donde la velocidad de la onda depende de las propiedades elásticas e inerciales del medio. Hay dos tipos básicos de movimiento para las ondas de carácter mecánico. Ondas longitudinales y ondas transversales [6].

En las ondas *longitudinales* las partículas tienen una oscilación en la misma dirección de la onda de propagación, mientras que las ondas *transversales* la oscilación de las partículas tiene una dirección perpendicular a la dirección de la onda de propagación. Las ondas del agua involucran el movimiento de ondas *longitudinales* como *transversales*, generando movimientos circulares en las partículas que generan la sensación de transporte en la onda. Los radios de movimiento decrecen en función de la profundidad.

El *modelo de onda* puede ser definido mediante funciones trigonométricas, que resultan simples de calcular. Fourier demostró que cualquier onda puede ser descompuesta como una suma única de funciones sinusoidales[15], basar el *modelo de onda* en funciones de este tipo resulta simple y por ser periódicas son convenientes para propósitos de animación.

[15] Onda cuya magnitud perturbada sigue la ley del seno de una variable.

1.6.2 Módulo de óptica

Parte de una fiel representación del agua debe considerar diversos fenómenos ópticos como refracción, reflexión y dispersión en función de condiciones atmosféricas y espaciales. Los cálculos necesarios para reproducir estos fenómenos ópticos de manera puntual pueden resultar laboriosos, por lo que es necesario adaptar un *modelo óptico* capaz de solventar estas necesidades, siendo responsable de procesar los fenómenos ópticos involucrados.

Los módulos anteriormente definidos corresponden exclusivamente al procesamiento de los datos que el cuerpo de agua involucra, adicionalmente incorporar un *módulo de visualización* para la representación de los datos de manera tridimensional se suma a las tareas que el hardware debe procesar.

Nuestro interés en las mallas persigue la representación de superficies de agua en *tiempo real*. Existen diversas definiciones para *tiempo real*, aunque en algunos casos son controversiales. Donald Gillies[16] define un sistema de tiempo real de la siguiente manera:

Un sistema de tiempo real es aquel en el que para que las operaciones computacionales estén correctas no depende exclusivamente de que la lógica e implementación de los programas computacionales sea correcto, sino también en el tiempo en el que dicha operación entregó su resultado. Si las restricciones de tiempo no son respetadas el sistema se dice que ha fallado.

Por ello habrá que delimitar el alcance de nuestro sistema, pues es notoria la demanda de cálculo de los módulos anteriormente descritos. Si se desea lograr gráficos 3D en tiempo real, será necesario hacer algunas consideraciones y simplificaciones, en la medida que se realicen esas simplificaciones ganaremos rendimiento pero estaremos más distantes de una representación realista.

Una simulación (atiende a las leyes físicas) de los diversos fenómenos mecánicos y ópticos que involucra un cuerpo de agua en tiempo real resulta imposible. Por lo que el presente trabajo se enfoca en la representación gráfica que no atiende estrictamente a las leyes físicas. Las cáusticas son un muy buen ejemplo de este caso, no existen simulaciones gráficas en tiempo real de cáusticas (aún), por lo que habrá que analizar cuál es el objetivo de la representación de las cáusticas para decidirse entre una simulación que se llevará a cabo mediante un *pre-renderizado* o puramente la visualización, que si bien no es una representación fiel de la realidad si logra convencer al observador siendo posible en tiempo real.

[16] http://www.ece.ubc.ca/~gillies/ Donald Gillies (1928-1975) conocido por su trabajo en teoría de juegos y computación.

1.7 CÁUSTICAS

1.7.1 Introducción

Dentro de los principales objetivos en gráficos por computadora está lograr la calidad foto-realista en las imágenes generadas digitalmente, alcanzar tal objetivo es un gran reto pues es necesario tener un buen control de la iluminación en la escena, entre otros factores. Lograr efectos convincentes en gráficos 3D no es tarea sencilla, muchos de los retos que se presentan en computación gráfica son resueltos mediante artificios *simples* que proporcionan la apariencia *deseada* aun cuando los modelos matemáticos en los que se basan no representen la realidad.

La iluminación es un proceso sumamente complicado que requiere de modelos matemáticos *representativos* de la física en el mundo real. La luz está sujeta a varios fenómenos físicos tales como: reflexión, refracción, dispersión, que dependen de las características (material y forma) de los objetos presentes en la escena. Esto sugiere que las imágenes generadas digitalmente que pretendan un efecto de iluminación con apariencia real estén menos alejadas de una *simulación* del fenómeno físico.

En algunos casos es necesario tomar en cuenta las aberraciones (defectos físicos de un sistema óptico debido al cual la imagen no es una reproducción verdadera del objeto)[7], para incluirlas como parte de nuestra composición final. Tanto en calidad como en contenido nuestra imagen digital requiere de ciertos *desperfectos* que logren dificultar el diferenciar una imagen real de una que fue generada digitalmente.

1.7.2 ¿Qué son las cáusticas?

La aberración esférica se origina precisamente por la forma esférica de las lentes, los rayos más distantes al eje se concentran demasiado en forma cercana al lente, mientras los rayos incidentes en la lente se acerquen más al eje se refractaran en menor grado. El punto focal se transforma en una región larga y delgada que se denomina cáustica. Algunos objetos actúan como lentes provocando dicho efecto.[8]

Las cáusticas son una consecuencia del fenómeno de refracción y reflexión, que visualmente deriva en una zona con áreas obscuras y otras luminosas debido a la concentración de *rayos* sobre esa área. Este fenómeno es la explicación de los patrones luminosos que acompañan el fondo de las albercas, objetos de vidrio y/o metal, considerándose un fenómeno de iluminación indirecta (iluminación producto de una desviación).

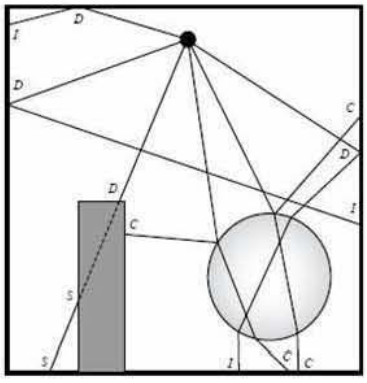
Figura 1.7
Ejemplo de una simulación de cáusticas mediante *Photon Mapping*

1.7.3 Modelando cáusticas

Computacionalmente lograr un *render* con verdaderas cáusticas es muy costoso, se requiere de varios minutos o incluso horas para el *render* de un solo *frame*, lo que coloca a las cáusticas fuera de alcance si se desean en tiempo real. Solo a través de una simulación sería posible obtener cáusticas reales, incluso predecir la apariencia de una cáustica teniendo una fuente de luz y un objeto en la vida real no resulta sencillo.

Existen algoritmos que generan "cáusticas" en tiempo real con apariencia verdadera, sin embargo solo son una imitación al fenómeno real.

Para simular una verdadera cáustica sería necesario tener un gran número de rayos o fotones que representaran la cantidad de luz emitida por la fuente de luz, disparar dichos fotones en todas las direcciones y calcular su trayectoria en función de los objetos presentes, es decir, calcular la refracción y reflexión de cada rayo cuando se encontrase con superficies translúcidas y poner fin a su trayectoria cuando encontrara una superficie opaca. Sería necesario recurrir a la ley de Snell[17] para el cálculo de la refracción y cálculos adicionales si se considera el fenómeno de reflexión. Logrado esto se marcarían las superficies con un mapa de fotones que describa el proceso anterior para después modificar los píxeles que se vieron afectados en dicho proceso. La calidad de la imagen generada radicaría en el número de fotones o rayos empleados en la simulación, y por ser un algoritmo de fuerza bruta resultaría demasiado costoso para el procesador y no se lograría tener una aplicación en tiempo real.



Tipo de fotón	Descripción de trayectoria
D Directo	Provenientes directamente de una fuente de luz.
I Indirecto	Provenientes al menos de una reflexión.
C Cáustico	Provenientes de alguna refracción
S Sombra	Continuación de una trayectoria a través de un objeto opaco.

Cada *colisión* de fotón es guardada en el mapa de fotones.

Figura 1.8
Ejemplo de trayectorias de fotones dando lugar a un mapa de fotones de diversos tipos.

Henrik Wann Jensen[18] propuso una extensión de r*ay tracing*, llamada p*hoton mapping*[9] que es bastante similar a la idea expresada anteriormente en la simulación. Es un modelo de iluminación de dos *pasadas*, la primera pasada construye el mapa de fotones (*photon tracing*), una vez completado el mapa de fotones la segunda pasada hace uso de él para generar la imagen final, haciendo uso de algún método similar a *ray tracing*.

---

[17] La ley de Snell dice que el producto del índice de refracción por el seno del ángulo de incidencia es constante para cualquier rayo de luz incidiendo sobre la superficie que separa dos medios ($n_1 sen\theta_1 = n_2 sen\theta_2$).

[18] Sitio personal de Henrik Wann Jensen http://graphics.ucsd.edu/~henrik/

Hay diversas variaciones al modelo de *photon mapping*, buscando la mejor calidad con el menor costo de cómputo. Pueden generarse por separado dos mapas de fotones (figura 1.9), uno que genere todos los fotones cáusticos y otro mapa para el resto; para la generación del primer mapa se bombardea con orientación exclusivamente a los objetos con características especulares (puede lograrse con *bounding boxes*), la calidad resultante será determinada por el número de fotones utilizados en ambos mapas.

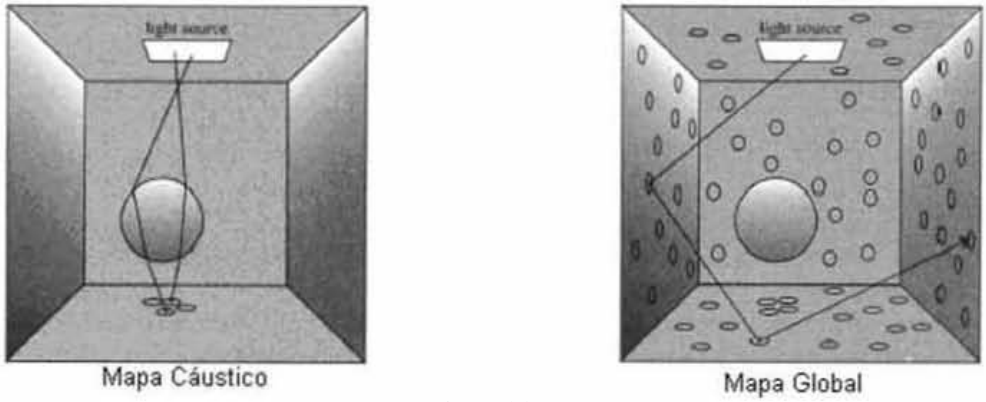


Figura 1.9
Mapas de fotones

La implementación de *photon mapping* no es simple, requiere de varias estructuras y algoritmos para poder ser implementado correctamente; por ejemplo, se sugiere el empleo de árboles KD para almacenar los fotones, en orden de reducir el número de fotones disparados se hace uso de técnicas como *ray tracing* usando la estimación de Monte Carlo. Con 16 rayos por píxel se consiguen efectos fotorealistas mas sin embargo resulta todavía costoso para tiempo real. Menos rayos por píxel estarían distantes de una simulación, pero es un buen inicio para lograr una apariencia y comportamiento cáustico en algunos ambientes en tiempo real, dependiendo de las características y el objetivo a alcanzar.

Finalmente existen otras alternativas menos costosas para imitar cáusticas en tiempo real. El uso de mapas de iluminación (*light mapping*) hace posible aplicar patrones de iluminación a superficies. Pero cabe mencionar que *light mapping* no es propiamente un método de iluminación. Se puede emplear *light mapping* y *radiosity* por citar un ejemplo.

El manejo de mapas de iluminación dinámico es un poco más laborioso, generalmente es empleado en escenas con iluminación estática, pero si es posible lograr la implementación de cáusticas bajo esta técnica. Una textura es la difusa (la imagen) y otra textura causante de las cáusticas (*light map*), se pueden tener varias texturas asociadas a diferentes comportamientos para crear diferentes efectos (por ejemplo *bump mapping*). Desde hace ya varios años es posible aplicar las diferentes texturas en una sola pasada, proceso conocido como *multitexturing*.

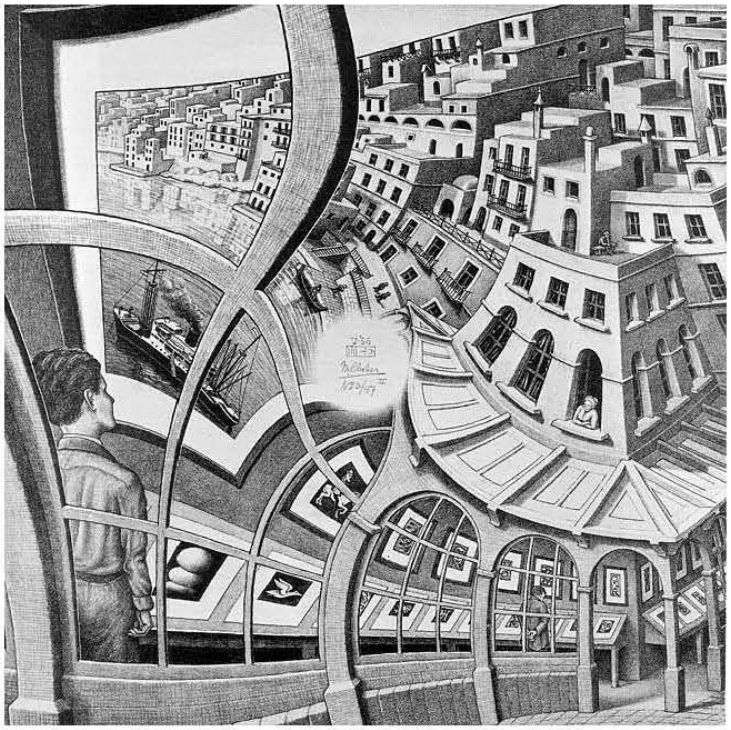
Si el escenario donde es aplicado un mapa de iluminación se modifica sería necesario cargar otro mapa de iluminación lo que pone en manifiesto su desventaja. Generalmente los mapas de iluminación son pequeños o por lo menos más pequeños que las texturas difusas, para ahorrar memoria además de no ser tan necesario contar con

mapas muy grandes. Es posible aplicar filtros para corregir y suavizar los pequeños desperfectos que pudiera ocasionar el mapa de iluminación.

Existen otras alternativas basadas en Píxel Shaders y/o Vertex Shaders, pero es necesario el uso de hardware para su funcionamiento, aunque pueden generarse efectos fotorealistas empleando ambas técnicas.

Capítulo 2

Arquitectura



"Caminar sobre las aguas y desarrollar software según las especificaciones es fácil, siempre que ambas estén congeladas"

Edward V. Berard, "Life-Cycle Approaches"

2.1 CONSIDERACIONES

La arquitectura debe ser lo más modular posible, de modo que permita flexibilidad para modificar, agregar o sustituir funcionalidades no previstas durante su fase de diseño y requerimientos. Posteriormente se desarrollará una librería, como implementación de la arquitectura propuesta.

La librería desarrollada deberá poderse emplear en conjunto con otros elementos para dar vida a aplicaciones 3D en tiempo real, por lo que se ha considerado que los recursos son compartidos y por ello limitados. Es decir, la librería propuesta debe ser capaz de ser ejecutada en equipos con desempeño *promedio* y adicionalmente ser independiente de la plataforma, esto significa que podrá ser empleada en más de una plataforma, se realizarán pruebas en Windows y UNIX.

El objetivo de la librería es satisfacer las necesidades de visualización de superficies en escenarios tridimensionales en tiempo real, de modo que enriquezca la experiencia visual dentro de escenarios virtuales.

Durante el diseño de la arquitectura se ha puesto especial empeño en lograr la mayor independencia posible entre los módulos que la componen, de modo que puedan realizarse futuras modificaciones o adiciones de ser necesario.

En síntesis, la librería pretende ser incorporada en diversas aplicaciones sin afectar gravemente su rendimiento, capaz de ser usada en diversas plataformas y permita modificaciones a los componentes que la conforman o agregar nuevas funcionalidades; por lo que se decidió el desarrollo de la librería bajo las siguientes características que permitan cumplir en mayor grado las características deseadas:

Lenguaje de programación	C/C++
API Gráfica	OpenGL
Plataforma	Windows, UNIX
Diseño	Orientado a Objetos

2.2 DISEÑO

2.2.1 Descripción General

Antes de detallar cada una de las partes que componen la arquitectura, se resume de manera breve la idea que encierra la arquitectura y como funciona, con base en el diagrama de la figura 2.1:

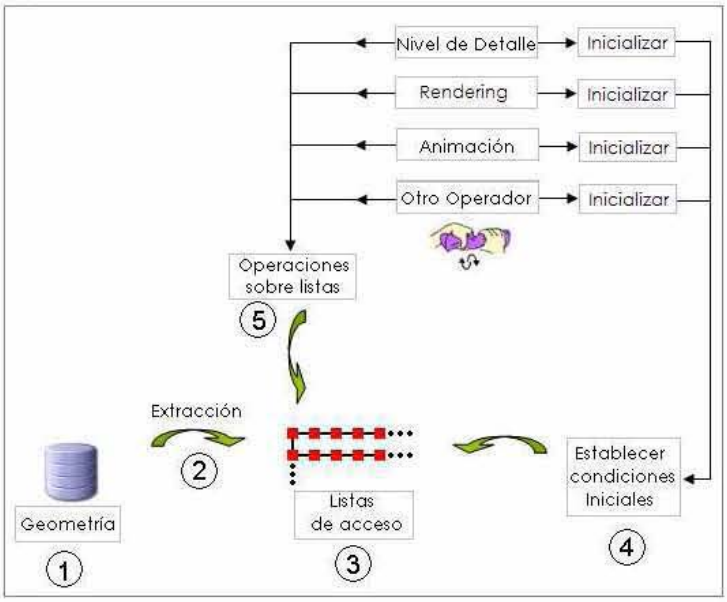


Figura 2.1
Arquitectura

① En primera instancia se tiene la geometría, que en el caso que nos ocupa trata de una superficie plana, una malla. Parte esencial de la geometría son los vértices, sin embargo la geometría comprende poco más que la posición de los vértices, como son las dimensiones, el número de vértices que la componen, el modo en que se unen los vértices y en general cualquier dato adicional que pueda proporcionar información adicional acerca de ella. Cualquier alteración que se desee aplicar sobre la malla se traduce en un acceso a los vértices y modificación de algunas de sus propiedades. El modo de acceder a los vértices depende de la naturaleza de la geometría y no es posible el acceso inmediato a cada vértice sin un índice que los contuviera a todos. *Indexar* los vértices principales que componen la malla permite un acceso inmediato, aunque esto no remedia por completo el problema de acceso si facilita en gran medida el acceso a los principales vértices de la malla.

La geometría aporta información acerca de los vértices pero no contiene de manera explícita información acerca de polígonos, este detalle aunado a la inexistente interfaz que permita un acceso ordenado en este nivel obliga a contar con una nueva forma de acceso para todos los vértices.

② El problema de acceso obliga a extraer los vértices de modo tal que permita un acceso ordenado y rápido. El proceso de extracción consiste en la creación de *listas de acceso* que contengan apuntadores a los vértices, pudiendo extraer solamente los vértices de un área parcial de la malla lo que reduce el costo de procesamiento pues solo se procesan los vértices incluidos en las *listas de acceso* y no la totalidad de la geometría, dicho proceso sigue siendo útil aun cuando la malla original haya realizado operaciones de nivel de detalle (cambio en la topología de la malla).

Terminado el proceso de extracción, se tienen las listas que permiten un acceso más rápido y ordenado a la geometría.

③ Las *listas de acceso* permiten acceder a los vértices de manera más simple. La tarea de este módulo consiste en:

- Administrar las listas de modo que se tenga información consistente entre la geometría y las *listas de acceso*.
- Permitir la aplicación de las operaciones definidas en otros módulos tales como *Rendering y Animación* a los vértices incluidos en las listas.
- Administrar las operaciones solicitadas por el módulo de LOD, que implican incremento y decremento de vértices, que se refleja en numerosas operaciones dentro de las listas.

De esta forma, solo es necesario definir las operaciones sobre la malla y por medio de las *listas de acceso* se aplicarán.

④ Una vez que se tienen los vértices en las *listas de acceso*, es necesario en algunos casos establecer las condiciones iniciales de cada operador antes de aplicar las operaciones sobre las listas. Esto es, modificar las propiedades de los vértices tales como posición, velocidad, etc.

⑤ Para manipular la geometría se definen operaciones que serán aplicadas preferentemente sobre las *listas de acceso*. No todas las operaciones modifican las propiedades de los vértices, en algunos casos solamente se requiere de ellos pero no se altera ninguna propiedad, como es el caso de las operaciones de *Rendering*.

Básicamente esa es la forma en que opera la arquitectura. A continuación se describe cada uno de los módulos que la componen de manera más detallada.

2.2.2 Geometría

Es la capa de más bajo nivel en el modelo, en ella están los datos que definen la geometría. Es decir, en primer término las propiedades de la geometría tales como las dimensiones, seguido de los vértices que la conforman y las reglas de unión entre ellos.

Cualquier modificación realizada en alguna capa superior se refleja de algún modo en la geometría.

Geometría es la capa que se construye primero. En este caso la geometría que nos ocupa es una superficie, a partir de la cual se pueden a manipular los vértices para lograr los efectos deseados sobre la superficie; agua, terrenos, tela, etc. Durante la creación de la malla se definen todos los vértices y las reglas de unión entre ellos.

Fue necesaria la abstracción de la geometría, de modo tal que pueda ser definida una superficie en términos de unidades más pequeñas, cada una conformada por un conjunto de vértices. Estas unidades que denominaremos *mosaicos* de aquí en adelante son un concepto creado para el propósito de esta tesis.

Un *mosaico* es la abstracción de la unidad mínima que conforma a la malla. Una manera análoga de entenderlo, es ver la malla como un piso conformado por mosaicos, la unidad básica del piso sería cada uno de los mosaicos. Es decir, el tamaño mínimo de la malla es un *mosaico*.

Al generarse la malla, cada *mosaico* esta conformado por un vértice central y, por al menos cuatro vértices más, uno a cada esquina del mosaico (figura 2.2).

El concepto de *mosaico* es de suma importancia, pues no solo permite la descripción de una superficie a partir de unidades pequeñas sino también establece las posibilidades en las operaciones de nivel de detalle descritas en el siguiente capítulo.

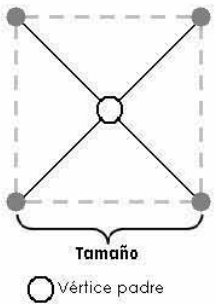


Figura 2.2
Abstracción de un mosaico

Los parámetros básicos en la construcción de la geometría (malla) son:

El número de *mosaicos* en '*x*'
El número de *mosaicos* en '*y*'
El tamaño de *mosaico*

A continuación se muestra una malla definida por los siguientes parámetros:

- Número de *mosaicos* en '*x*': 6
- Número de *mosaicos* en '*y*': 4
- Tamaño de *mosaico*: *s*

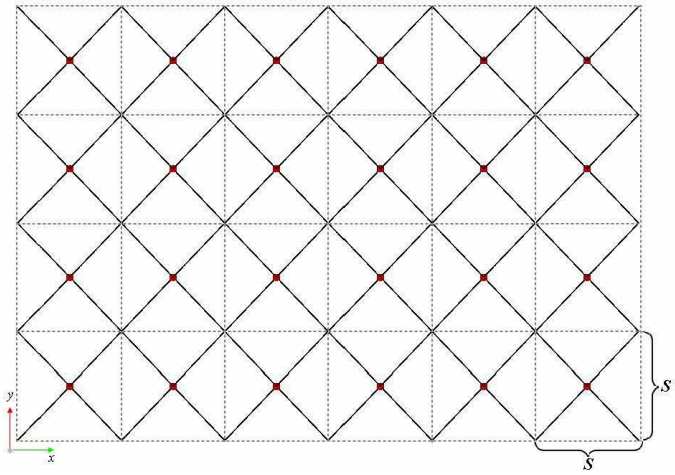


Figura 2.3
Ejemplo de una malla de 6x4 *mosaicos*

La malla generada para este ejemplo mediante esos parámetros tiene un área de *(6s)*(4s)* con un total de *59* vértices. El número inicial de vértices en la malla está definido por un total de *(x+1)(y+1)+(x)(y)* vértices al momento de su generación (esta relación no es válida después de haberse aplicado operaciones de nivel de detalle).

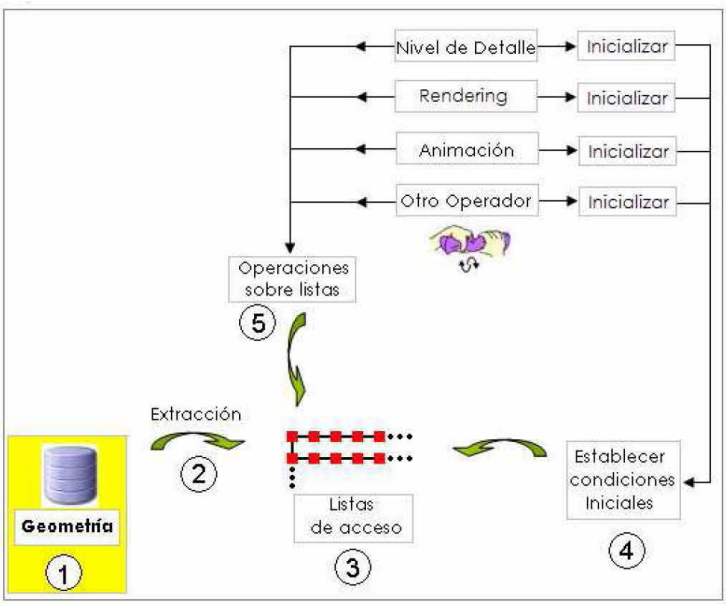


Figura 2.4
La geometría no ofrece una interfaz de acceso ordenado a los vértices

Anteriormente se comentó que la geometría no ofrece ninguna interfaz que permitiera interactuar con los vértices directamente de no ser a través de los apuntadores existentes que permite la comunicación entre vértices (figura 2.5).

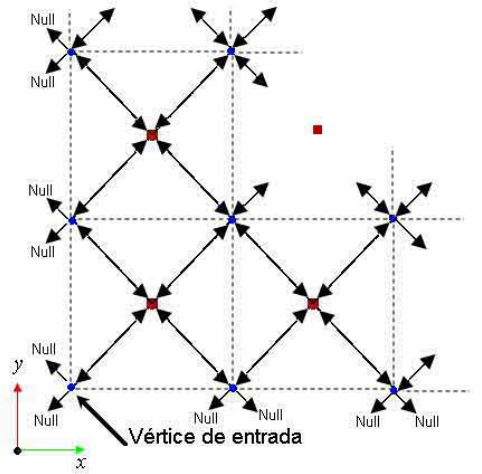


Figura 2.5
Relación de los vértices con sus vecinos por medio de apuntadores (*flechas)*

Contar con una única referencia como *vértice de entrada* implicaría un alto número de accesos alcanzar el vértice deseado y no se realizará un acceso ordenado si no se cuenta con información adicional que indique el orden en que deben visitarse los vértices. Un buen acercamiento para acceder fácilmente a las diversos *mosaicos* es reservar un mayor número de referencias indexando todos los *vértices centrales* de los mosaicos en un arreglo bidimensional (*Index[mosaicoY][mosaicoX]*). A partir de los vértices centrales es posible reducir el número de accesos.

Cuando se crea la malla, se *Indexan* todos los vértices padres de *nivel*[1] *cero* (vértices centrales de cada mosaico figura 2.2), esto permite acceder a los vértices centrales de modo inmediato por medio del arreglo bidimensional.

2.2.3 Listas de acceso

El proceso de extracción constituye una interfaz entre la geometría y las *listas de acceso.*

Construir las *listas de acceso* consiste en crear un nodo por cada *mosaico* en la malla. El *mosaico* es una abstracción y no existe explícitamente en la geometría, al nivel de la geometría es un conjunto de vértices relacionados por apuntadores. El proceso de extracción se encarga de reconocer y agrupar el conjunto de vértices de un *mosaico* para hacer referencia a dicho conjunto a través de un nodo en las *listas de acceso.* Un nodo de las *listas de acceso* consiste en una lista ligada de vértices.

[1] Nivel refiere a la profundidad del vértice dentro del esquema de LOD. Los vértices de nivel cero son los primeros en crearse y son los únicos vértices persistentes durante la existencia de la geometría. Nivel se define más detalladamente en el apartado de LOD.

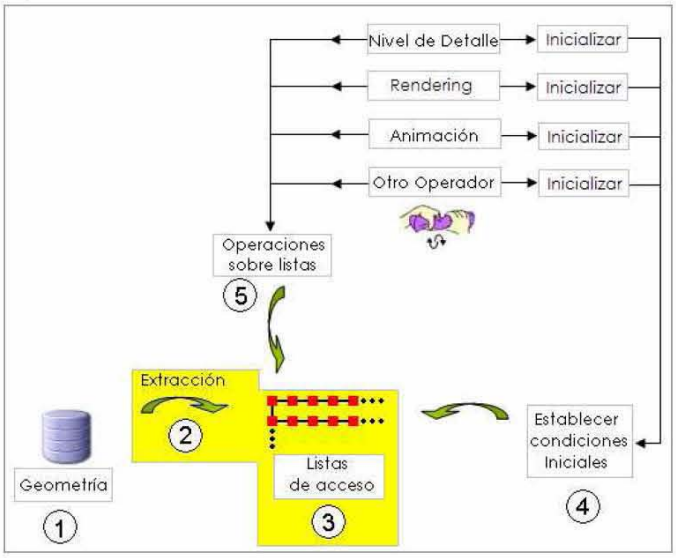


Figura 2.6
Las *listas de acceso* ofrecen una interfaz entre operadores y geometría

Es posible que la malla haya podido ser afectada por operaciones de nivel de detalle, traducido en creación y/o destrucción de vértices, no resultando trivial la identificación de la estructura en la malla cuando esto sucede.

Se generarán tantas *listas de acceso* como renglones de la malla se deseen extraer, cada una de esas listas ligadas tendrá tantos nodos como *mosaicos* se hayan determinado en el renglón respectivo.

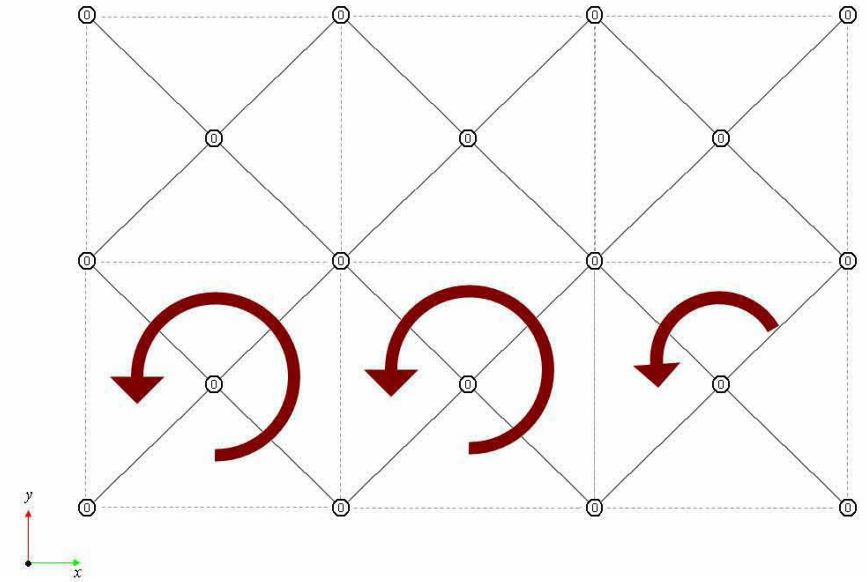


Figura 2.7
Representación de la extracción de los mosaicos

La malla de la figura 2.7 consta de dos renglones, cada uno con tres *mosaicos*, la extracción de cada una de ellos se representa mediante , es decir, para esta figura se está generando el tercer mosaico del primer renglón. Finalmente se tendrían dos *listas de acceso* con tres nodos cada una. Posteriormente se explicará detalladamente el algoritmo empleado en la extracción.

Las *listas de acceso* brindan la interfaz necesaria para interactuar con la geometría, haciendo posible el acceso *mosaico* a *mosaico* a través de cada nodo de la lista, haciendo más sencilla la interacción con los vértices.

Las *listas de acceso* definen un procedimiento que consiste en visitar cada nodo de las *listas de acceso* y aplicarle la o las operaciones que se hayan definido como parámetro, por ello solo es necesario definir las operaciones sobre los vértices y dicho procedimiento se encargará de aplicarlo.

En ocasiones la malla definida puede tener una gran resolución, que resultará en un mayor costo de procesamiento por el número vértices presentes, el proceso de extracción puede reducir dicha carga extrayendo solamente un área parcial de la malla, lo que permite tener una mejora del rendimiento porque solamente se procesarían los vértices contenidos en las listas y no en la malla.

2.2.4 Operaciones sobre los vértices

Se crearon diversos módulos con operaciones definidas sobre los vértices de modo que puedan llevarse a cabo las distintas tareas para poder representar la superficie. No todos los módulos tienen por objetivo modificar las propiedades de los vértices, en algunos casos como el módulo de *Rendering*, se definen operaciones que emplean los vértices pero no los modifican.

Los vértices que conforman un nodo en las *listas de acceso* están ordenados de modo que puedan conformarse polígonos mediante *triangle_fan*[2]. Las operaciones definidas en los módulos son aplicadas de nodo en nodo en las *listas de acceso*, por ello cuando se define una operación debe considerarse que se operará sobre un conjunto de vértices dispuestos en modo de *triangle_fan* (figura 2.7)

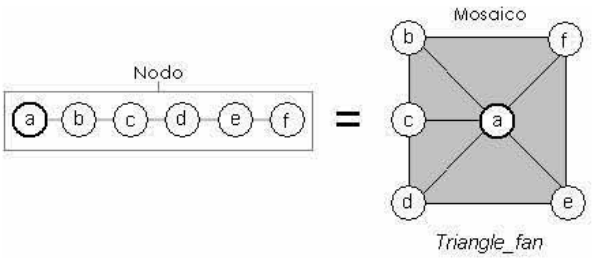


Figura 2.7
Disposición geométrica de los vértices en un nodo de las *listas de acceso*

El primer vértice del nodo es el vértice central, que es vértice común a todos los triángulos, el resto de los vértices se disponen en sentido *anti-horario*.

[2] *Triangle_fan* es una técnica que permite unir múltiples triángulos por un vértice común.

A continuación se muestra un ejemplo simple de una operación, en este caso consiste en llevar a cero la posición de cada vértice del nodo en dirección el eje '*z*'.

```
ToZero(SNode nodo)
{
        SVertex vertex;
        vertex = nodo->vertex;
        while(vertex)
        {
                vertex.positionZ = 0;
                vertex = vertex->nextVertex;
        }
}
```

Código 2.1
Operación definida para los nodos en las *listas de acceso* que modifica la posición de los vértices.

A continuación se explicarán los módulos que actualmente están definidos para la arquitectura, es posible incorporar nuevos módulos dependiendo del propósito de la superficie. De la misma manera, los modelos propuestos en cada módulo son modificables, por lo que siempre serán objeto de mejoras.

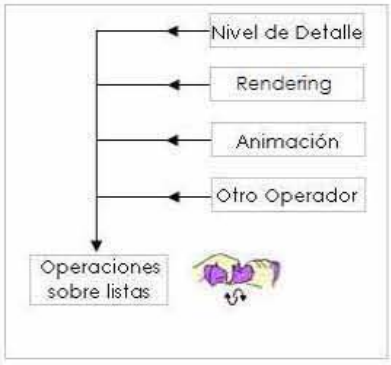


Figura 2.8
Módulos definidos por operadores que interactúan con la geometría por medio de las *listas de acceso*

2.2.5 Animación

En este módulo se definen las operaciones responsables de controlar el movimiento de los vértices.

Los vértices de la geometría poseen un atributo para efectos de animación, el tiempo. La animación está basada en una función paramétrica, cuyo parámetro es el tiempo. Por tanto, el comportamiento esta definido en función de la variable tiempo propia de cada vértice, por lo que dos vértices tendrán el mismo comportamiento si tienen el mismo valor en su atributo tiempo, esto pudiera ser una aseveración falsa si se establece una función periódica.

Cuando se genera la geometría, los vértices poseen un tiempo igual a cero, por ello parte de la inicialización general del sistema, es inicializar el módulo de animación que

consiste en una función que asigne por única vez los valores iniciales de tiempo a cada vértice. Las dos superficies de la figura 2.9, tienen el mismo modelo de animación, aunque difieren en la distribución de tiempo inicial, la primera de las imágenes tiene una distribución de tiempo aleatoria que genera un comportamiento menos uniforme, mientras que la segunda imagen tiene una distribución de tiempo en función de la distancia al centro de la malla, lo que provoca un efecto simétrico y uniforme en todos los vértices que equidistan del centro.

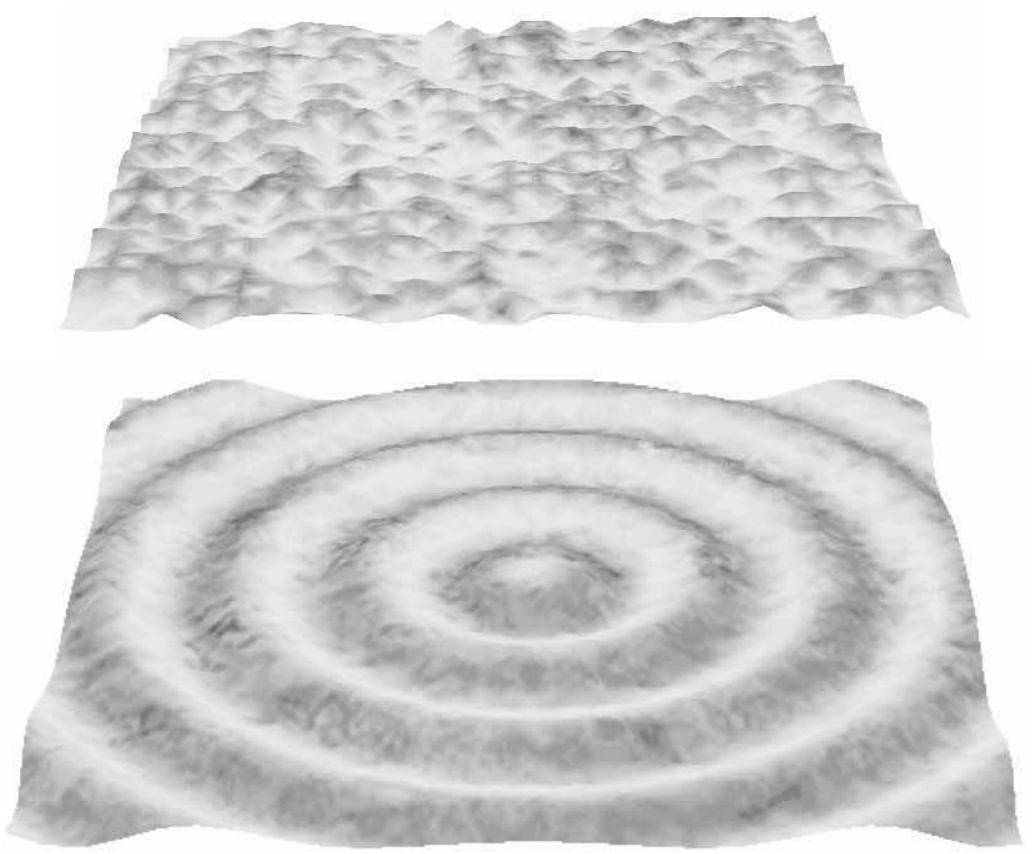


Figura 2.9
La imagen superior muestra una distribución de tiempo aleatoria mientras que en la imagen inferior el tiempo fue distribuido en función de la distancia al centro de la malla.

El módulo de animación define otros factores que afectan el modo en que se comporta el tiempo de cada vértice en las funciones paramétricas y de inicialización.

El efecto de movimiento es consecuencia de incrementar en Δt el tiempo en cada vértice, lo que provoca el efecto de continuidad al evaluarse nuevamente la posición determinada por la función paramétrica. Cuando se desea invertir el *flujo*, Δt es un valor negativo.

2.2.6 Rendering

El módulo de *rendering* se encarga de la visualización de la geometría en 3D y el texturizado (si es requerido).

El proceso de *rendering* pretende imitar el efecto de flujo direccional, reflexión y transparencia (*alpha blending*)[3] mediante operaciones sobre las texturas aprovechando la capacidad de *multitexturing* que ofrece el hardware.

El proceso de *renderizado* se realiza por cada mosaico, el orden de los vértices en cada nodo de las *listas de acceso* facilita la tarea al estar dispuestos en orden de *triangle_fan*, aunque esto no limita a *renderizar* mediante esa técnica. Es posible disponer de los vértices como más convenga y realizar el *rendering* mediante otra técnica.

Adicionalmente el módulo de *rendering* se encarga de reconocer si el hardware tiene soporte para *multitexturing*. El propósito que se persigue al emplear *multitexturing* es lograr un mayor realismo, dinamismo en el proceso de texturizado y proporcionar un mayor control sobre la apariencia, aunque no es indispensable dicha cualidad en el hardware para lograr el *rendering* sí es una cualidad deseada que favorece la estética y posibilita la creación de distintos efectos. Con *multitexturing* es posible aplicar distintas texturas a la vez, disponer de cada textura para distinto propósito dentro del proceso de *rendering* es el propósito de emplearla.

Generalmente el proceso de texturizado consiste en la aplicación de una textura a la vez a cada polígono, aunque es posible aplicar más de una textura a cada polígono en *varias pasadas* (*multipass*). Esto se traduce en mejores efectos visuales permitiendo un *rendering* visualmente más atractivo, sin embargo serán necesarias tantas pasadas por *frame* como texturas se deseen aplicar sobre el polígono, lo que puede disminuye notablemente el *framerate*.

Actualmente es posible lograr el mismo objetivo en una *sola pasada*, realizando una serie de operaciones sobre las texturas, proceso llamado *multitexturing* (multitexturizado).

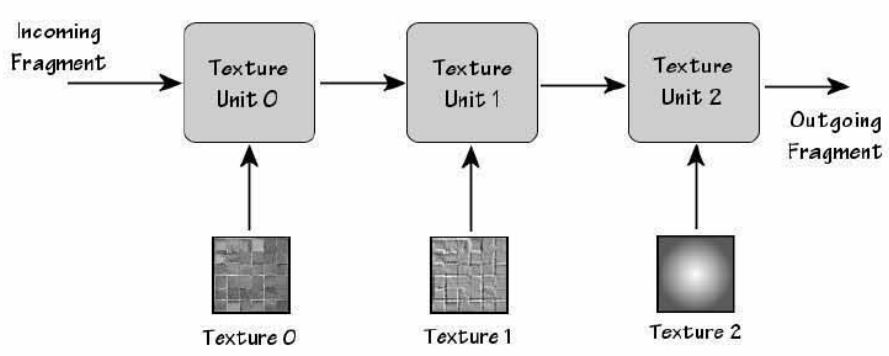


Figura 2.10
Texture Unit Pipeline

[3] Alpha blending brinda la capacidad de crear el efecto de transparencia. Esto es posible cuando se cuenta con cuatro canales por color, tres de ellos son colores primarios. El cuarto canal es conocido como canal *alpha* que especifica la transparencia, es decir, como debiera ser fusionado este color sobre otro ya existente.

Multitexturing hace uso de una serie de texturas, conocidas como *texture units* (unidades de textura). Cada unidad representa una aplicación simple de la textura, cuando se realiza *multitexturing* cada unidad de textura pasa los resultados a la siguiente unidad (figura 2.10).

Pudiera suponerse que este proceso pudiera ser simplificado, mediante el procesamiento previo de las texturas con algún editor de imágenes como Photoshop® de Adobe Software, evitando dicho procesamiento en tiempo real y consiguiendo el mismo resultado en la textura final. Pero *multitexturing* nos posibilita realizar aplicar distintos tratamientos a cada unidad de manera independiente y con ello lograr efectos que no serían posibles con una sola textura como lo propondría un procesamiento previo. Cada unidad de textura tiene un conjunto de estados asociados independiente, es decir, tiene sus propias matrices de textura, sus propiedades de imagen y filtrado, incluso cada unidad pudiera ser distinta tratándose cada textura como unidimensional, bidimensional, tridimensional o *cube map*[4] según fuese el caso. Además se tienen tantas texturas disponibles como combinaciones existentes entre ellas.

Antes de emplear *multitexturing* es necesario comprobar que el equipo que ejecutará la aplicación lo soporta y ofrecer alguna alternativa de rendering en caso contrario. Los equipos que soportan *multitexturing* pueden diferir en el número de unidades de textura que soportan, por lo que también debe verificarse si el número de unidades soportado se acopla a nuestra necesidad.

Para el caso que nos ocupa se emplean dos unidades de textura en el procedimiento de *multitexturing* aunque es posible mediante la librería representar superficies sin soporte de *multitexturing*.

La primera unidad de textura corresponde al mapa difuso, es decir, la textura base de la superficie. Esta textura se mantiene estática y se considera como el color constante de la superficie. En esta textura es posible inducir efectos, como distorsión en las coordenadas de texturizado que brindan elementos para lograr una apariencia más realista.

La segunda unidad de textura corresponde al mapa de ruido y efecto de flujo, es decir, esta textura se modificará en su posición en cada ciclo de animación, lo que brinda un efecto de flujo en la superficie. Es posible modificar la velocidad y dirección de traslado. Es posible conseguir distintos efectos dependiendo de la transparencia y color base dispuestos en la geometría y las transparencias de las texturas en los formatos de imagen que lo soportan (figura 2.11).

[4] Cube-map una colección de 6 texturas, cada una corresponde a una cara del cubo. Se trata de un vector en el centro del cubo apuntando a una dirección, los texels provenientes de la dirección apuntada por el vector son los que se emplearán. El principal uso de cube maps es crear efectos de reflexión en superficies brillantes.

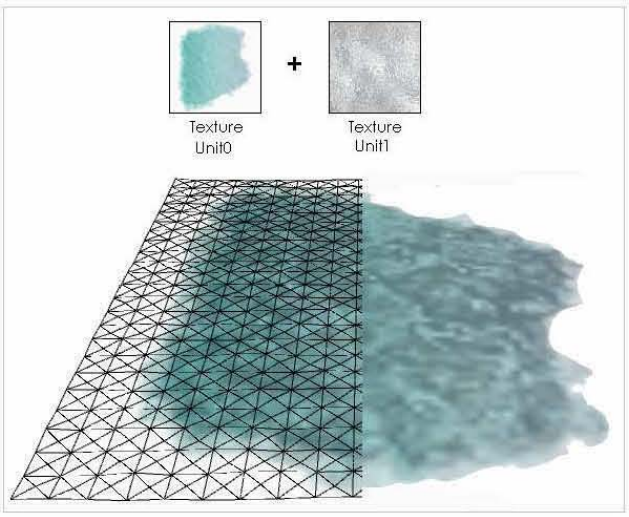


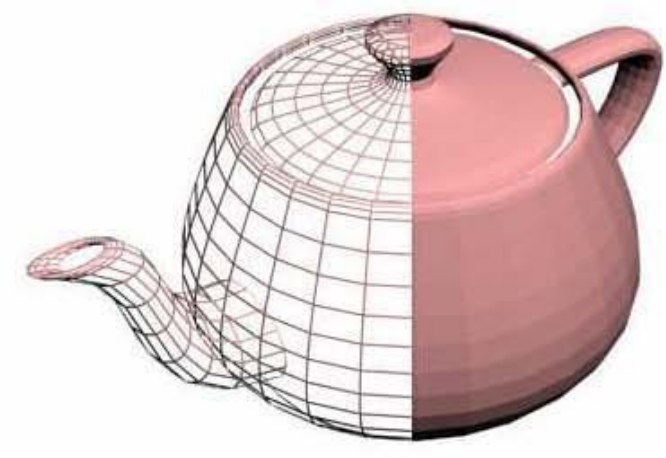
Figura 2.11
Es posible lograr *renders* de superficies no rectangulares y agregar un mapa de *ruido* trasladando el ruido sin alterar la forma de la primera textura, difícilmente se lograría este efecto sin *multitexturing*.

Esta característica es posible en OpenGL mediante extensiones, OpenGL Architectural Review Board (ARB) aprobó esta extensión como parte del core de OpenGL para la versión 1.3, antes de esta versión se consideró como una extensión opcional (también es posible llevar a cabo *multitexturing* con *DirectX*).

Extension name:	ARB_multitexture
Name string:	GL_ARB_multitexture
Function names:	glActiveTextureARB(),glMultiTexCoord{1234}{sifd}[v]ARB()
Tokens:	GL_TEXTUREn_ARB, GL_MAX_TEXTURE_UNITS_ARB

Capítulo 3

Nivel de detalle



La tetera de Utah es icono de las gráficas por computadora. El primer *render* de la ahora famosa tetera data de 1974-1975, usada como símbolo de SigGraph Boston en 1989. Fue el primer objeto de gráficas por computadora en ser diseñado y *renderizado* como una escultura más que como una primitiva o conjunto de polígonos.

3.1 MODELO PROPUESTO DE LOD

El método clásico para modelar alguna geometría tridimensional es mediante el uso de polígonos. Un objeto es aproximado mediante una malla de polígonos conectados entre sí. La mayor parte de las veces es mediante triángulos, esto por simplicidad y generalidad. Es más simple el manejo de triángulos, pues los vértices siempre son coplanares lo que permite facilitar los procesos de *rasterización*[1] y resulta simple conocer la normal del polígono. Además es posible representar polígonos más complejos con base en este. La limitante obvia a los triángulos es que producen una apariencia geométrica plana, aunque hay diversas técnicas de interpolación y suavizado que mejoran notablemente la apariencia.

La técnica LOD (*Level of Detail*) considerada para este trabajo está basada en el concepto de *mosaico* que se explico anteriormente. Los *mosaicos* componen un conjunto de triángulos dispuestos mediante *triangle_fan*. No es forzosa la construcción de mallas o geometrías mediante triángulos ni mucho menos basados en *triangle_fan*, aunque si es muy sugerido el empleo de triángulos en la construcción de geometrías.

3.1.1 Operaciones de LOD sobre la malla

El sistema de LOD ejecuta operaciones sobre la malla, operaciones que tienen por finalidad el *incremento* o *decremento* de vértices. Dichas operaciones están definidas para ser aplicadas sobre vértices *padre* (vértices centrales) de los *mosaicos*. Para dicho propósito cada vértice cuenta con información acerca de su estado LOD (figura 3.2):

- Padre: Bandera que indica si un vértice es *padre* o no.
- Nivel: Indica el número de *incrementos* que fueron necesarios para la creación de dicho vértice.
- Disponibilidad: Se dice que un vértice *padre* se encuentra *no disponible* (falso) cuando la última operación aplicada sobre él corresponde a una operación de *incremento*, en cualquier otro caso el vértice está *disponible* (verdadero). Carece de sentido la bandera de *disponibilidad* cuando no se trate de un vértice *padre*.

Operación de incremento.

La operación de *incremento* tiene por finalidad aumentar el número de polígonos en la geometría. La aplicación de *incremento* sobre un vértice *padre* se traduce en la creación de cuatro nuevos *mosaicos* que sustituyen al *mosaico* original, ocupando un área equivalente (figura 3.3).

[1] Rasterización es el proceso de tomar información descrita vectorialmente y convertirla en una imagen *raster*, es decir, una imagen definida por píxeles.

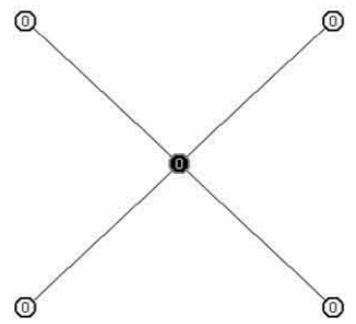


Figura 3.2
⓿ Vértice padre, de nivel cero, disponible.
⓪ Vértice no padre, de nivel cero, carece de sentido hablar acerca de su disponibilidad.

Deben cumplirse ciertas condiciones de modo que pueda llevarse a cabo la operación de *incremento*, si alguna condición no se satisface quedará inalterada la geometría. La operación de *incremento* recibe como parámetro un vértice, para que tenga efecto dicha operación el vértice debe:

- Ser *padre*.
- Estar *disponible*.

Una vez aplicada la operación de *incremento* al vértice *padre* cambia su estado a *no disponible* (falso).

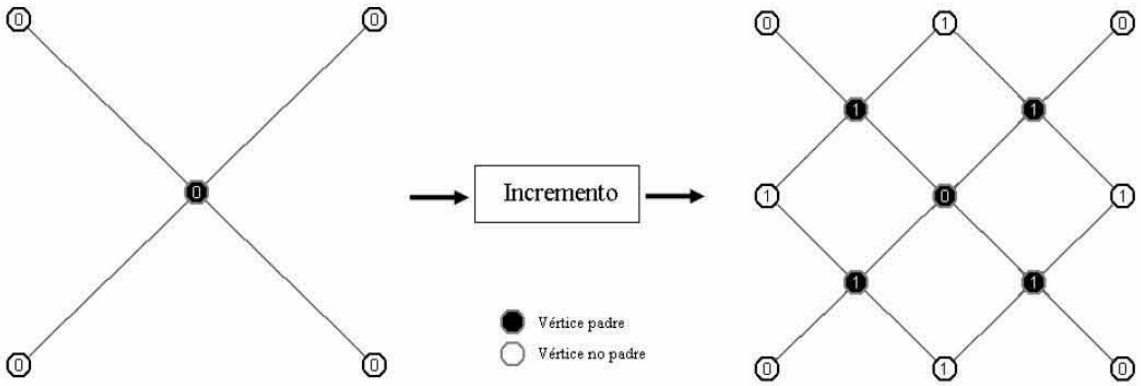


Figura 3.3
Resultado de aplicar una operación de *incremento* sobre el vértice ⓿
Antes del incremento el vértice ⓿ está *disponible*, después del *incremento* (derecha) el vértice ⓿ no está *disponible*.

Es posible aplicar la operación de *incremento* a los nuevos vértices *padre* creados, aumentando aun más el número de vértices en la geometría (figura 3.4), sin embargo es recomendable establecer una condición límite en función de la propiedad *nivel*, de modo que se niegue la operación de *incremento* a vértices que posean el *nivel* establecido como límite máximo.

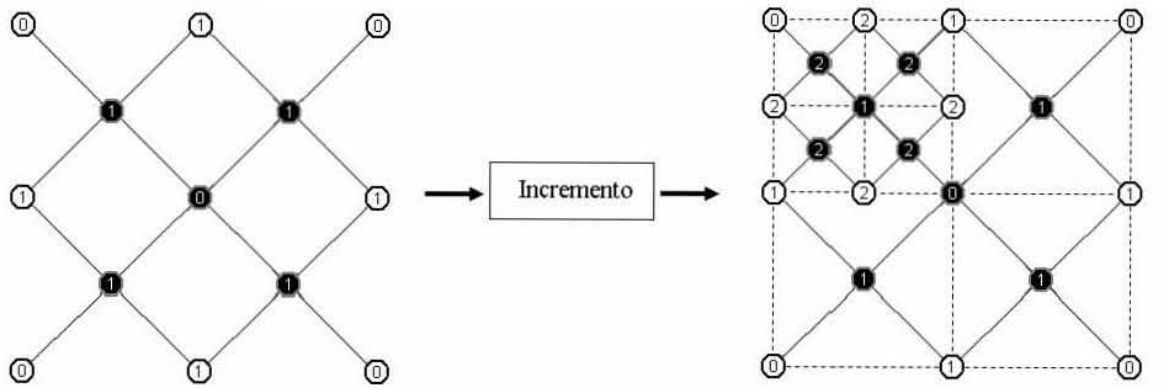


Figura 3.4
Aplicación de la operación de *incremento* sobre uno de los vértices padre ❶

Operación de decremento.

La operación de *decremento* tiene por finalidad disminuir el número de polígonos en la geometría. La aplicación de *decremento* sobre un vértice *padre* se traduce en la simplificación de cuatro *mosaicos* por uno cuya área sea equivalente (figura 3.5).

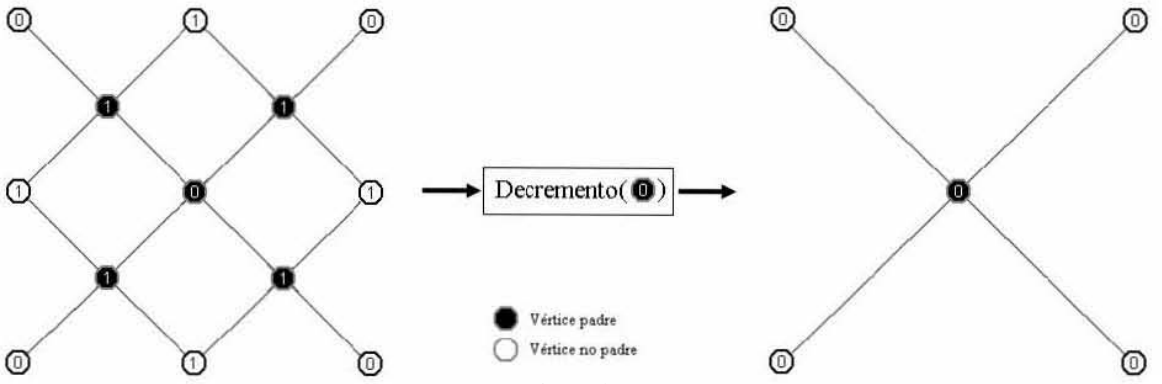


Figura 3.5
Aplicación de la operación *decremento* sobre el vértice ❶

La operación de *decremento* requiere que ciertas condiciones sean cumplidas antes de ser aplicada, en caso contrario la geometría permanecerá inalterada. Es posible aplicar la operación de *decremento* al vértice que cumpla con las siguientes condiciones:

- Ser *padre*.
- Encontrarse *no disponible*.
- Los vértices *padre* correspondientes a cada uno de los cuatro *mosaicos* por reducir deben ser mayores en un nivel al vértice objetivo de la operación *decremento* (figura 3.6).

Una vez aplicada la operación de *decremento* al vértice *padre*, cambia su estado a *disponible* (verdadero).

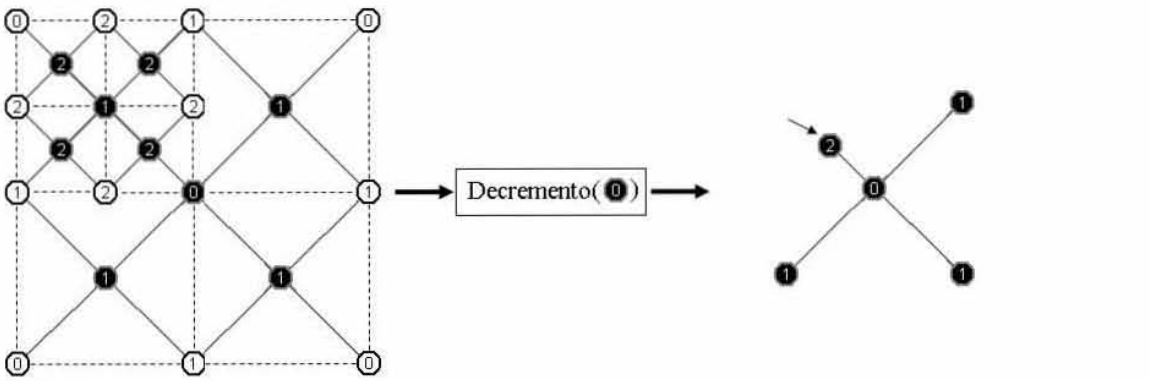


Figura 3.6
La operación de *decremento* no tiene efecto sobre el vértice ❶, dicho vértice tiene diferencia con al menos un vértice *padre* en más de un nivel. El único vértice capaz de efectuar la operación de *decremento* es el vértice *padre* ❶, que apunta a cuatro vértices *padre* ❷

Las operaciones de *incremento* y *decremento* son las operaciones de más bajo nivel que se realizan en el esquema de la técnica LOD propuesta, sin embargo se requieren de tareas adicionales que permitan finalizar de manera exitosa el proceso de LOD.

Cada operación de *incremento* o *decremento* modifica la topología de la malla lo que se asocia a nuevos problemas por resolver. Al modificarse la topología de la malla es necesario reconstruir las *listas de acceso*, de modo que la información contenida en las *listas de acceso* sea consistente con el estado actual de la geometría.

La figura 3.7 muestra los distintos módulos que intervienen en los procesos de LOD.

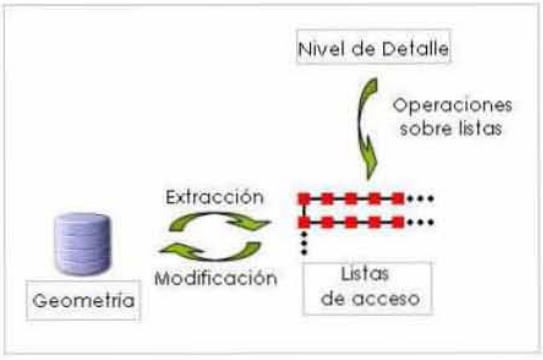


Figura 3.7
Las operaciones de nivel de detalle implican actualizar la *listas de acceso* para que representen las modificaciones aplicadas a la geometría por LOD.

Las operaciones de *incremento* o *decremento* además de crear o destruir vértices obligan a actualizar otros *mosaicos* que se ven afectados por compartir vértices con los *mosaicos* modificados. No considerar un proceso de actualización en dichos *mosaicos* podría traer graves inconsistencias entre las *listas de acceso* y la geometría generando mallas como la que se muestra en la figura 3.8.

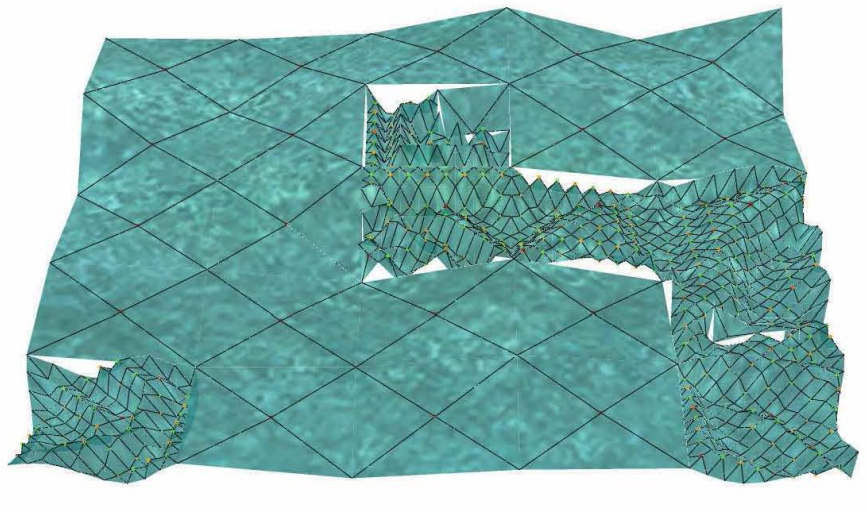


Figura 3.8
Las rupturas presentes en la malla son consecuencia de operaciones de *incremento* sin la correspondiente actualización en los mosaicos vecinos.

Inconsistencias como las presentadas en la figura 3.8 tienen lugar cuando coexisten *mosaicos de diferente nivel*[2]. Recordemos que los vértices *padre* apuntan a tan solo cuatro vértices, cualquier vértice adicional que sea compartido por *mosaicos* de distinto nivel resultará *invisible* en principio para el vértice *padre* de menor nivel (figura 3.9). Aunque no es posible para el vértice *padre* alcanzar directamente cada uno de los vértices involucrados en el *mosaico*, es posible encontrar una ruta de vértices que los conecte (parte del proceso de extracción).

Los *mosaicos* dejan de ser sólo un concepto para convertirse en estructura durante el proceso de extracción de los vértices, dispuestos como nodos en las *listas de acceso*; es en ese proceso donde son incluidos todos los vértices en la construcción de cada nodo y se resuelven situaciones de vértices *invisibles* al vértice *padre* o referencias a vértices que fueron eliminados en el último proceso LOD.

[2] Nivel del *mosaico*: El nivel de un mosaico corresponde al nivel del vértice padre.

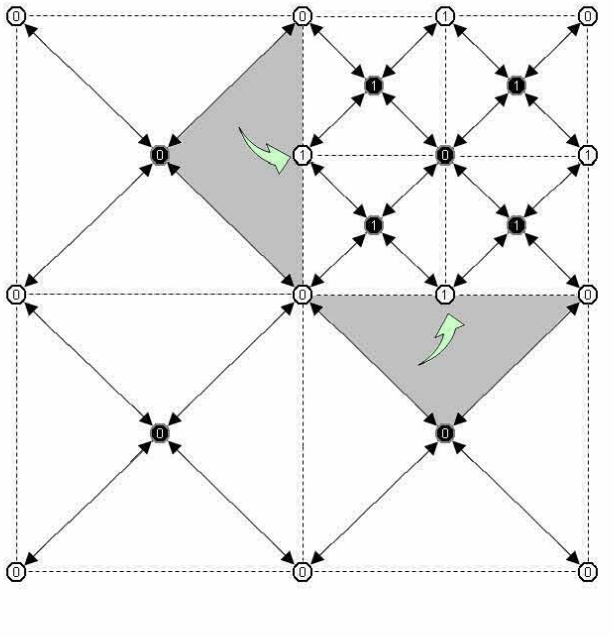


Figura 3.9

Los vértices indicados por  han sido considerados por los vértices *padre* ❶.
Erróneamente la figura sugiere triángulos con cuatro vértices (triángulos sombreados) o vértices *invisibles* al no existir apuntador que los relacione con sus también padres ❶.

```
typedef struct SVertex
{
       //...
       unsigned char  iLevel;
       bool    bAvailable;
       bool    bParent;
       SVertex* aLinks[4];
//aLinks [0]=Superior Izq. [1]=Inferior Izq. [2]=Inferior Der. [3]=Superior Der.
}SVertex;
```

Código 3.1
Estructura vértice con los campos empleados para LOD

3.1.2 Extracción de vértices

El algoritmo de extracción consiste de dos procesos: Recorrer uno a uno todos los vértices *padre disponibles* que componen la malla y construir los nodos (que representan a los *mosaicos*) a partir de éstos. La construcción de un nodo consiste en agrupar los vértices en una lista de modo que sea factible emplearlos en la construcción de los polígonos que componen la geometría. Al finalizar el proceso de extracción se habrán generado las *listas de acceso* (figura 3.10), cada nodo de las *listas de acceso* refiere a un *mosaico* (como se muestra en la figura 3.12<abajo>)

La construcción de un nodo inicia en un vértice *padre*, si este vértice está *disponible* es posible iniciar la recolección de los vértices. `BuildTile` construye un *mosaico* a partir de un vértice *padre disponible* (código 3.2)

El recorrido de la malla consiste en la búsqueda de cada vértice *padre disponible* y aplicarle `BuildTile`. Cuando un vértice padre no está disponible se procederá con sus cuatro *mosaicos* mayores en un nivel, se almacenarán referencias a tres de ellos y se procederá a aplicar `BuildTile` al restante si está *disponible*, en caso contrario, se procederá con sus cuatro *mosaicos* de manera similar nuevamente. Una vez que se ha cumplido la condición de *disponibilidad* en un vértice *padre* para aplicarle `BuildTile`, se procede con las referencias almacenadas en la pila y en el índice. (figura 3.11, código 3.3)

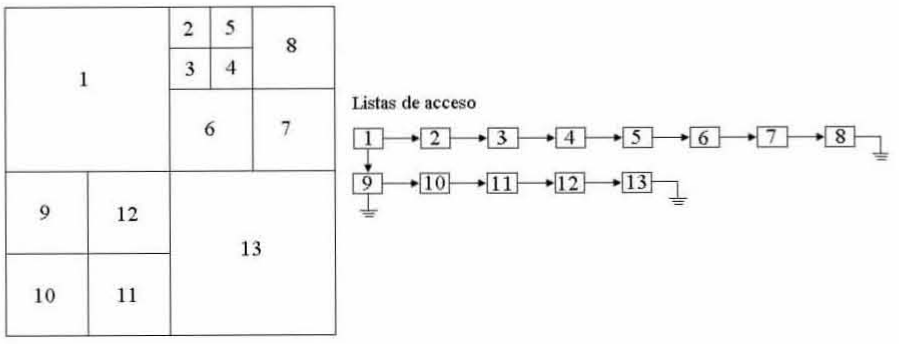


Figura 3.10
El número representa el orden en que se crearon los nodos y su lugar en las *listas de acceso*

```
CNode BuildTile (SVertex* padre)
{
	if(padre->bAvailable == false)//Sólo de vértices disponibles se construyen mosaicos
		return 0;

	SVertex* temp;
	CNode    list;
	unsigned char i;

	list.Add(padre);//Vértice central (padre) se agrega en principio
	i=1;
	temp = padre->aLinks[0];
	while(i<5)// 4 pasadas de recolección de vértices (izquierda, abajo, derecha, arriba)
	{
		list.Add(temp);
		if(temp->aLinks[i%4]!=NULL)//Verificando la existencia de mosaicos vecinos
		{
			temp = temp->aLinks[i%4]
			if(temp->iLevel > padre->iLevel)//Para evitar vértices invisibles
			{
				temp = temp->aLinks[(i+1)%4];
				while( temp->iLevel > padre->iLevel )
				{//Mientras aun no se alcance esquina del mosaico
					list.Add(temp);
					//Zig_Zag
					temp = temp->aLinks[i%4];
					temp = temp->aLinks[(i+1)%4];
				}
			}
		}
		temp = padre->aLinks[i%4];
		i++;
	}

	return list;
}
```

Código 3.2
Construcción de un nodo para la lista de acceso a partir de un vértice *padre* en la malla

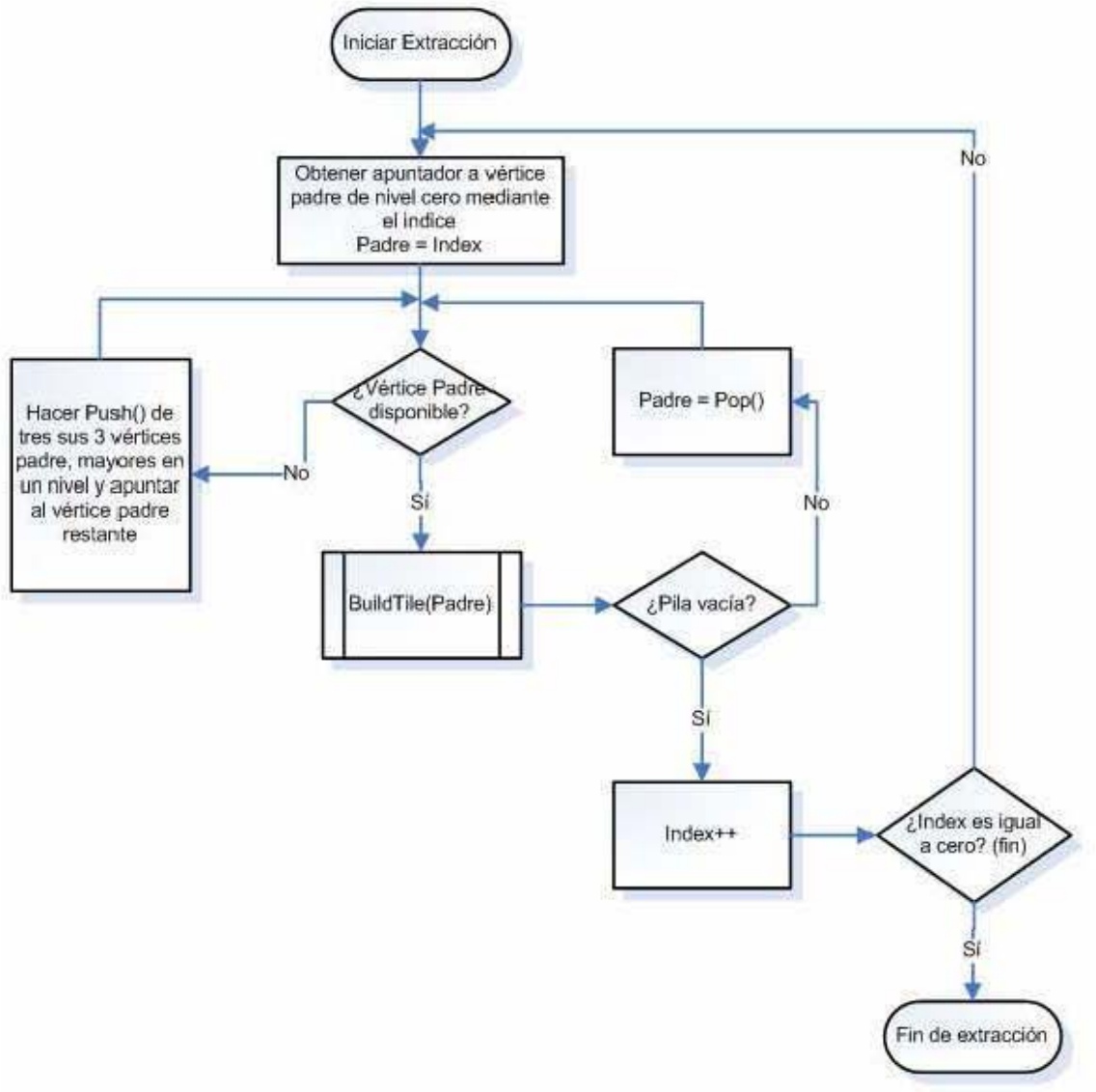


Figura 3.11
Proceso de extracción, recorre todos los *mosaicos*

```
void Extractor(CMesh malla)
{
	CStack stack;
	CLista *list;
	CLista *header;
	CNode tile;
	SVertex* previous;
	SVertex* temp;
	unsigned char level;
	int x=0, y=0;

	temp = malla.Index[0][0]; //Índice de vértices padre de nivel cero
	list = new CLista();
	header = list;//Evita que se pierda la cabecera de la lista
	while(true)
	{
		if(temp->bAvailable)
		{
			tile = BuildTile(temp);//Construcción de un nodo
			list->Add(tile);

			if(stack.IsEmpty() == true)
			{
				x++;
				if(x==malla.SizeX())//Alcanzamos fin de malla en dirección X
				{
					x = 0;
					y++;
					if(y==m_iYEnd)//Alcanzamos fin de malla en dirección Y
					{//Se recobra la cabecera del las listas de acceso
						list = header;
						return; //Fin de malla, fin de extracción
					}
					temp = malla.Index[y][x];
					list->nextList = new CLista();
					list = list->nextList;
					list->nextList = 0;//Fin de listas de acceso
				}
				else
				{
					temp = malla.Index[y][x];//Al siguiente padre
				}
			}
			else
				temp = stack.Pop();//Recobramos un vértice padre por procesar
		}
		else
		{
			previous = temp;
			level = temp->iLevel;
			for(int i=3;i>=1;i--)//Se buscan los padres de siguiente nivel
			{
				do
				{
					temp = temp->aLinks[i];
				}
				while(temp->iLevel!=(level+1));
				stack.Push(temp);//Se guarda la ruta
				temp = previous;
			}
			do
			{//Al siguiente padre de nivel inmediato superior
				temp = temp->aLinks[0];
			}
			while(temp->iLevel!=(level+1));
		}
	}
}
```

Código 3.3
Extracción de los mosaicos para incluirlos en las *listas de acceso*

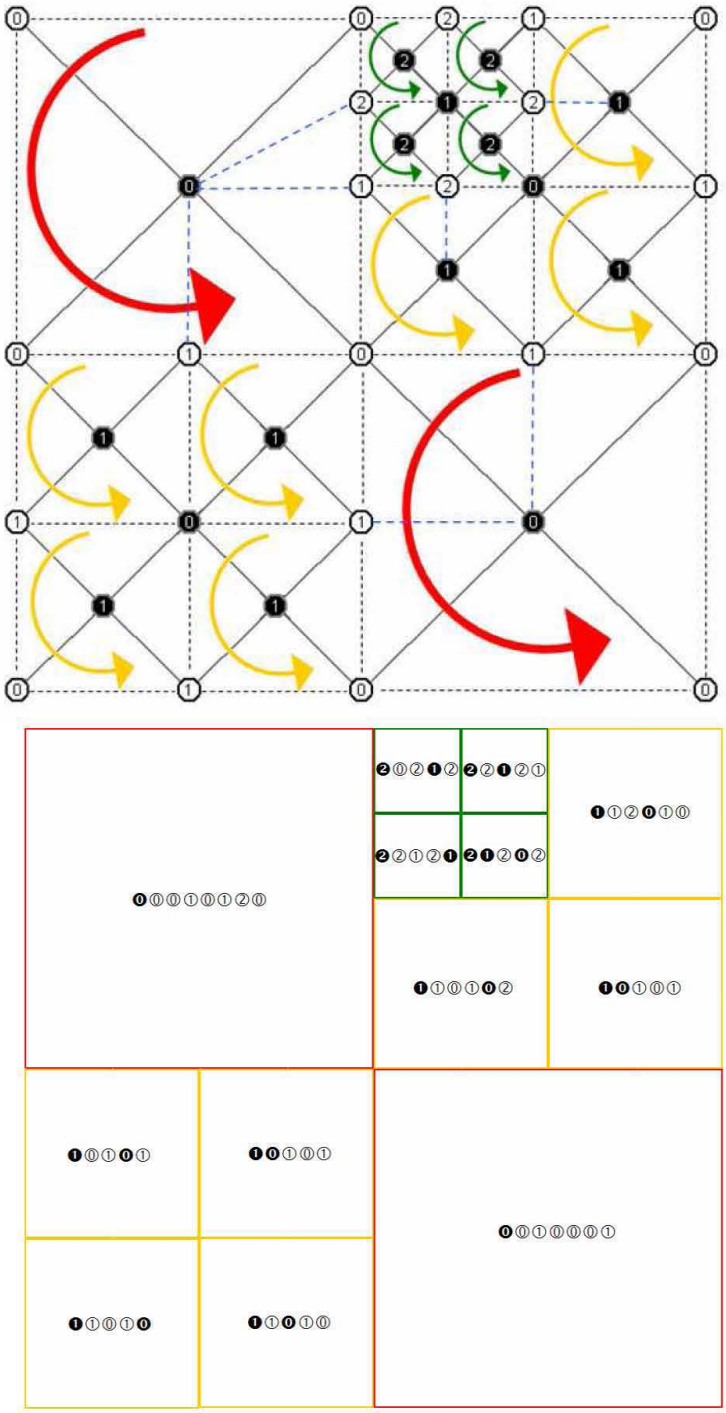


Figura 3.12
<Arriba> Extracción de los mosaicos. Durante la extracción se consideran los todos los vértices, aun los que son *invisibles* al vértice *padre*. El proceso de construcción es representado por las flechas curvas .
<Abajo> Abstracción de la geometría, representados por las listas de vértices generadas durante el proceso de construcción de los nodos, cada conjunto de vértices (dispuestos en modo de *triangle_fan*) corresponde a un nodo en las *listas de acceso*.

3.1.3 Operaciones de LOD en las listas de acceso

Cuando se aplican operaciones de *incremento* o *decremento* (independientemente de la métrica empleada para ello) se verá modificada la topología de la malla existiendo una diferencia entre los *mosaicos* y los nodos en las *listas de acceso*. Por lo que es necesario actualizar las *listas de acceso* como parte de las operaciones de nivel de detalle, para evitar nuevos vértices no referenciados o vértices con apuntadores a localidades de memoria correspondientes a vértices que han sido eliminados. Por ello parte de las operaciones de *incremento* y *decremento* se asocia con identificar los nodos de las *listas de acceso* por actualizar.

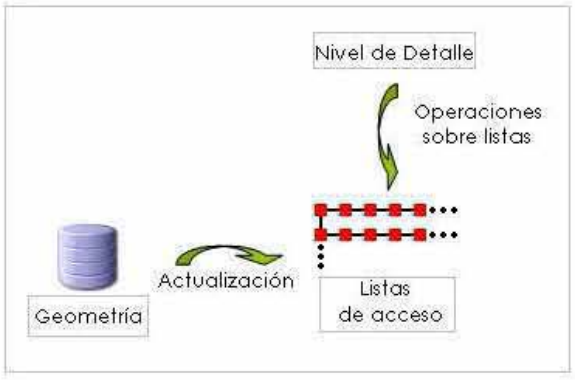


Figura 3.13
Actualización de los nodos en las *listas de acceso*

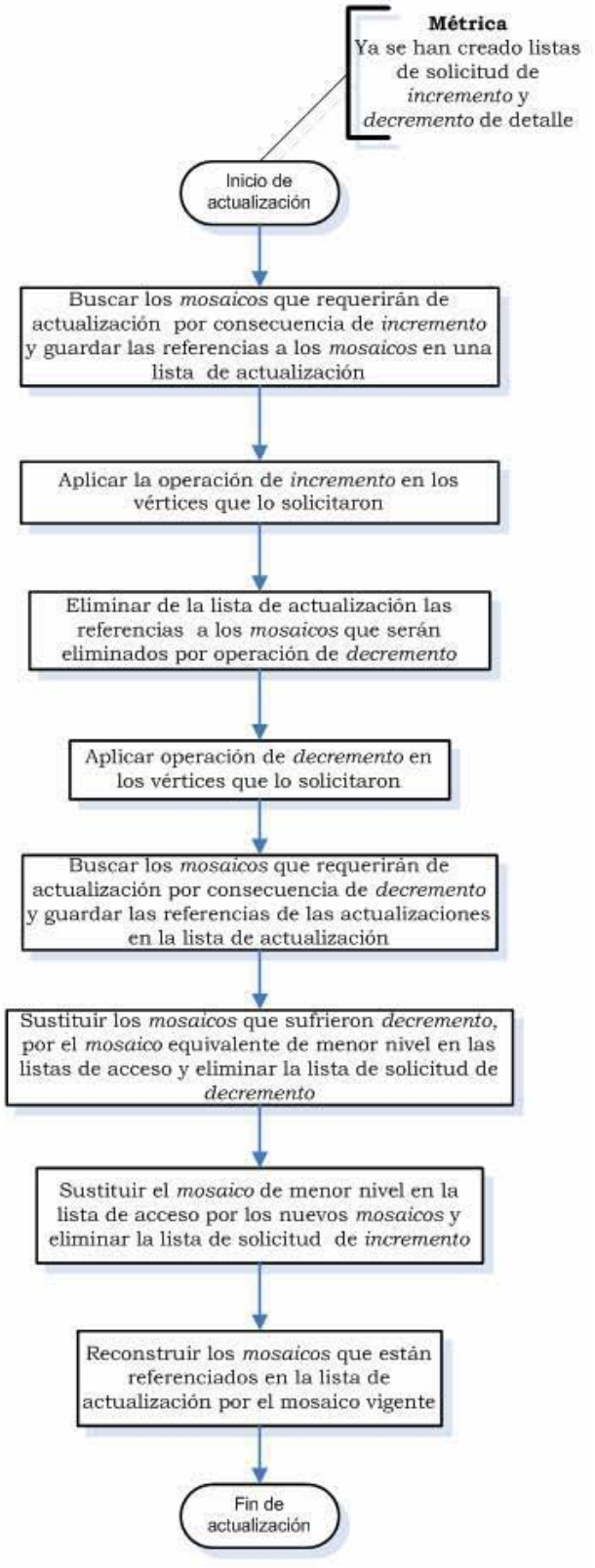


Figura 3.14
Proceso de actualización de *listas de acceso*, después de sufrir operaciones de LOD

3.1.4 Métrica

Habiendo definido las operaciones LOD se tiene el control de la malla para el incremento y decremento de vértices. La parte de las operaciones de LOD corresponde al *¿cómo?* pero en ello no se establece cuándo deben aplicarse ni en dónde, eso lo indica la métrica. Algunos algoritmos no consideran la aplicación de alguna métrica, delegando al usuario la aplicación del algoritmo con diferentes valores decidiendo para cada objeto como debiera ser representado.

Los algoritmos que regulan la simplificación pueden basar su métrica en alguna de dos categorías. Simplificación basada en fidelidad, técnicas que permiten al usuario especificar el nivel deseado de fidelidad en la simplificación de alguna manera para luego intentar minimizar el número de polígonos sin violar la restricción de fidelidad. La otra técnica de simplificación tiene por restricción el número de polígonos o vértices donde se pretende maximizar la fidelidad del modelo simplificado sin exceder el límite establecido para su construcción, por lo que también se conoce como técnica *basada en presupuesto* [4].

Para que la técnica LOD sea más útil, necesita tener soporte de ambos enfoques *fidelidad* y *presupuesto*, claro está que se prefiera *fidelidad* para la generación de imágenes que requieran de exactitud, mientras que alguna técnica de *presupuesto* es preferible donde el tiempo de *rendering* es crucial. Probablemente la descripción anterior despierte en usted la interrogante acerca de métricas de *fidelidad*, es solo que ciertamente ello no existe como propiedad en sentido paramétrico, sin embargo pretende abstraerse como un conjunto de parámetros que de algún modo en su conjunto arrojen información que puedan determinar el grado de similitud con el modelo original; parámetros que (adicionalmente al error geométrico introducido por la simplificación) determinen atributos tales como colores, normales, coordenadas de textura. Finalmente el resultado final es determinado por la forma en que luce el objeto.

La simplificación y la métrica deben considerar el propósito original del modelo, el modelo puede ser simplificado, el propósito no; es decir, la simplificación del modelo tridimensional podría resultar en modificaciones importantes dentro de la topología de la malla y esto puede no ser válido para el carácter de ciertas aplicaciones tales como CAD, bioquímica y en medicina como son las tomografías y las resonancias magnéticas. La aplicación de alguna técnica LOD sin previsión de ello puede traer graves errores o dejar sin sentido alguno el análisis aplicado. El presente trabajo no tiene por objeto ninguna aplicación de carácter crítico como las descritas en el párrafo anterior y tratándose de una malla cerrada convexa es posible aplicar una técnica LOD de manera menos rigurosa que las requeridas en otros casos[3].

Los algoritmos de simplificación requieren de alguna técnica que permita ir guiando la aplicación de los operadores de simplificación local, es decir, evaluar de algún modo la simplificación implica comparar, comparar entre el *antes* y *después* de la simplificación contra el modelo original de manera interna, pretendiendo la correcta aplicación de los operadores durante la simplificación. Esto resultaría imposible si el modelo original se desconoce, lógicamente un modelo es simplificado solo cuando se conoce una versión con mayor detalle del mismo. Por ello no es posible aplicar alguna técnica de manera usual a nuestra malla si su geometría es dependiente de la animación

[3] Las simplificaciones topológicas en aplicaciones de naturaleza crítica deben ser guiadas por un experto en la materia, incluso sería sugerido evitar simplificaciones en análisis rigurosos donde la simplificación puede atraer serios problemas antes que algún beneficio *Fuente*: [4].

y el modelo de animación se desconoce, es decir, en el caso que nos ocupa no existe el modelo original y la técnica LOD debe aplicarse en tiempo de ejecución totalmente.

El modelo propuesto para determinar cuando es necesario aplicar las operaciones de LOD (*incrementento* y *decremento* descritas anteriormente) están basadas en función de la varianza de cada *triangle_fan*. Para el propósito que nos ocupa, solo es necesario estimar la dispersión (varianza) de la *altura* (*eje z*) de los vértices que componen el *mosaico* y establecer en función de la estimación si es requerida la aplicación de alguna operación LOD. La determinación de la profundidad de las operaciones LOD puede ser restringida en función de la distancia del observador. Es decir la naturaleza de la geometría determina si requiere la aplicación de alguna operación de LOD mientras que la distancia al observador acota el nivel de detalle con el que debe ser tratada esa operación de LOD. La idea de modular el LOD en función de la distancia es en principio simple, pocos son los detalles apreciables a una distancia *considerable* lo que permite reducir el detalle de la geometría sin afectar gravemente la fidelidad de la imagen.

El factor distancia es popular entre las técnicas LOD, es simple y eficiente, solo algunas condiciones son suficientes para comprobar si una distancia excede los umbrales predefinidos y el único cálculo potencialmente costoso está en el computo de la distancia entre dos puntos. Aunque es una idea simple y útil al emplearse tiene ciertas desventajas asociadas; la elección de un punto arbitrario del objeto puede traducirse en inexactitud y la distancia puede verse afectada cuando la orientación del observador es modificada. El cálculo de la distancia no considera los parámetros de la proyección de la perspectiva empleada, por ello, bajo ciertas circunstancias pueden aparecer efectos de *popping*[4].

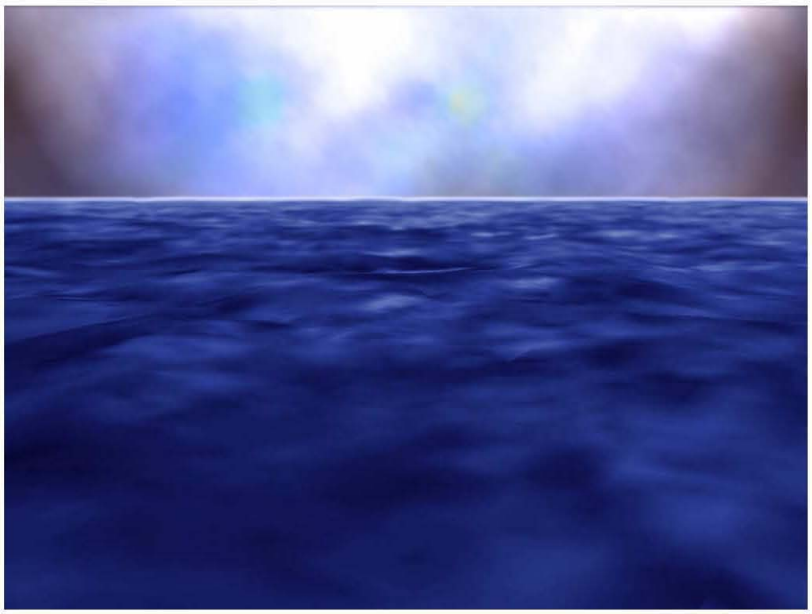
El modelo propuesto también considera el efecto de histéresis[5] como consecuencia de los cambios de LOD, el propósito es evitar el constante cambio en el modelo evitando algún efecto visual no deseado como titilo y evitar la innecesaria aplicación de incremento o decremento. Por ello se ajusta dentro de la técnica LOD una zona neutra como la comentada en la figura 1.5 denotada como zona de umbral.

[4] *Popping*: Es un efecto no deseado que puede presentarse durante la aplicación de alguna técnica LOD. Consiste en percibir notablemente los cambios en la resolución poligonal de los modelos tridimensionales durante las operaciones de LOD.

[5] En el contexto de LOD, es un grado del retraso introducido en la conmutación entre los niveles del detalle. La histéresis se utiliza para evitar el efecto de cambio en los objetos que cambian continuamente por encontrarse en la distancia umbral.

Capítulo 4

Implementación



El capítulo cuatro detalla una implementación de la arquitectura propuesta. La presente imagen es un ejemplo de imagen generada por computadora apoyada en la librería desarrollada y retocada en Photoshop.

4.1 DISEÑO

4.1.1 Descripción General

En el diseño de la arquitectura se ha considerado la flexibilidad como una cualidad deseable, de modo que sea posible modificar o extender las capacidades definidas en un principio.

A continuación se pretende describir más detalladamente su implementación, con el propósito de establecer sus límites y funcionalidades básicas.

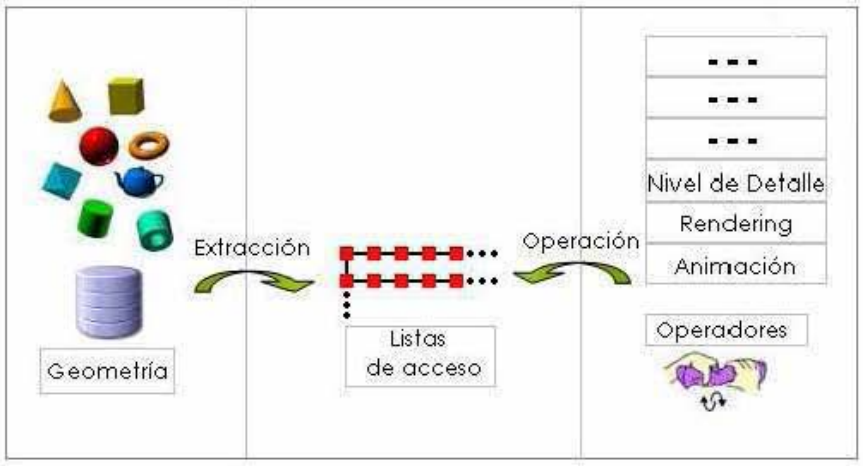


Figura 4.1
Diseño de la arquitectura

La figura 4.1 muestra la arquitectura de manera similar a la presentada en el capítulo dos (figura 2.1), se hace énfasis en tres partes que podemos diferenciar dentro de la arquitectura para fines de la implementación; la geometría, las listas de acceso y los operadores.

Para el caso que nos ocupa la geometría es una malla plana que hemos discutido a detalle en los capítulos anteriores, sin embargo es posible adaptar la arquitectura para el empleo de una geometría distinta. Las *listas de acceso* es la interfaz entre la geometría y los operadores, permitiendo un acceso ordenado y más rápido que si se accediera a los vértices directamente a través de la geometría. Finalmente los operadores, que permiten manipular la geometría.

Cada uno de los bloques anteriores (geometría, *listas de acceso* y operadores), han sido definidos como módulos independientes. Pueden modificarse o reemplazarse si las especificaciones presentes no satisfacen la necesidad del usuario. El diseño orientado a objetos permite esta flexibilidad y cada uno de los componentes de la arquitectura han sido modelados como clases (figura 4. 6)

Se ha puesto especial atención en los operadores, de modo que el usuario pueda definir nuevos operadores e incorporarlos de manera simple a la arquitectura.

El presente capítulo ofrece información acerca de la implementación de la arquitectura como librería sin embargo no es indispensable dicha información para el empleo de la librería. Basta establecer el tamaño y el número de *mosaicos* para crear una instancia de la malla y poder incorporarla en una aplicación. Las características de la malla dependerán del hardware. Establecer las propiedades de cada módulo (*settings*) es opcional, sin embargo es muy sugerido realizar los ajustes para conseguir los efectos deseados.

Básicamente son dos las formas en que se manejan los operadores que se aplican sobre la malla, el primero de ellos que denominaremos *no administrado* consiste en la aplicación de los operadores sobre la totalidad de la malla representada en las *listas de acceso*, aun cuando la malla no sea visible en la escena.

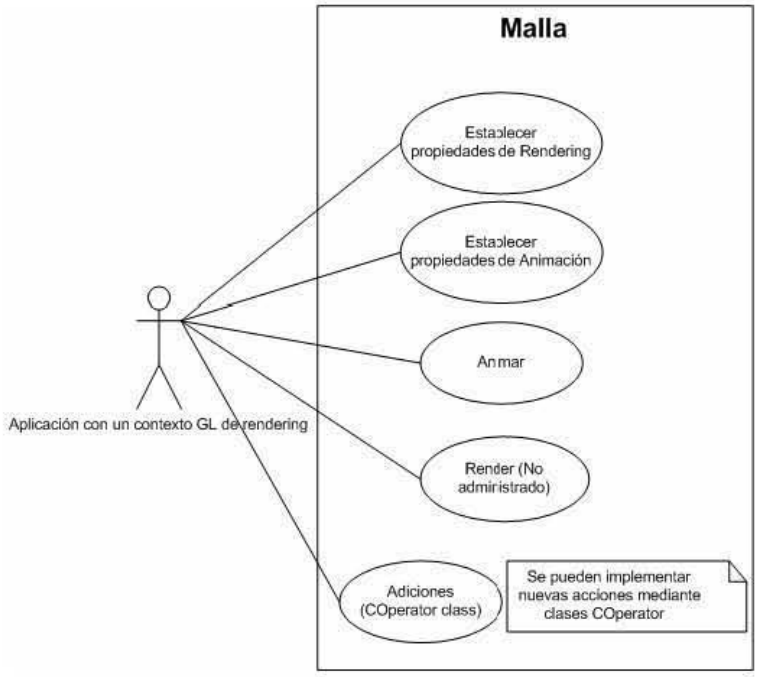


Figura 4.2
Caso de Uso 1

El primer *caso de uso* (figura 4.2), ilustra las tareas más comunes (animación y *rendering*). Pueden modificarse las propiedades de ambos operadores y aplicarlos, obteniendo una visualización tridimensional de una malla con movimiento. En este caso las *listas de acceso* no dependerán de la visibilidad de la geometría en la escena, por lo que los operadores seguirán aplicándose sobre las *listas de acceso* que seguirán representando la totalidad de la geometría.

Es posible establecer un escenario similar al presentado en el primer *caso de uso* pero de modo *administrado* (figura 4.3), es decir, si se proporciona información acerca de la cámara en escena es posible aplicar las operaciones sobre la sección de la malla que esta visible en escena, por ello, si la geometría está dentro o parcialmente fuera de escena las *listas de acceso* se reestructurarán de modo que las operaciones se apliquen exclusivamente en la geometría visible representada por las *listas de acceso*.

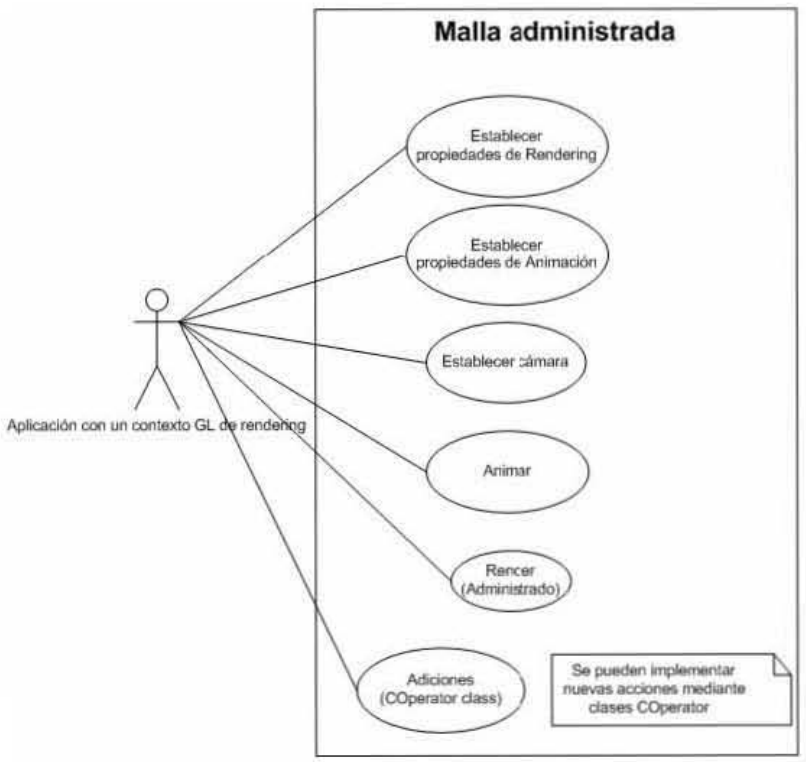


Figura 4.3
Caso de Uso 2

La figura 4.4 muestra un diagrama de secuencia, donde la clase *CWSystem* es la interfaz que agrupa los distintos módulos que operan sobre la geometría, pero no esta limitado a esos módulos, pueden definirse clases adicionales e interactuar con *CWSystem* de manera independiente a los demás módulos aun cuando estas nuevas definiciones no hayan sido definidas originalmente dentro de *CWSystem*. *CWSystem* ha sido definido exclusivamente como una agrupación de los distintos módulos de modo que sea una interfaz para el usuario final y delegar menos actividades al usuario final en el empleo de la librería.

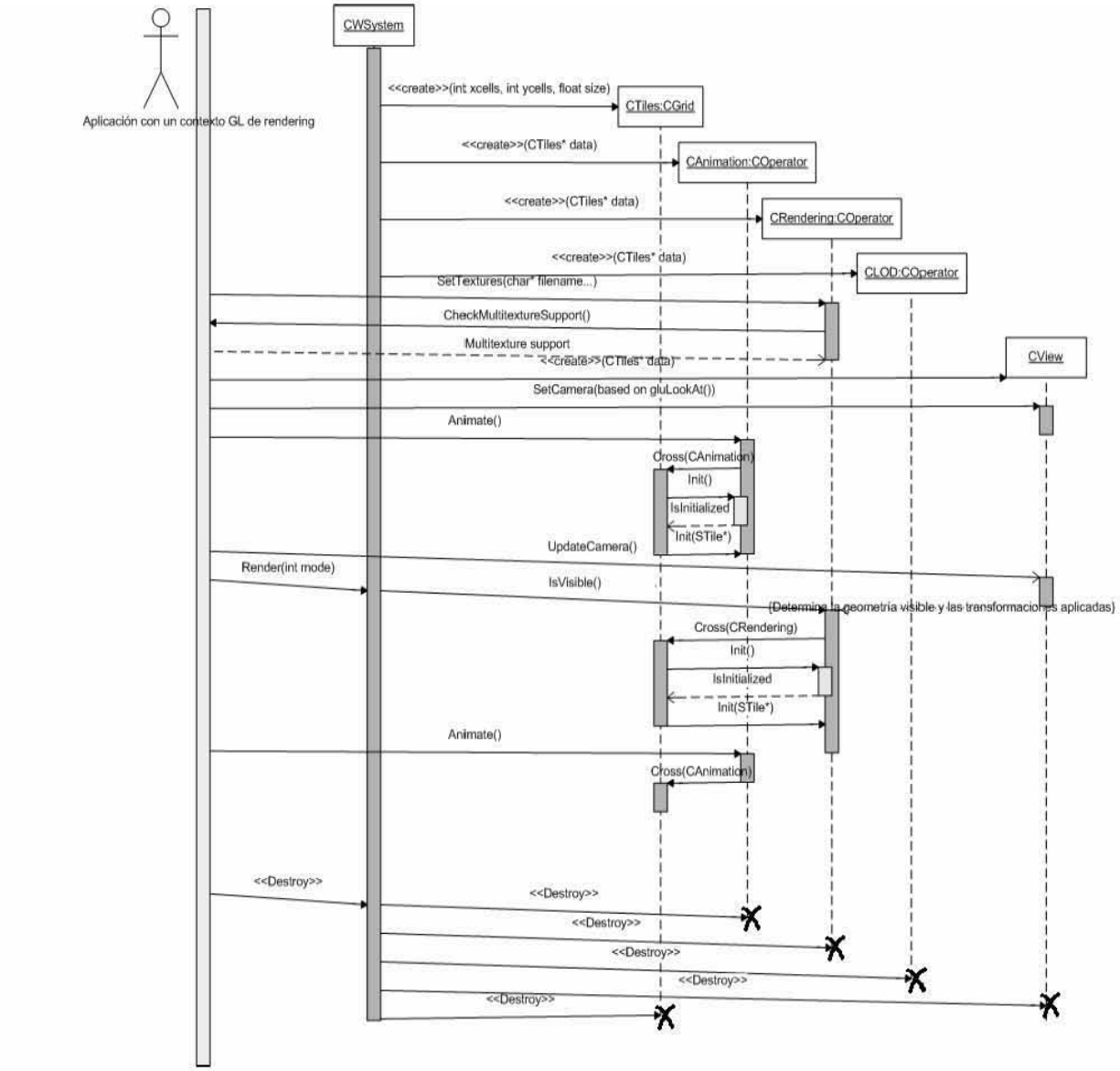


Figura 4.4
Diagrama de secuencia

La figura 4.5 muestra el diagrama de actividad que representa de manera simplificada la creación de *CWSystem* (interfaz que agrupa los distintos módulos) y los diferentes estados que dentro de la aplicación se traducen finalmente en el despliegue de una geometría animada.

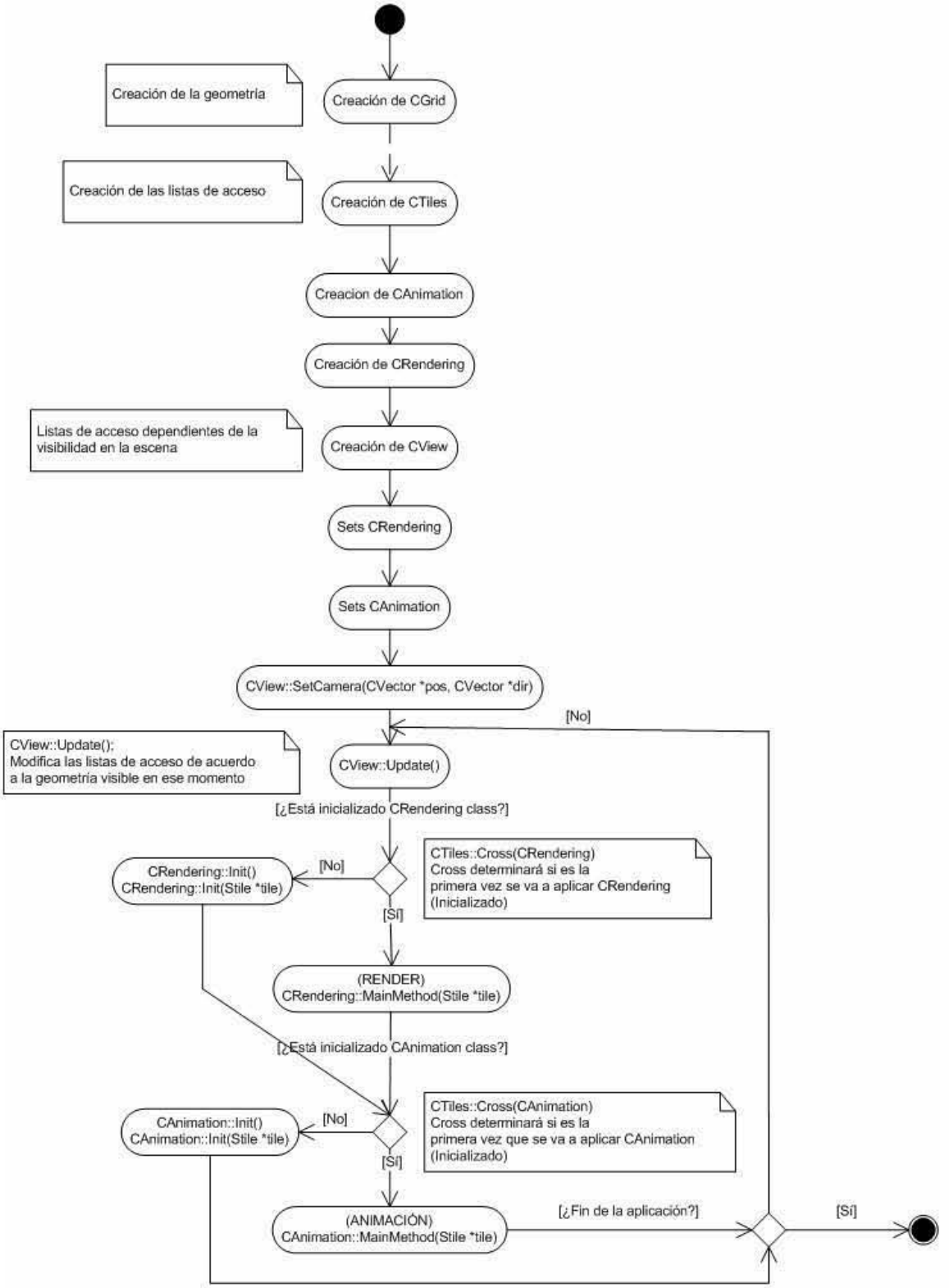


Figura 4.5
Diagrama de Actividad

El diagrama de actividad detalla estado a estado las actividades que implica disponer de la visualización tridimensional de una malla animada en una escena a través de la creación de una instancia de *CWSystem*, que como se comentó anteriormente reduce las tareas de usuario final al instanciar cada módulo automáticamente. El código 4.1 muestra dos formas de crear una superficie, una mediante el empleo de *CWSystem* y otra por medio de los módulos de manera independiente.

```
//Ejemplo, DrawCW y Draw son funciones equivalentes
void DrawCW(void)
{
       CWSystem *Water = new CWSystem(23,32,10); //Establece el tamaño de la malla
       Water->m_CRendering->SetTextures("marino.png");//Establece la textura
       Water->m_CAnimation->SetTide(2.2f);//Establece la amplitud de las oscilaciones
       while(…)
        {
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glMatrixMode(GL_MODELVIEW);
            glLoadIdentity();
            gluLookAt(0,0,0,0,10,0,0,0,1);
            …

             Water->m_CAnimation->Animate(); //Anima los vértices de la malla
             Water.Render(SRB_RTEXTURE);
            …
            SwapBuffers();
        }

}

void Draw(void)
{
       CTiles *data = new CTiles(23,32,10); ); //Establece el tamaño de la malla
       //Crea la instancia de Rendering asociada a la malla data
       CRendering *rendersys = new CRendering(data);
       //Crea la instancia de Animación asociada a la malla data
       CAnimation *animationsys = new CAnimation(data);
       rendersys->SetTextures("marino.png");//Establece la textura
       animationsys->SetTide(2.2f);//Establece la amplitude de las oscilaciones
       while(…)
        {
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glMatrixMode(GL_MODELVIEW);
            glLoadIdentity();
            gluLookAt(0,0,0,0,10,0,0,0,1);
            …
             animationsys->Animate();//Anima los vértices de la malla
             rendersys->Render(RENDER_TEXTURE);//Renderiza la geometría
            …
              //rendersys->Render(RENDER_TEXTURE); es equivalente a Cross(rendersys)
              //animationsys->Animate() es equivalente data->Cross(animationsys)

            SwapBuffers();
        }
}
```

Código 4.1
CWSystem es solo una interfaz.

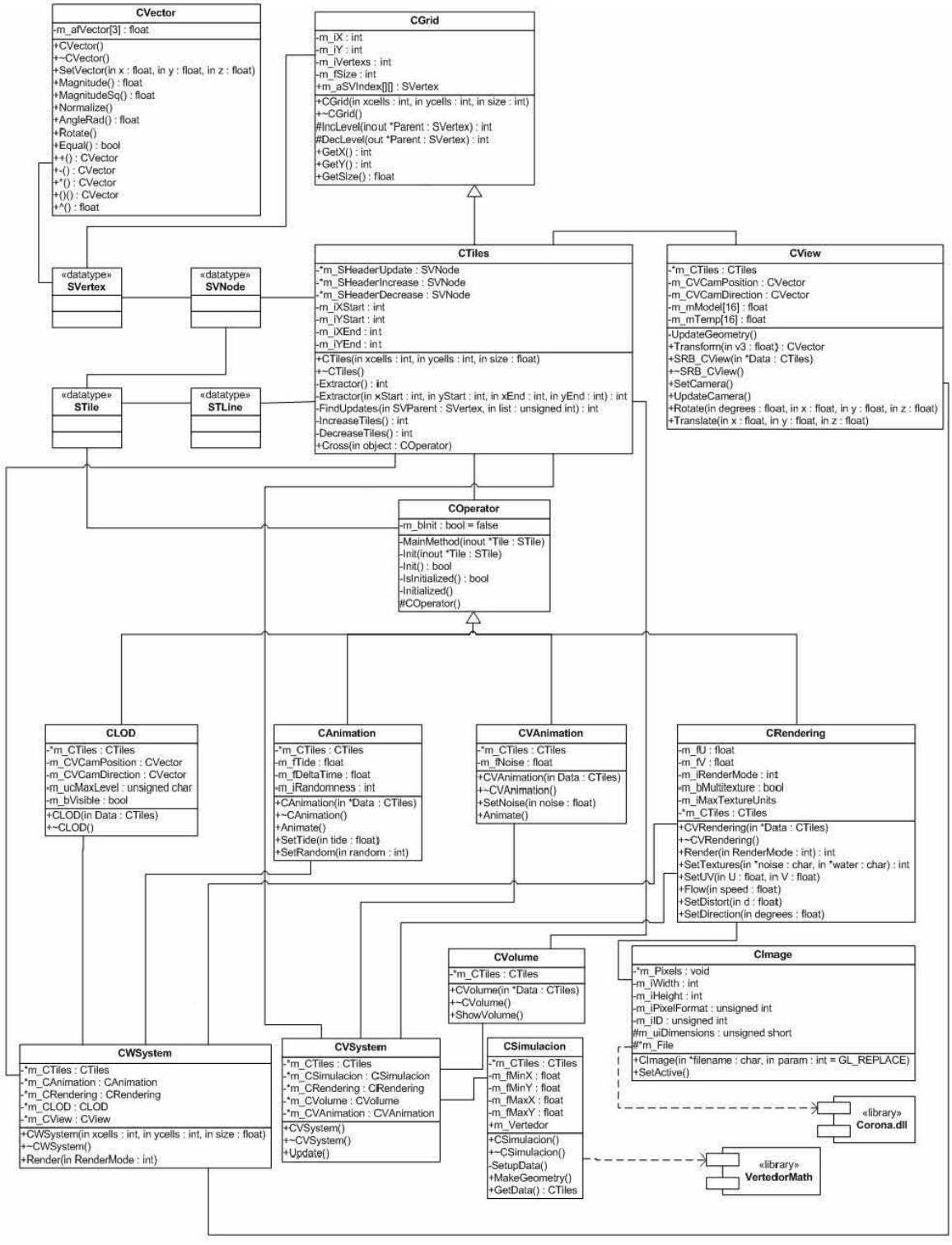


Figura 4.6
Estructura estática (Diagrama de clases)

4.1.2 Flexibilidad de la arquitectura

Es razonable considerar la posibilidad de incorporar nuevas clases que puedan interactuar con la malla, por ello se describe a continuación una guía para generar nuevas clases y adaptarlas a la implementación existente.

El primero de los modos sugeridos para extender la arquitectura original consiste en la creación de nuevos operadores. Esto implica codificar en la nueva clase los métodos virtuales heredados declarados en *COperator* (Código 4.2).

```
class COperator
{
	friend void CTiles::Cross(COperator* object);

	private:
		bool m_bInit;
		virtual void MainMethod(STile* tile) = 0;
		virtual void Init(STile* tile) = 0;
		virtual bool Init(void) = 0;

		bool IsInitialized(void)
		{
			return m_bInit;
		}
		void Initialized(void)
		{
			m_bInit = true;
		}

	protected:
		COperator(void)
		{
			m_bInit = false;
		}
		~COperator(void)
		{}

};
#endif
```

Código 4.2
Implementación de la clase COperator

- **void MainMethod(STile* tile)**

En este método se define la operación a aplicar sobre el conjunto de vértices definidos en cada *STile*. Recordemos que la estructura *STile* es una lista de vértices contenidos en orden de *triangle_fan* como se comentó anteriormente en la figura 2.7.

Propongamos un nuevo método que permita hacer un *rendering* de la geometría con puntos.

El ciclo *while* en el código propuesto (Código 4.3), permite visitar cada vértice*(SVNTemp->SVertex)* contenido en la lista definida por *tile*. Es común emplear un ciclo *while* en esta clase de métodos para modificar cada vértice.

```
void CNewClass::MainMethod(STile *tile)
{

        if(tile==0)
        return; //Apuntador nulo

        SVNode* SVNTemp;
        SVNTemp = tile->SVNode; //Nodo de la lista de vértices

        while(SVNTemp) //Mientras el nodo sea diferente de nulo
        {
                glBegin(GL_POINTS);
                        glVertex3fv(SVNTemp->SVertex->CVPosition.GetVector());
                glEnd();

                SVNTemp = SVNTemp->next; //Siguiente nodo
        }
}
```

Código 4.3
Ejemplo, definición de MainMethod(STIle *tile) para lograr un *rendering* con puntos

- **void Init(STile* tile)**

En algunos casos es resulta conveniente modificar o conocer alguna(s) propiedad(es) de los vértices antes de llamar a `MainMethod(STile* tile)`, para ese propósito se creo `Init(STile* tile)`. Se procede de manera idéntica que `MainMethod(STile* tile)` pero se aplicará solamente una vez sobre la geometría antes de `MainMethod(STile* tile)`.

- **bool Init(void)**

A diferencia de los dos métodos anteriores, `Init(void)` esta ideado en la inicialización de la propia clase, cualquier tarea propia de la clase que deba ser ejecutada antes de `Init(STile *tile)` y/o `MainMethod(STile *tile)` puede ser incluida en `Init(void)`. Si esto no es necesario será suficiente regresar verdadero.

```
bool CNewClass::Init(void)
{
        return true;
}
```

Código 4.4
Aun cuando no sea necesaria alguna tarea, es obligada la definición de los métodos virtuales

Una vez que ha sido definida la nueva clase es necesaria una interfaz que permita la interacción entre las *listas de acceso* y los métodos descritos anteriormente.

CTiles (clase responsable de las *listas de acceso*) cuenta con un método que recibe por parámetro instancias de la clase *COperator* y aplica los métodos descritos anteriormente a cada nodo (*STile*) de las *listas de acceso*. Aunque no es posible instanciar objetos de la clase *COperator* puesto que el constructor es `private`, *COperator* esta diseñada como una clase base que permita crear nuevas clases que hereden de ella y sería posible hacer uso de la interfaz proporcionada por *CTiles* por la propiedad de polimorfismo.

```
class CNewClass: public COperator
{
	private:
	 CTiles *m_CTiles;

	 void MainMethod(STile *tile);
	 bool Init(void);
	 void Init(STile *tile);

	public:
	 CNewClass(CTiles *data)
	{
		m_CTiles = data;
	}
	~CNewClass()
	{}
	void Render(void); //Método
};

void CNewClass::MainMethod(STile *tile)
{
	if(tile==0)
		return; //Apuntador nulo

	SVNode* SVNTemp;
	SVNTemp = tile->SVNode; //Nodo de la lista de vértices

	while(SVNTemp) //Mientras el nodo sea diferente de nulo
	{
		glBegin(GL_POINTS);
			glVertex3fv(SVNTemp->SVertex->CVPosition.GetVector());
		glEnd();
		SVNTemp = SVNTemp->next; //Siguiente nodo
	}
}

void CNewClass::Init(STile *tile)
{
	return;
}

bool CNewClass::Init(void)
{
	return true;
}

void CNewClass::Render(void)
{
	m_CTiles->Cross(this);//Aplicar la operación definida(MainMethod) a la geometría
}
```

Código 4.5
Interacción entre nuevas clases y la geometría

`void CTiles::Cross(COperator* object)` es la interfaz que permite la interacción entre las *listas de acceso* y algún operador. `Cross(COperator* object)` será responsable de aplicar `MainMethod(STile* tile)` a cada tile en las *listas de acceso*, pero también es responsable de inicializar la clase llamando a Init(void) e Init(STile* tile) si no ha sucedido anteriormente.

En algunos casos las nuevas clases requieren tener información adicional acerca de la geometría para establecer el valor de las variables miembro, por ello incorporan como variable miembro de la clase un apuntador a la instancia de *CTiles*, esto permite adicionalmente que la nueva clase pueda ejecutar su método principal por medio de un método que en ella misma este definida (Código 4.5).

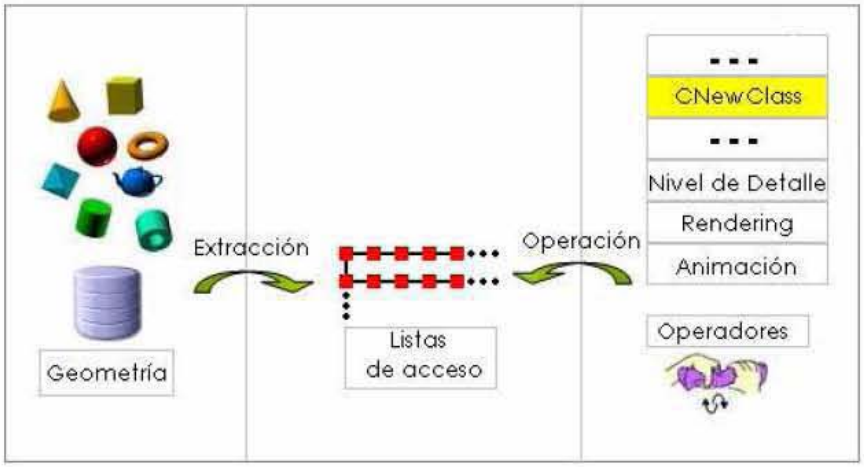


Figura 4.7
CNewClass (Código 4.5) se sitúa en el esquema de la arquitectura como un nuevo operador.

En algunos casos quizá sea requerida una interacción con la geometría de distinta manera a la que ofrece la clase *COperator*. Esta clase de interacción consiste en acceder al índice de vértices *padre de nivel cero* a través de *CTiles*.

`CTiles::m_aSVIndex[ycells][xcells]` contiene todos los vértices centrales de la malla, y podría resultar más compleja la interacción cuando se llevan a cabo operaciones de LOD, pues los nuevos vértices *padre* no están indexados.

Se debe considerar que cada uno de los vértices puede acceder a vértices vecinos por medio de alguno de sus cuatro apuntadores (figura 4.8).

La forma en que se acceda a los vértices a través del arreglo y se manipulen dependerá del propósito de dicha interacción. Sin embargo, puede resultar inconveniente manipular los vértices de este modo si se han aplicado operaciones de LOD, pues el arreglo `CTiles::m_aSVIndex[ycells][xcells]` sólo contiene vértices *padre* de nivel cero.

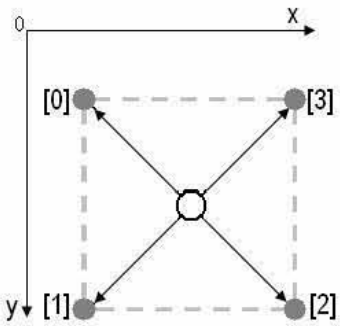


Figura 4.8
Apuntadores del vértice a sus vecinos SVertex->aLinks[0-3]

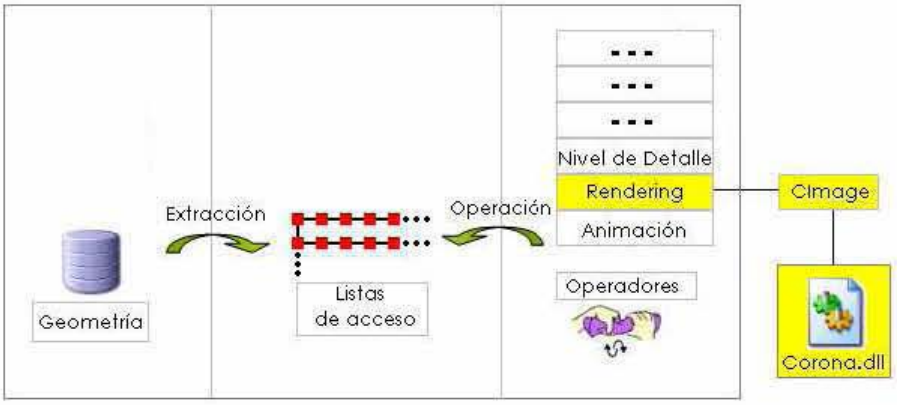


Figura 4.9
Una arquitectura flexible

La arquitectura hace uso de componentes externos para el manejo de archivos de imagen para luego ser empleadas como texturas. La arquitectura no es responsable de la manipulación o carga de archivos de imagen, eso es delegado a componentes externos, en este caso dicho componente se llama Corona[1]. *CImage* es una interfaz que permite establecer los archivos de imagen como texturas para un contexto de OpenGL siendo posible el empleo de texturas dentro de la arquitectura (particularmente en el módulo de *Rendering* o cualquier otro nuevo módulo que requiriera de texturas) de manera transparente sin necesidad de codificación extra por parte del usuario.

Corona soporta los siguientes formatos para su lectura: PNG, JPEG, PCX, BMP, TGA, y GIF. Aunque es posible la sustitución de Corona por otra librería que satisfaga las requisitos del usuario sin la necesidad de realizar cambios internos a la arquitectura, no obstante Corona se adecuó muy bien a los requisitos del diseño pues cubre un amplio número de formatos de imagen, es posible su empleo en distintos sistemas operativos, está escrita en C++ además de ser distribuida bajo licencia zlib/png[2].

[1] Corona es una librería open-source capaz de leer, guardar y manipular archivos de imagen, escrita en C++ y portable a sistemas Windows, Linux y Irix http://corona.sourceforge.net/
[2] Detalles de la licencia zlib/libpng en http://opensource.org/licenses/zlib-license.html Básicamente permite el uso del código como se desee, siempre que no se clame como propio.

Conclusiones

La arquitectura propuesta posibilita la representación de superficies con soporte para el control de nivel de detalle. Se desarrolló una librería como implementación de la arquitectura propuesta, empleándose en aplicaciones de tiempo real que se ejecutan en sistemas Windows y Unix, lo que verifica la portabilidad de su implementación.

La modificación y/o sustitución de los módulos que conforman la arquitectura es posible, dichas modificaciones deberán ser guiadas por los requerimientos determinados por el propósito de la aplicación. Su diseño modular permitió incorporar nuevas funcionalidades no previstas en su fase de diseño lo que prueba su extensibilidad para futuros requerimientos.

Por el reto que implica el diseño de una métrica, la arquitectura ofrece la capacidad de adaptarse a nuevas métricas según el objetivo que se persiga. La arquitectura ofrece soporte para las tareas de control de nivel de detalle, sin embargo, el diseño de una métrica para el control de nivel de detalle para propósito general en tiempo real manifiesta dos problemas: El primer problema recae en el hardware por ser una tarea demandante, por ello algunas técnicas LOD requieren de información generada en un procesamiento previo que facilite su aplicación en tiempo real. El segundo problema en el diseño de la métrica consiste en definir la *mejor apariencia* en términos matemáticos lo cual no es posible y se dificulta aun más cuando la mejor apariencia es dependiente del contexto de la aplicación.

Para futuros desarrollos se sugiere la implementación de *shaders*[1] para lograr nuevos efectos de rendering y aminorar la carga de trabajo en la CPU. El diseño de una métrica robusta que permita lograr la aplicación de técnicas LOD en grandes superficies dinámicas y finalmente sería recomendable implementar un administrador de memoria que maneje las solicitudes de dicho recurso.

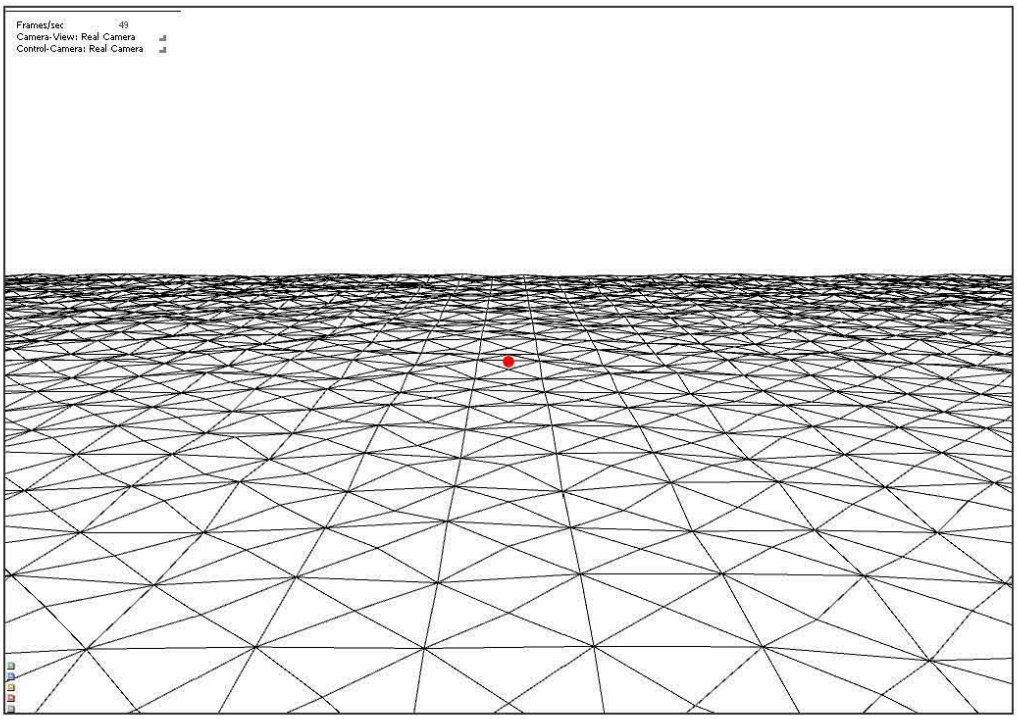
Su diseño ha permitido acoplarse a diversas aplicaciones, como se muestra en el proyecto de visualización del proyecto hidráulico El Cajón para el laboratorio de visualización Ixtli, tanto en su versión Windows como en en Unix.

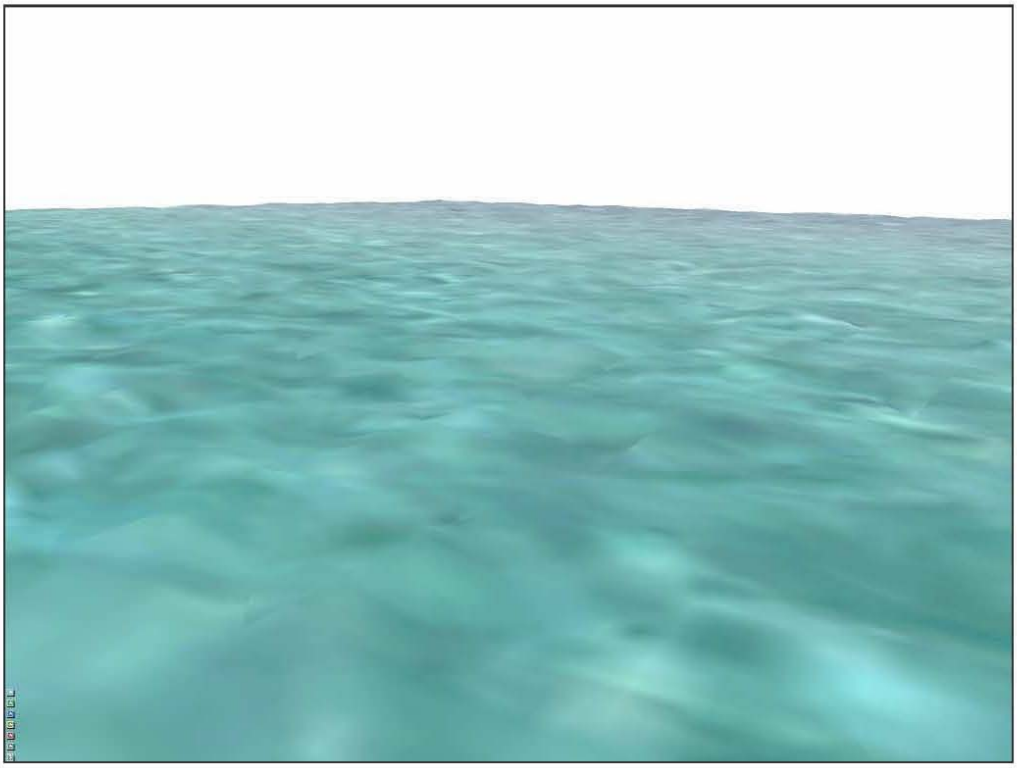
Los resultados del presente trabajo cumplen con los requisitos planteados y de manera alentadora, pues actualmente sigue en extensión el presente trabajo para satisfacer algunas de las necesidades que se presentan en proyectos para el laboratorio de visualización Ixtli.

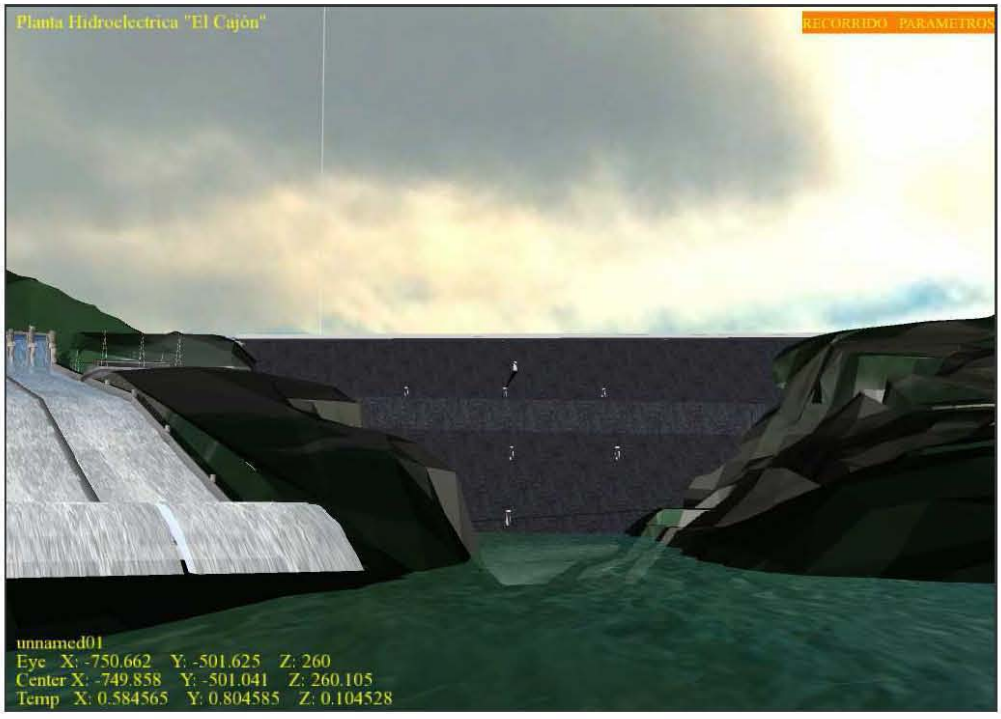
En las páginas siguientes se muestran algunas imágenes generadas por computadora que involucran la presente tesis.

---

[1] El término *shader* es empleado para definir un conjunto de instrucciones que modifican una etapa especifica del *graphics pipeline* (serie de etapas necesarias en la rasterización de una escena 3d) . Esto es posible pues estas instrucciones permiten acceder a características avanzadas del hardware y permite mayor flexibilidad al poder programar sobre la GPU (unidad de procesos gráficos).


Frames/sec 49
Camera-View: Real Camera
Control-Camera: Real Camera





Imágenes referentes al proyecto de visualización *El Cajón* de la sala Ixtli.

Apéndice

A.1 Documentación del API

Aquí se presenta como referencia la documentación de cada una de las clases que en conjunto permiten la representación tridimensional de superficies.

La librería desarrollada en C++ ha sido desarrollada en Visual Studio 2003, sin embargo también es posible su compilación en ambientes UNIX.

```
#include <CWSystem.h>
CWSystem
Atributos públicos
      CTiles      *m_CTiles
      CAnimation  *m_CAnimation
      CRendering  *m_CRendering
      CLOD        *m_CLOD
Métodos públicos
      <constructor> CWSystem(int xcells, int ycells, float size)
      <destructor>  ~CWSystem(void)
      void  Render(int RenderMode)
      void  SetCamera(SRB_CVector *Position, SRB_CVector *ViewPoint)
      void  ResetPosition(void)
      void  Translate(float x, float y, float z)
      void  Rotate(float degrees, float x, float y, float z)
```

CWSystem es la clase más importante desde el punto de vista del usuario, básicamente es una clase que conjunta las clases diseñadas para interactuar con la geometría, son pocos los métodos implementados en ella, la intención es mantener una interfaz simple que permita realizar modificaciones de manera más sencilla y brindar al usuario una interfaz sobria basada más en las clases miembro que en métodos *wrapper* que permitieran llamadas a los métodos de dichas clases.

La intención es permitir por medio de las clases miembro acceder a sus métodos, la inclusión de una nueva clase bien definida establecerá los atributos privados, públicos o protegidos según sea conveniente y podrá manipular la geometría.

```
CWSystem::CWSystem(int xcells, int ycells, float size)
(Constructor)
Parámetros
      xcells      mosaicos en x
      ycells      mosaicos en y
      size        tamaño del mosaico
```

Construye las instancias variables miembro necesarias para visualizar y animar una malla. (Ver código 4.1)

CWSystem::~CWSystem()

(Destructor)

Destructor de la instancia.

void CWSystem::Render(int RenderMode)

Parámetros

SRB_RWIRE	modo de malla
SRB_RSOLID	modo sólido sin texturizado
SRB_RTEXTURE	modo sólido texturizado

Aplica las transformaciones geométricas sobre la malla y llama al método de *rendering CRendering::Render(int RenderMode)*..

void CWSystem::SetCamera(SRB_CVector *Position, SRB_CVector *ViewPoint)

Parámetros

Position	Apuntador a vector que represente la posición de la cámara
ViewPoint	Apuntador a vector que represente la orientación de la cámara

Establece la referencia al modelo de cámara que permite aplicar las operaciones sobre la geometría visible. Sino se llama a este método todas las operaciones se seguirán aplicando a la malla aun cuando este fuera de la vista de escena.

void CWSystem::ResetPosition(void)

Sustituye los valores de la matriz de transformación de la geometría por la matriz identidad, la geometría regresa a la posición y orientación original.

void CWSystem::Translate(float x, float y, float z)

Traslada la geometría de manera similar a *glTranslatef*. La traslación realizada es persistente y no tiene que ser llamada cada *frame* (en cuyo caso sería acumulativa). Es aconsejado realizar las transformaciones de traslación por medio de este método en lugar de *glTranslatef*, con el fin de mantener consistencia en las operaciones de visibilidad habilitadas cuando se establece la referencia a la cámara por medio de *CWSystem::SetCamera(SRB_CVector *Position, SRB_CVector *ViewPoint)*. Este método es útil aun cuando no se haya establecido alguna referencia a los vectores de cámara, pero no obligado.

void CWSystem::Rotate(float degrees, float x, float y, float z)

Parámetros

degrees	ángulo en grados que rotará la geometría alrededor del eje definido por las componentes *x*, *y*, *z*.
x	componente en x del vector de rotación
y	componente en y del vector de rotación
z	componente en z del vector de rotación

Rota la geometría de manera similar a *glRotatef*. La rotación realizada es persistente y no tiene que ser llamada cada *frame* (en cuyo caso sería acumulativa). Es aconsejado realizar las transformaciones de rotación por medio de este método en lugar de *glRotatef*, con el fin de mantener consistencia en las operaciones de visibilidad habilitadas cuando se establece la referencia a la cámara por medio de *CWSystem::SetCamera(SRB_CVector *Position*, SRB_CVector *ViewPoint) Este método es útil aun cuando no se establecido referencia a los véctores de la cámara, pero no obligado.

```
#include <CTiles.h>
CTiles:CGrid
Atributos públicos
      SVertex* m_aSVIndex[][]
Métodos públicos
      <constructor>  CTiles(int  xcells,int  ycells,  float  size)  :  CGrid
(xcells,ycells,size)
      <destructor>  ~CTiles()
      void     Cross(COperator *objects[], int nObjects)
      void     Cross(COperator* object)
      int      GetIndexXmin(void)
      int      GetIndexYmin(void)
      int      GetIndexXmax(void)
      int      GetIndexYmax(void)
      void     Remake(int xStart, int yStart, int xEnd, int yEnd)
      int      GetVertexs(void)
      int      GetX(void)
      int      GetY(void)
      float    GetSize(void)
```

CTiles contiene la geometría organizada en listas de *STiles*(arreglos de vértices ordenados en modo de triangle_fan). Sirve de interfaz con la geometría de manera de que pueda manipularse de manera más sencilla. En esta clase se establece la geometría activa (geometría visible en la escena), es decir, todas las clases *COperator* que operen sobre *CTiles* lo hacen exclusivamente sobre la geometría presente en las listas que puede diferir de la geometría total.

CTiles::CTiles(int xcells, int ycells, float size):CGrid(xcells, ycells, size)

(Constructor)

Parámetros

xcells	número de mosaicos en *x*
ycells	número de mosaicos en *y*
size	tamaño original del mosaico

Construye la instancia responsable de las *listas de acceso* a partir de la malla generada en la instancia de *CGrid*.

CTiles::~CTiles(void)

(Destructor)

Destructor de la instancia.

void CTiles::Cross(COperator *objects[], int nObjects)

Parámetros

objects[]	Un arreglo de apuntadores a *COperator*
n	Tamaño del arreglo

Este método provee una interfaz para las instancias que hereden de *COperator* apliquen sus operaciones en la geometría a través de las *listas de acceso*.

void CTiles::Cross(COperator* object)

Parámetros

Object	Apuntador a un objeto de tipo *COperator*

Este método provee una interfaz para las instancias que hereden de COperator apliquen sus operaciones en la geometría a través de las *listas de acceso*.

int CTiles::GetIndexXmin(void)

Retorna el índice menor vigente en *x* correspondiente a la última extracción (vértices visibles en la escena) a través del índice de apuntadores de vértices *padre* de nivel cero *CTiles::SVertex** `m_aSVIndex[y][x]`.

int CTiles::GetIndexYmin(void)

Retorna el índice menor vigente en *y* correspondiente a la última extracción (vértices visibles en la escena) a través del índice de apuntadores de vértices *padre* de nivel cero *CTiles::SVertex** `m_aSVIndex[y][x]`.

int CTiles::GetIndexXmax(void)

Retorna el índice mayor vigente en *x* correspondiente a la última extracción (vértices visibles en la escena) a través del índice de apuntadores de vértices *padre* de nivel cero *CTiles::SVertex** `m_aSVIndex[y][x]`.

int CTiles::GetIndexYmax(void)

Retorna el índice mayor vigente en *y* correspondiente a la última extracción (vértices visibles en la escena) a través del índice de apuntadores de vértices *padre* de nivel cero *CTiles::SVertex** `m_aSVIndex[y][x]`.

void CTiles::Remake(int xStart, int yStart, int xEnd, int yEnd)

Reconstruye las *listas de acceso* extrayendo únicamente los vértices delimitados por los índices. Es útil auxiliarse de los métodos *CTiles::GetIndex...*

int CTiles::GetX(void)

Retorna el número de vértices *padre* de *nivel cero* en dirección del eje *x*.

int CTiles::GetY(void)

Retorna el número de vértices de *nivel cero padres* en dirección del eje *y*.

float CTiles::GetSize(void)

Retorna el tamaño original de un *tile* de *nivel cero*.

int CTiles::GetVertexs(void)

Retorna el número total de vértices en la malla.

```
#include <CRendering.h>
CRendering:COperator
Atributos públicos

Métodos públicos
      <constructor> CRendering(CTiles *data)
      <destructor>  ~CRendering(void)
      void  Render(int RenderMode)
      void  SetDirection(float degress)
      int   SetTextures(char* noise,char* water)
      int   SetTextures(char* noise)
      void  SetUV(float U, float V)
      void  Flow(float speed)
      void  SetDistort(float d)
      void  SetRGBA(float red, float green, float blue, float alpha)
```

CRendering es responsable de realizar la tarea de *rendering* en modo de malla o sólido, la carga de texturas y su manipulación forman parte de las tareas ejecutadas por la clase. Habilitar la extensión de *multitexturizado* es parte de sus funciones.

CRendering::CRendering(CTiles *data)

(Constructor)

Parámetros

data Es un apuntador a una instancia *CTiles*(*listas de acceso*)

Crea una instancia capaz de realizar el *rendering* de alguna instancia CTiles

CRendering::~CRendering(void)

(Destructor)

Destructor de la instancia.

void CRendering::Render(int RenderMode)

Parámetros

SRB_RWIRE	modo de malla
SRB_RSOLID	modo sólido sin texturizado
SRB_RTEXTURE	modo sólido texturizado

Lleva a cabo exclusivamente la tarea de *rendering.*

void CRendering::SetDirection(float degrees)

Parámetros

degrees Establece la dirección en grados del flujo (traslación de la textura) mediante *CRendering::Flow(float speed).*

int CRendering::SetTextures(char *noise)

Parámetros

noise Es el nombre de archivo de imagen. Soporta formatos png, jpeg, pcx, bmp, tga, y gif. Si el formato de imagen soporta transparencia se apreciará en el *rendering*.

Retorno

Regresa el número de unidades de textura soportadas en *Multitexturing*. Si el retorno es igual a uno, no hay soporte para *multitexturing*.

Establece un archivo de imagen como textura. Esta textura no es afectada por CRendering::SetDistort(float d).
Los formatos soportados corresponden con a los formatos soportados por el la librería Corona.

int CRendering::SetTextures(char* noise, char* water)

Parámetros

noise Es el nombre de archivo de imagen. Soporta formatos png, jpeg, pcx, bmp, tga, y gif. Si el formato de imagen soporta transparencia se apreciará en el *rendering*. Esta textura es empleada generalmente para brindar el efecto de flujo

water Es el nombre de archivo de imagen. Soporta formatos png, jpeg, pcx, bmp, tga, y gif. Si el formato de imagen soporta transparencia se apreciará en el *rendering*. Esta textura es empleada generalmente para brindar el aspecto y color del agua, se mantiene estática.

Retorno

Regresa el número de unidades de textura soportadas en *Multitexturing*.

Establece un par de archivos de imagen como texturas. El uso de este método implica contar con la capacidad de *Multitexturing* en la tarjeta de video, si el hardware no soporta esa característica se llevará a cabo el *rendering* solamente con la textura correspondiente al archivo *noise*. La textura del archivo *water* solo es afectada por el método *CRendering::SetDistort(float d).*

void CRendering::SetUV(float U, float V)

Parámetros

U Factor por el que se repite la imagen en la totalidad de la malla en *x*

V Factor por el que se repite la imagen en la totalidad de la malla en *y*

Modifica las coordenadas de texturizado para la textura *noise*. El valor por defecto es 1.0f para *U* y *V*, ambos parámetros corresponden al escalamiento de la textura en *x* y *y* respectivamente.

void CRendering::Flow(float speed)

Parámetros

speed Representa la velocidad de flujo (incremento en la translación de la textura por ciclo de *rendering*). Este factor es afectado por los valores de *CRendering::U* y *CRendering::V*

Traslada una de las texturas de modo que brinde un efecto de flujo.

void CRendering::SetDistort(float d)

Parámetros

d Representa la magnitud de la distorsión.

Afecta el modo en que es mapeada la textura *water*. Provocando un efecto de distorsión.

void CRendering::SetRGBA(float red, float green, float blue, float alpha)

Parámetros

red Las componentes que refieren a los valores RGBA.
green Rango[0.f – 1.f]
blue
alpha

Establece el color base sobre el cual van a ser aplicadas las texturas.

```
#include <CAnimation.h>
CAnimation:COperator
Atributos públicos

Métodos públicos
      <constructor> CAnimation(CTiles *data)
      <destructor>  ~CAnimation(void)
      void  SetTide(float tide)
      void  SetThick(float thick)
      void  SetDelta(float time)
```

CAnimation es responsable de realizar las tareas de animación.

CAnimation::CAnimation(CTiles *data)

(Constructor)

Parámetros

data Es un apuntador a una instancia *CTiles*(*listas de acceso*)

Crea una instancia capaz de realizar la animación de alguna instancia *CTiles*.

CAnimation::~CAnimation(void)

(Destructor)

Destructor de la instancia.

void CAnimation::SetTide(float tide)

Parámetros

tide Representa la amplitud de la oscilación (amplitud de las funciones trigonométricas)

Establece la magnitud de la oscilación en la animación.

void CAnimation::SetThick(float thick)

Parámetros

thick Factor por el cual se modifica la energía de cada vértice.

Representa el factor por el cual es disminuido el valor de energía de cada vértice por ciclo de animación, el valor tiene un rango abierto de (0,1). No tiene efecto sino existen eventos (clases/métodos) que modifiquen la energía de cada vértice.

void CAnimation::SetDelta(float time)

Parámetros

time Representa el incremento de tiempo para la evaluación de la función de animación.

Modifica el incremento de tiempo en que se modifica el tiempo particular de cada vértice. El método de animación es función del tiempo. Generalmente no es necesario modificarlo.

A continuación se muestra un ejemplo que permite incorporar la malla dentro de una escena OpenGL.

```
#include <CWSystem.h>

void DrawScene(void);

void main(void)
{
       …
       CreateGLWindow();
       SetupGL();
       …
       DrawScene();
       …
}

void DrawScene(void)
{
       CVector *pos = new CVector(0,0,0);
       CVector *dir = new CVector(300,50,0);
       CWSystem mesh(27,34,12.5f);
       //Establece en el modulo de rendering las texturas que vamos a emplear
       mesh.m_CRendering->SetTextures("textures/noise.png","textures/water.png");
       //Establece la velocidad de traslado de la textura0 (en este caso noise.png)
       mesh.m_CRendering->Flow(0.0025f);
       //Traslada la geometría a otra posición (x,y,z)
       mesh.Translate(322,120,-10);
       mesh.SetCamera(pos,dir);

       while(…)
       {
           glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
           glMatrixMode(GL_MODELVIEW);
           glLoadIdentity();
           gluLookAt(pos->GetVx(),pos->GetVy(),pos->GetVz(),dir->GetVx(),
            dir->GetVy(),dir->GetVz());
           …

                mesh.m_CAnimation->Animate(); //Anima los vértices de la malla
             /*Aplica las transformaciones, actualiza la lista de acceso con
             respecto a la camara y renderiza la geometría*/
               mesh.Render(RENDER_TEXTURE);
           …
           SwapBuffers();
       }
}
```

Código 4.6
Inclusión de una malla en una escena de OpenGL

A.2 EXTENSIONES GL

Una extensión es fundamentalmente una especificación, como lo es OpenGL. Es una forma de incorporar nueva funcionalidad a OpenGL.

De este modo algunos vendedores de hardware pueden proveer nuevas funcionalidades sin que esta forme parte base del API. Cuando dicha extensión es demasiado amplia o se considera de bastante importancia por la ARB (Architecture Review Board), se promueve para ser parte base del API. Por ello muchas de las funciones que forman parte de OpenGL actualmente, comenzaron exclusivamente como una extensión. Una vez que es creada la extensión, contiene en su registro información acerca de si misma incluyendo una breve justificación de su existencia, el nombre de exacto de la cadena que la identifica y finalmente las funciones y tokens que haya agregado.

La mayoría de las extensiones agregan nuevas funciones o nuevos tokens (símbolos), pero algunas otras simplemente permiten que las funciones y tokens ya existentes puedan ser empleadas en formas que anteriormente no estaban permitidas[11][12].

Para emplear las extensiones hay que seguir tres pasos:

- ✓ Determinar si la extensión es soportada
  Esto es fundamental, pues las extensiones dependen de la plataforma y hardware para ser soportadas, por ello es obligado verificar que se cuente con soporte antes de hacer uso de ellas.

- ✓ Obtener el punto de entrada para cualesquiera de las funciones de la extensión
  Es necesario porque no se tienen disponibles las funciones de extensión cuando es compilado el programa, por ello es necesario ligarlas dinámicamente en tiempo de ejecución, es decir obtener el punto de entrada.

- ✓ Definir cualquier token que se vaya a emplear.
  El empleo de extensiones puede requerir del uso de tokens, siendo necesario definirlos de acuerdo a la especificación de la extensión, auque esto puede que sea innecesario si se usa un header como glext.h, glee.h u otro header de terceros donde estos tokens ya vienen definidos.

Hay que considerar que una extensión puede no ser soportada en algunos equipos, ello nos obliga a tomar acciones apropiadas que pudiese significar emplear otra extensión similar, deshabilitar lo dependiente a la extensión o incluso salir del programa de la manera más elegante posible; es importante no asumir que la extensión existe, sobretodo si se trata de extensiones específicas de algún fabricante. Un buen comienzo es determinar la versión del API de OpenGL, esto puede ser determinado usando `glGetString(GL_VERSION)`. La versión está basada en las capacidades del *rederer* y es variable por *renderer*, es importante hacer notar que la función es válida únicamente después de haber establecido el contexto de *rendering*, en otro caso el *renderer* es indeterminado. De igual manera, es necesario contar con un contexto de *rendering* antes de revisar la disponibilidad de las extensiones.

Una vez determinada la versión, puede conocerse el soporte de las extensiones en esa versión y/o proceder a buscar funcionalidades extendidas[1]. Las extensiones pueden diferir de plataforma a plataforma o entre *drivers*, por lo que los desarrolladores no pueden esperar encontrar interfases para todas las extensiones definidas en los archivos estándar de cabecera gl.h, glx.h y wgl.h, existen archivos adicionales disponibles en el sitio de opengl (glext.h, glxext.h, wglext.h) para solventar dicho inconveniente. Es útil el empleo de algún encabezado con la definición de los tokens que pudieran ser requeridos por las extensiones, por lo que es muy sugerido emplear algún archivo con las definiciones recientes. Los últimos encabezados[2] y librerías disponibles en sistemas Windows son para OpenGL 1.1, es decir los últimos encabezados y librerías de Windows están al menos cinco versiones y diez años atrasados. En el particular caso de la implementación del presente trabajo, se empleo GLee[3] como encabezado, lo que hace innecesaria la inclusión de gl.h pues lo contiene internamente.

A.2.1 Empleando extensiones

Verificar que la extensión sea soportada se realiza a través de `glGetString(GL_EXTENSIONS)`, dicha función regresa una cadena de texto listando todos los nombres de las extensiones soportadas por la implementación presente. Una vez obtenida la cadena de texto será necesario analizarla y buscar la subcadena que identifique la extensión que interesa.

Si la extensión es soportada (está presente la cadena en el arreglo regresado por `glGetString(GL_EXTENSIONS)`) será necesario obtener el punto de entrada para las funciones relativas a la extensión, porque no se tiene disponible la implementación de la extensión en tiempo de compilación y será necesario ligarla dinámicamente en tiempo de ejecución.

Finalmente resta definir los tokens que sean requeridos por la extensión según su especificación.

La web ofrece numerosos recursos e información adicional en relación a las extensiones OpenGL[4]. Realtech ofrece de manera gratuita un visor (OpenGL Extensión Viewer[5]) de extensiones OpenGL para sistemas Windows y MacOS proporcionando una lista detallada acerca de las extensiones de la implementación presente e información adicional acerca del sistema.

---

[1] Puede encontrarse más información acerca del registro de extensiones en el sitio oficial de OpenGL http://www.opengl.org/registry/

[2] El archivo de cabecera gl.h contiene definiciones de constantes para cada versión de opengl que el archivo de cabecera soporte, algo similar a `#define GL_VERSION_1_1` pero esas constantes no tienen relación con la versión de OpenGL soportada por el sistema operativo ni con el *renderer* actual presente, simplemente indica las declaraciones de las funciones base contenidas en ese particular archivo. Es suficiente (y muy sugerido) con actualizar los *drivers* de video ofrecidos por los fabricantes de hardware, para aprovechar la ultima implementación ofrecida en nuestra tarjeta de video.

[3] GLee (GL easy extension library), es una librería multiplataforma gratuita para la carga de extensiones OpenGL disponible en http://elf-stone.com/glee.php El sitio de OpenGL cita a la librería en http://www.opengl.org/sdk/libs/GLee/

[4] http://developer.apple.com/technotes/tn2002/tn2080.html Una buena nota que ayuda a entender más acerca de las extensiones y su detección.
http://oss.sgi.com/projects/ogl-sample/registry/ Documentación acerca del registro de extensiones OpenGL.

[5] http://www.realtech-vr.com/glview/crossplatform.html sitio con información acerca del visor OpenGL Extensión Viewer por realtech.

```
#include  "gl.h"

//Definiciones que podemos evitar, empleando algun encabezado
typedef  void  (APIENTRYP  PFNGLMULTITEXCOORD2FARBPROC)  (GLenum  target,  GLfloat  s,
GLfloat t);
typedef void (APIENTRYP PFNGLACTIVETEXTUREARBPROC) (GLenum texture);
typedef void (APIENTRYP PFNGLCLIENTACTIVETEXTUREARBPROC) (GLenum texture);


void draw(void)
{
        ...//Ya existe un contexto de rendering


        if(CheckExtension("GL_ARB_MULTITEXTURE"))//Existe el soporte?
        {
            //apúntadores a funciones
            PFNGLMULTITEXCOORD2FARBPROC      glMultiTexCoord2fARB  = NULL;
            PFNGLACTIVETEXTUREARBPROC        glActiveTextureARB    = NULL;
            PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB = NULL;

            //Solicitamos los puntos de entrada para las funciones
            glMultiTexCoord2fARB=(PFNGLMULTITEXCOORD2FARBPROC)
            wglGetProcAddress("glMultiTexCoord2fARB"); //válido sólo para windows
            glActiveTextureARB=(PFNGLACTIVETEXTUREARBPROC)
            wglGetProcAddress("glActiveTextureARB");//válido sólo para windows
            glClientActiveTextureARB=(PFNGLCLIENTACTIVETEXTUREARBPROC)
            wglGetProcAddress("glClientActiveTextureARB"); //válido sólo para windows


            /* Ya es posible emplear funciones tales como glMultiTexCoord2f(...)
             para llevar a cabo tareas de multitexturing */
            ...
        }
        else
        {
               //código ~sin soporte de multitexture
               ...
        }


}

bool CheckExtension(char* extensionName)
{
        /*
        Se  hace  casting  porque  la  función  regresa  originalmente  un  arreglo  de
        unsignged chars y las funciones de manipulación de cadenas requieren de signed
        chars
        */
        char* extensionList = (char*) glGetString(GL_EXTENSIONS);
        if (!extensionName || !extensionList)
            return false;
        while (*extensionList)
        {
            // find the length of the first extension substring
            //Busca la longitud de la primera subcadena
            unsigned int firstExtensionLength = strcspn(extensionList," ")
            if(strlen(extensionName) == firstExtensionLength &&
            strncmp(extensionName, extensionList, firstExtensionLength) == 0)
            {
                return true;
            }
                // recorrerse a la siguiente subcadena
                extensionList += firstExtensionLength + 1;
        }
        return false;
}
```

Código A.2.1
Empleo de extensiones

```
#include  "Glee.h" //No es necesario incluir gl.h

void draw(void)
{
       ...//Ya existe un contexto de rendering
       if (GLEE_VERSION_1_3)
           glMultiTexCoord2f(...)//Es soportado
       else
           if (GLEE_ARB_multitexture)//Es soportado
              glMultiTexCoord2fARB(..)
           else
           {
              //No es soportada la extensión
           }
}
```

Código A.2.2
Empleo de extensiones, con GLee.

BIBLIOGRAFÍA

[1] David H. Eberly. 3D Game Engine Design. *A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann Publishers, 2001.

[2] Luis Joyanes Aguilar, Ignacio Zahonero Martínez. Estructura de Datos. *Algoritmos, abstracción y objetos*. Mc Graw Hill, 1998.

[3] Stefan Zerbst, Oliver Düvel. 3D Game Engine Programming. Premier Press (Thomson Course technology), 2004.

[4] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. Level of Detail for 3D Graphics. Morgan Kaufmann Publishers, 2003.

[5] Kelly Dempski. Focus On Curves and Surfaces. Premier Press, 2003.

[6] Paul G. Hewitt. Física Conceptual. Trillas, 2003.

[7] Hecht Zajac. Óptica. Addison Wesley, 1997.

[8] John H. Mauldin. Luz, láser y óptica. McGrawHill, 1992.

[9] Henrik Wann Jensen. Realistic Image Synthesis Using Photon Mapping. AK Peters, 2001.

[10] Sebastien St-Laurent. Shaders for game programmers and artists. Premier Press (Thomson Course technology), 2004.

[11] Kevin Hawkins, Dave Astle. Beginning OpenGL, Game Programming. Premier Press, 2001.

[12] Kevin Hawkins, Dave Astle. Beginning OpenGL, Game Programming. Premier Press (Thomson Course technology), 2004.

[13] Charles D. Hansen, Chris R. Jonson. The visualization handbook. Elsevier Academic Press.

Christopher Tremblay. Mathematics for game developers. Premier Press (Thomson Course technology), 2004.

Francisco Javier Ceballos. Enciclopedia del lenguaje C++. Alfaomega Ra-Ma, 2004.

Bruce Eckel. Thinking in C++, Volume 1 & 2. Prentice Hall, Second Edition 2000.

SITIOS WEB

The Cornell Box,Cornell University Program of Computer Graphics
http://www.graphics.cornell.edu/online/box/

Microsoft DirectX Home Page
http://www.microsoft.com/windows/directx/default.mspx

OpenGL Platform & OS Implementations
http://www.opengl.org/documentation/implementations/

Hugues Hoppe's Home Page
http://research.microsoft.com/~hoppe/#pm

Don Gillies Personal Data
http://www.ece.ubc.ca/~gillies/

Corona Home Page
http://corona.sourceforge.net/

Open Source Initiative
http://opensource.org/licenses/zlib-license.html

OpenGL ® Extension Registry
http://www.opengl.org/registry/

Elf-Stone.com The GLee Official Site
http://elf-stone.com/glee.php

OpenGL SDK-GLee
http://www.opengl.org/sdk/libs/GLee/

Technical Note TN2080: Understanding and Detecting OpenGL Functionality
http://developer.apple.com/technotes/tn2002/tn2080.html

SGI- Developer Central Open Source | OpenGL® Sample Implementation
http://oss.sgi.com/projects/ogl-sample/registry/

Realtech-vr, OpenGL Extensions Viewer
http://www.realtech-vr.com/glview/crossplatform.html

Realistic Natural Effect Rendering
http://www.gamedev.net/reference/articles/article2138.asp

Rendering of Natural Waters
http://www.cs.utah.edu/~michael/water/

Reflections on Water Simulation
http://www.darwin3d.com/gamedev/articles/col0100.pdf

Henrik Wann Jensen's Home Page
http://graphics.ucsd.edu/~henrik/